

Владимир Давыдов

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ C++

Рекомендовано УМО вузов Российской Федерации по образованию
в области радиотехники, электроники, биомедицинской техники и автоматизации
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по специальности 210100 "Управление и информатика
в технических системах"

Санкт-Петербург

«БХВ-Петербург»

2005

УДК 681.3.068+800.92C++(075.8)
ББК 32.973.26-018.1я73
Д13

Давыдов В. Г.

Д13 Технологии программирования. C++. — СПб.: БХВ-Петербург, 2005. — 672 с.: ил.

ISBN 5-94157-605-6

Рассмотрены объектно-ориентированная и обобщенная (с использованием стандартной библиотеки) технологии программирования, иллюстрируемые примерами решения классических задач прикладного программирования: сортировок массивов и файлов, транспортной задачи, поиска в таблице, обработки списков и работы с очередями. В качестве базового используется язык программирования высокого уровня C++. Подробно рассматривается стандартная библиотека языка C++. В пособие и компакт-диск включены демонстрационные программы, вопросы и упражнения для самопроверки с ответами, тесты и задания для курсового проектирования, а также справочная информация по C++.

*Для студентов и преподавателей технических вузов
и самообразования*

УДК 681.3.068+800.92C++(075.8)
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. гл. редактора	<i>Людмила Еремеевская</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Екатерина Капалыгина</i>
Компьютерная верстка	<i>Натальи Караваевой</i>
Корректор	<i>Виктория Пиотровская</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Игоря Цырульников</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.12.04.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 54,18.

Тираж 3000 экз. Заказ №

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-605-6

© Давыдов В. Г., 2005
© Оформление, издательство "БХВ-Петербург", 2005

Содержание

ПРЕДИСЛОВИЕ	1
Используемые обозначения	2
 ЧАСТЬ I. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ [5, 8]	3
 Глава 1. Инкапсуляция. Классы	6
1.1. Класс = данные + функции для работы с ними	6
1.2. Инициализация и разрушение объектов. Конструкторы и деструкторы	15
1.3. Статические члены классов	19
1.4. Друзья класса	22
1.5. Перегрузка операций (операторов) для классов	28
1.6. Шаблоны классов	35
1.7. Конструктор копирования. Поверхностное и глубинное копирование	42
1.8. Вопросы и упражнения для самопроверки [4]	55
 Глава 2. Наследование. Иерархия классов	58
2.1. Иерархия наследования классов	58
2.2. Доступ к членам базовых классов	63
2.3. Виртуальные базовые классы	76
2.4. Рекомендации по составу класса. Отличия структур и объединений от классов	81
2.5. Вопросы и упражнения для самопроверки [4]	82
 Глава 3. Полиморфизм	84
3.1. Виртуальные методы классов	84
3.2. Виртуальные деструкторы	93
3.3. Абстрактные методы и классы	96
3.4. Вопросы для самопроверки [4]	101

Глава 4. Области действия и пространства имен [9]	103
4.1. Области действия и время жизни	103
4.2. Пространство имен	105
4.3. Пространство имен стандартной библиотеки	113
4.4. Вопросы для самопроверки [4]	113
Глава 5. Ввод-вывод в языке C++ средствами стандартной библиотеки. Потоковые классы	114
5.1. Ввод-вывод встроенных (стандартных) типов	117
5.2. Состояния предопределенных объектов (потоков). Ошибки потоков	118
5.3. Ввод-вывод типов, определенных пользователем	121
5.4. Форматированный ввод-вывод. Манипуляторы	124
5.5. Методы обмена с потоками	133
5.6. Файловый ввод-вывод	135
5.7. Вопросы для самопроверки [4]	158
Глава 6. Обработка исключительных ситуаций [2, 4, 7, 9]	159
6.1. Общий механизм обработки исключений	160
6.2. Синтаксис исключений	161
6.3. Перехват исключений	162
6.4. Примеры обработки исключений	166
6.5. Вопросы для самопроверки [4]	177
Глава 7. Динамическое определение типа и преобразование типов	178
7.1. Операция <code>const_cast</code>	179
7.2. Преобразование типов во время компиляции (операция <code>static_cast</code>)	180
7.3. Преобразование типов во время выполнения программы (операция <code>dynamic_cast</code>)	183
7.4. Преобразование "на свой страх и риск" (операция <code>reinterpret_cast</code>)	198
7.5. Вопросы для самопроверки	199
ЧАСТЬ II. ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ	201
Глава 8. Сортировка массивов	203
8.1. Спецификация класса для сортировки массива	203
8.2. Сортировка массивов с использованием шаблонных классов	208
8.3. Упражнения для самопроверки	223
Глава 9. Транспортная задача (задача коммивояжера)	224
9.1. Спецификация класса для решения транспортной задачи	224
9.2. Решение транспортной задачи с использованием иерархии обычных классов	230
9.3. Упражнения для самопроверки	240

Глава 10. Поиск в таблице	241
10.1. Спецификация класса для поиска в таблице	241
10.2. Поиск в таблице с использованием шаблонного класса	245
10.3. Упражнения для самопроверки	260
Глава 11. Списки. Очереди и стеки	261
11.1. Бинарные деревья.....	263
11.2. Очереди и стеки.....	264
11.2.1. Универсальная очередь неограниченного размера	266
11.2.2. Универсальная очередь ограниченного размера	276
11.2.3. Динамический стек	289
11.3. Вопросы для самопроверки	295
Глава 12. Сортировка файлов	296
12.1. Сортировка файлов простым слиянием.....	296
12.2. Сортировка файлов естественным слиянием	297
12.2.1. Спецификация класса для сортировки файлов	298
12.2.2. Шаблон классов для сортировки файла естественным слиянием	301
 ЧАСТЬ III. СТАНДАРТНАЯ БИБЛИОТЕКА ЯЗЫКА C++ [9, 10].	
ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ	313
Глава 13. Строки [9, 10]	316
13.1. Создание строк. Конструкторы и деструктор строк.....	318
13.2. Операции над строками.....	320
13.2.1. Присваивание и добавление частей строк.....	322
13.2.2. Преобразования строк.....	324
13.2.3. Поиск подстрок	330
13.2.4. Сравнение частей строк.....	339
13.2.5. Получение характеристик строк.....	343
13.3. Вопросы для самопроверки	346
Глава 14. Строковые потоки.....	347
Глава 15. Контейнерные классы.....	353
15.1. Последовательные контейнеры. Вектор (<i>vector</i>)	358
15.2. Последовательные контейнеры. Вектор логических значений (<i>vector<bool></i>)	376
15.3. Последовательные контейнеры. Двусторонняя очередь (<i>deque</i>).....	378
15.4. Последовательные контейнеры. Список (<i>list</i>)	383
15.5. Адаптеры последовательных контейнеров. Стек (<i>stack</i>)	392
15.6. Адаптеры последовательных контейнеров. Очередь FIFO (<i>queue</i>).....	398
15.7. Адаптеры последовательных контейнеров. Очередь с приоритетами (<i>priority_queue</i>).....	403

15.8. Ассоциативные контейнеры. Пары	406
15.9. Ассоциативные контейнеры. Словари (<i>map</i>)	411
15.10. Ассоциативные контейнеры. Словари с дубликатами (<i>multimap</i>)	420
15.11. Ассоциативные контейнеры. Множества (<i>set</i>)	422
15.12. Ассоциативные контейнеры. Множества с дубликатами (<i>multiset</i>)	427
15.13. Специальные контейнеры. Битовые множества (<i>bitset</i>)	432
15.14. Вопросы и упражнения для самопроверки	438

Глава 16. Итераторы и функциональные объекты 439

16.1. Итераторы	439
16.2. Обратные итераторы	441
16.3. Итераторы вставки	442
16.4. Потокковые итераторы	442
16.5. Функциональные объекты	443
16.6. Вопросы для самопроверки	450

Глава 17. Алгоритмы 451

17.1. Немодифицирующие операции с последовательностями	452
17.2. Модифицирующие операции с последовательностями	459
17.3. Алгоритмы, связанные с сортировкой	476
17.4. Алгоритмы работы с множествами и пирамидами	486
17.5. Другие средства стандартной библиотеки	493
17.5.1. Распределители памяти [10]	493
17.5.2. Средства для численных расчетов	493
17.5.3. Средства поддержки языка	494
17.5.4. Средства диагностики	494
17.5.5. Средства локализации и работы с комплексными числами	496
17.6. Вопросы для самопроверки	496

ПРИЛОЖЕНИЯ 497

Приложение 1. Ответы и решения к вопросам и упражнениям для самопроверки .. 499





П1.1. Глава 1	499
П1.2. Глава 2	511
П1.3. Глава 3	512
П1.4. Глава 4	513
П1.5. Глава 5	514
П1.6. Главы 6 и 7	515
П1.7. Глава 8	516
П1.8. Глава 9	535
П1.9. Глава 10	549
П1.10. Глава 11	566
П1.11. Глава 13	567
П1.12. Глава 15	569
П1.13. Глава 16	577
П1.14. Глава 17	580

Приложение 2. Тесты и программные проекты. Варианты заданий.....	581
П2.1. Классы. Инкапсуляция. Члены класса. Конструкторы и деструктор. Статические члены. Друзья класса. Перегрузка операций. Шаблоны. Варианты тестов	581
П2.2. Классы. Наследование. Полиморфизм. Варианты тестов.....	589
П2.3. Пространства имен. Ввод-вывод в языке C++ средствами стандартной библиотеки. Поточковые классы. Обработка исключительных ситуаций. Варианты тестов	595
П2.4. Прикладное программирование: сортировка массивов, транспортная задача, поиск в таблице, списки, очереди и стеки, сортировка файлов. Варианты тестов	600
П2.5. Стандартная библиотека языка C++. Варианты тестов.....	604
П2.6. Экзаменационное тестирование	620
П2.7. Курсовое проектирование. Варианты заданий.....	621
П2.7.1. Обработка текстов. Варианты заданий.....	622
П2.7.2. Обработка массивов. Варианты заданий.....	627
П2.7.3. Решение геометрических задач. Варианты заданий	630
 Приложение 3. Создание и отладка программного проекта консольного приложения в Microsoft Visual Studio C++ .NET.....	 633
ПЗ.1. Создание программного проекта консольного приложения.....	633
ПЗ.1.1. Создание нового проекта для консольного приложения.....	633
ПЗ.1.2. Создание нового файла и включение его в проект	635
ПЗ.1.3. Добавление в проект существующего файла	636
ПЗ.1.4. Открытие для работы существующего проекта.....	636
ПЗ.2. Отладка программного проекта консольного приложения	637
ПЗ.2.1. Компиляция и компоновка программного проекта. Устранение синтаксических ошибок.....	638
ПЗ.2.2. Отладка программного проекта. Устранение логических (алгоритмических) ошибок	641
ПЗ.2.3. Тестирование программного проекта.....	644
ПЗ.3. Русификация консольных приложений.....	645
 Приложение 4. Прилагаемый компакт-диск	 648
Литература	649
Предметный указатель	651

Предисловие

Учебное пособие обеспечивает курс *"Технологии программирования"* и соответствует разработанной с участием автора примерной программе этого курса, рекомендованной Министерством образования для подготовки бакалавров, магистров и инженеров по направлению 220200 "Автоматизация и управление". Поскольку названный курс является продолжением курса "Программирование и основы алгоритмизации", то и учебное пособие, которое Вы сейчас читаете, является продолжением учебного пособия по курсу "Программирование и основы алгоритмизации" [3], вышедшего в издательстве "Высшая школа". Пособие ориентировано на студентов, но может быть полезным и преподавателям высших учебных заведений, а также программистам, создающим программные продукты с использованием языка высокого уровня C++.

При создании программных продуктов можно использовать различные технологии программирования: *структурную, процедурную* (модульную), *объектно-ориентированную* и *обобщенную* (с использованием стандартной библиотеки). Первые две технологии рассмотрены в [3], а объектно-ориентированная и обобщенная технологии программирования рассматриваются в этой книге. И в [3], и в данной книге в качестве базового рассматривается язык программирования высокого уровня C++. Учебное пособие состоит из трех частей и приложений. В *части I* рассматривается объектно-ориентированная технология программирования. *Часть II* учебного пособия посвящена решению классических задач прикладного программирования, таких как сортировка массивов, транспортная задача (задача коммивояжера), поиск в таблице, обработка списков, работа с очередями и сортировка файлов. Кроме утилитарного значения, рассмотрение решения этих задач является хорошей практикой для освоения технологии объектно-ориентированного программирования. В *части III* подробно, с примерами рассматривается стандартная библиотека языка C++ и основанная на ее использовании технология обобщенного программирования. Здесь же приводятся решения некоторых из классических задач прикладного программирования с использованием обобщенной технологии программирования. В *приложениях* приведены следующие полезные сведения:

-  ответы к вопросам и упражнениям для самопроверки;
-  варианты тестов и заданий на выполнение программных проектов;
-  методика создания и отладки программного проекта консольного приложения в интегрированной среде программирования Microsoft Visual Studio C++ .NET;
-  информация о способах русификации экранного вывода консольных приложений.

Для удобства преподавателей и студентов пособие содержит по 20—60 вариантов контрольных заданий по основным разделам курса, заданий на выполнение программных проектов и пример тестовых экзаменационных вопросов. В пособие включены снабженные ответами вопросы и упражнения для самопроверки, что позволяет использовать его и для *самостоятельного* изучения материала. К учебному пособию прилагается компакт-диск, содержащий файл с вариантами контрольных заданий по основным разделам курса, заданий на выполнение программных проектов, примером тестовых экзаменационных вопросов, файл с перечнем заголовочных файлов стандартной библиотеки языка C++ и сведениями о их назначении, перечнем констант, макросов, типов данных и функций стандартной библиотеки и файл с описанием стандарта языка C++, а также полные исходные тексты демонстрационных программ автора, имеющихся в учебном пособии. Из сказанного со всей очевидностью следует, что учебное пособие поддерживает не только лекционную часть курса, но и полностью поддерживает практические занятия.

Используемые обозначения

Исходные тексты программ и результаты их работы, приводимые в книге, для удобства читателей печатаются с использованием моношириного шрифта *Courier New*, а служебные слова выделяются **полужирным шрифтом**. Таким же образом служебные слова оформляются в тексте книги.

Курсивом в тексте выделяются определяющие вхождения новых понятий, а также отдельные слова или выражения, на которые следует обратить внимание.

Имена файлов пишутся без кавычек и без выделения.

Кроме шрифтовых выделений, используется три типа специальных абзацев: советы, замечания и примечания.

СОВЕТ

Самым тщательным образом изучите эти примеры — в них в виде комментариев изложена очень важная информация.

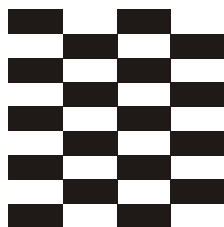
ЗАМЕЧАНИЕ

Спецификаторы доступа `private`: и `public`: могут быть использованы в определении класса многократно и в произвольном порядке, хотя делать это не рекомендуется. В дальнейшем, в блоке класса мы будем вначале перечислять данные в порядке увеличения их доступности, а затем — методы в порядке убывания их доступности, как это было сделано в приведенном примере.

ПРИМЕЧАНИЕ

Как же быть с перегрузкой арифметических операций? Должны ли соответствующие перегружающие функции возвращать значение объекта или ссылку на объект? Для решения этого вопроса нужно руководствоваться следующим правилом. Если важнее быстрое действие приложения, то перегружающие арифметические операции функции должны возвращать ссылку на объект. И наоборот, если требуется минимизировать память, требуемую приложению, то перегружающие арифметические операции функции должны возвращать значение объекта.

Ваши отзывы об учебном пособии, конструктивные замечания и критику направляйте по адресу: davydov@aivt.ftk.spbstu.ru.



Часть I

Объектно-ориентированное программирование [5, 8]

Глава 1. Инкапсуляция. Классы

Глава 2. Наследование. Иерархия классов

Глава 3. Полиморфизм

Глава 4. Области действия и пространства имен [9]

**Глава 5. Ввод-вывод в языке C++ средствами
стандартной библиотеки. Потокные классы**

Глава 6. Обработка исключительных ситуаций [2, 4, 7, 9]

**Глава 7. Динамическое определение типа
и преобразование типов**

Технология объектно-ориентированного программирования (ООП) является дальнейшим развитием идей структурного и процедурного (модульного) программирования, рассмотренных в [3] и в другой многочисленной литературе. Центральным понятием ООП является *класс*. Класс является типом данных, определяемым пользователем. В классе задаются свойства и поведение какого-либо объекта в виде данных и функций (методов) для их обработки. Создаваемый тип данных (класс) обладает практически теми же свойствами, что и стандартные типы: задает внутреннее представление данных в памяти компьютера, множество их значений, а также операции и функции (методы), применяемые к ним. Таким образом, ООП является новым подходом, предполагает отказ от старых стереотипов и переход от мышления в терминах данных и функций (процедур) к мышлению в терминах объектов, наделенных определенными свойствами и поведением.

Язык C++ относится к языкам объектно-ориентированного программирования и поэтому может рассматриваться как язык нового поколения. ООП характеризуют три основных свойства:

- *Инкапсуляция* (encapsulation) — слияние данных и функций, работающих с этими данными, в одном объекте (классе, структуре) одновременно со скрытием ненужной для использования объекта информации. Указанный подход в усеченном виде применялся в структурном и модульном программировании, а в ООП получил свое логическое завершение. Инкапсуляция повышает степень абстракции программы. Это означает, что информация о данных и реализации работающих с ними функций не требуется для написания программы, использующей класс. Класс используется только через его интерфейс, определяемый прототипами функций-членов класса. Благодаря этому можно изменить реализацию класса без модификации основной программы, использующей класс, если интерфейс класса остался неизменным.

Инкапсуляция позволяет использовать класс в другом программном окружении и при этом не испортить не принадлежащие классу области памяти. На основе инкапсуляции строятся библиотеки классов, которые применяются во многих программах.

- *Наследование* — порождение нового объекта (класса, структуры) на основе уже имеющегося. Новый класс (структура) наследует данные и функции порождающего класса (структуры), но может добавлять новые данные и функции. Таким образом, механизм наследования позволяет порождать иерархию классов с общими данными и функциями.

При наследовании свойства порождающего класса повторно не описываются, что сокращает объем программы. Выделение общих черт различных классов в общий класс-предок также является мощным механизмом абстракции, который помогает справиться со сложностью решаемой задачи.

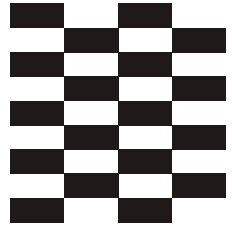
Иерархию классов можно представить в виде древовидной структуры, в которой общие классы располагаются ближе к корню, а специализированные — на ветвях и листьях.

- *Полиморфизм* — перегрузка операций, функций, шаблоны классов. Полиморфизм, с одной стороны, позволяет одни и те же операции и функции использовать для различных типов объектов, а с другой стороны, позволяет в производном классе переопределять некоторые из функций базового класса. Такие функции в ООП

называют виртуальными функциями. Механизм виртуальных функций обеспечивает объектам высокую гибкость и универсальность. Таким образом, можно заключить, что полиморфизм обеспечивает возможность использования в различных классах иерархии одного и то же имени функции для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы.

Базируясь на ООП, программа представляется в терминах состояния и поведения объектов, более близких к рассматриваемой предметной области. Поэтому она легче читается и воспринимается. Однако проектирование объектно-ориентированной программы является более трудоемким из-за необходимости разработки иерархии классов и сложности освоения средств ООП.

Глава 1



Инкапсуляция. Классы

Глава посвящена рассмотрению совокупности понятий, определяющих инкапсуляцию как один из основных принципов ООП.

1.1. Класс = данные + функции для работы с ними

Определение *класса* вводит новый тип и специфицирует *члены-данные*, необходимые для представления объектов определенного типа, и операции для работы с этими объектами, т. е. *функции-члены* класса (методы класса). Члены-данные класса определяют *состояние* объекта, а методы обуславливают *поведение* объекта класса.

Листинг 1.1. Пример определения класса

```
class ANYCLASS
{
    // Члены-данные

private:

    int    x;
    double y, z;
    // ...

public:

    char  ch, ch1;
    // ...

    // Методы

public:

    int f3( int, int );
    int GetX( void )
    {
```

```
        return x;
    }
    // ...

private:

    void f1( void );
    int f2( int );
    // ...

};
```

В листинге 1.1 *спецификаторы доступа* `private:` и `public:` определяют режим доступа к членам класса: спецификатор `private:` делает члены класса, определяемые после него, закрытыми, а спецификатор `public:` — наоборот, открытыми.

ЗАМЕЧАНИЕ

Следует отметить, что спецификаторы доступа `private:` и `public:` могут быть использованы в определении класса многократно и в произвольном порядке, хотя делать это не рекомендуется. В дальнейшем, в блоке класса мы будем вначале перечислять данные в порядке увеличения их доступности, а затем — методы в порядке убывания их доступности, как это было сделано в приведенном примере.

К *закрытым* (`private:`) членам класса (как к методам, так и к данным) имеют доступ только методы данного класса, а также функции-друзья класса (о них речь пойдет позже). Открытые (`public:`) члены класса доступны любым функциям. Они предназначены для общения объектов класса с программой, в которой они существуют.

ЗАМЕЧАНИЯ

Проектируя класс, необходимо тщательно продумать, какие его члены сделать открытыми, а какие — закрытыми. Объекты спроектированного класса в идеале должны быть подобны хорошему автомобилю: водителю нужно знать, как крутить баранку, когда нажимать на тормоз, а когда на газ, а что там под капотом — дело изготовителя.

При определении класса с помощью ключевого слова `class` его члены будут считаться по умолчанию закрытыми. В языке C++ для определения класса допустимо также использовать ключевые слова `struct` и `union`. В случае `struct` члены класса станут по умолчанию открытыми, но их можно закрыть при помощи спецификатора `private:`. При использовании `union` члены класса могут быть только общедоступными.

В большинстве случаев определение класса не локализовано в блоке, и областью действия имени класса является весь файл. Если же класс определен внутри блока, то его называют *локальным* классом, его область действия — блок и на такой класс накладываются два ограничения, о которых будет сказано позже.

Определения методов класса могут находиться непосредственно внутри определения класса и в таком случае они автоматически считаются *подставляемыми*. Как указывалось в [3], подставляемые функции следует использовать, если функция очень простая и короткая.

Обычно, с целью повышения защищенности, члены-данные класса не делают общедоступными и к ним нельзя непосредственно обратиться из произвольной про-

граммной среды. Тем не менее, с ними можно работать и из произвольной программной среды, используя общедоступные методы класса. Для этого в классы часто включают очень короткие открытые методы. Для рассмотренного ранее примера таким методом является:

```
int GetX( void )
{
    return x;
}
```

Открытый метод `GetX()` возвращает значение закрытого члена-данного `x`, что делает его доступным из произвольной программной среды. Для лаконичности целесообразно "тощее" тело таких методов помещать внутрь определения класса, сделав их для повышения эффективности подставляемыми.

Альтернативой является определение метода класса где-либо в другом месте подобно "обычной" функции. С помощью оператора разрешения области видимости `::` компилятору сообщается, к какому классу принадлежит данный определяемый метод, например:

```
// Определяется функция f2 - член класса ANYCLASS
int ANYCLASS :: f2( int x )
{
    ...                               // Тело метода
}
```

Определенный таким образом метод тоже можно сделать подставляемым, используя ключевое слово `inline`:

```
inline int ANYCLASS :: f2( int x )
{
    ...                               // Тело метода
}
```

Определение метода класса необходимо поместить *перед* использованием этого метода.

ЗАМЕЧАНИЯ

Отметим, что для *локальных классов* методы класса должны быть подставляемыми и их определения должны обязательно помещаться в блоке определения класса. Это является первым из упомянутых ранее ограничений для локальных классов.

Обратите внимание, что определение класса не создает объектов данного класса. Объекты создаются только путем их определения, например:

```
ANYCLASS      Obj1, Obj2, Obj3, ObjArray[ 10 ];
```

Размер объекта класса в памяти определяется *суммой размеров членов-данных класса*. Методы класса *не занимают* место в области памяти, выделенной для объекта класса.

При работе с программными проектами, состоящими из нескольких файлов, определение класса должно присутствовать во всех файлах, где используются объекты данного класса или определяются его методы. Поэтому целесообразно помещать определение класса в заголовочный файл, включаемый при помощи директивы `#include` в те файлы, в которых оно необходимо. Если определение метода класса находится вне определения класса, то он обязательно должен быть определен в тех файлах, где используется (и, конечно же, до его вызова). Определение такого метода также лучше поместить в заголовочный файл вместе с определением класса.

Доступ к открытым членам объекта некоторого класса осуществляется при помощи операторов прямого "." и косвенного "->" выбора.

Проиллюстрируем все сказанное ранее содержательным примером (листинги 1.2—1.4), который был рассмотрен в [3].

Листинг 1.2. Файл LINSР.H

```
/*
    Включаемый файл для LINSР.CPP
    Содержит стандартные includеемые файлы, определение структуры для элемента списка,
    определение класса для работы с однонаправленным линейным списком и методов класса.
    Алгоритмы выполнения операций над линейным списком подробно рассмотрены в [3].
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предотвращение многократного включения данного файла
#ifdef __LINSР_H
#define __LINSР_H

#include <stdio.h>    // Для ввода-вывода
#include <stdlib.h>    // Для exit ( )

// Структура для элемента списка
struct ELEM
{
    char    ch;        // Данные (символ)
    ELEM    *next;     // Указатель на следующий элемент
};

// Определение класса
class LINSР
{
    // Член класса может быть:
    // private: (закрытый) - доступный только внутри методов и
    // друзей этого класса;
    // protected: (защищенный) - доступный только внутри методов,
    // друзей этого класса и производных от него классов;
    // public: (открытый) - доступный из произвольной программной
    // среды

    // Данные

protected:

    ELEM    *start;    // Указатель на начало списка
```

```
// Методы
```

```
public:
```

```
// Конструктор (имя совпадает с именем класса). Конструктор
// всегда вызывается при создании объекта класса с
// конструктором и не имеет возвращаемого значения
// (подставляемый метод)
LINSF( void )
{
    start = NULL; // Вначале список пуст
    printf( "\n The designer is called(caused) " );
}

// Деструктор (имя совпадает с именем класса, но перед ним
// ставится '~'). Деструктор автоматически вызывается, когда:
// - объект уходит из области действия;
// - завершается программа;
// - используется операция delete для объектов, размещенных
// операцией new;
// - явно используется вызов с полным именем деструктора.
// Деструктор не имеет параметров и возвращаемого значения!!!
~LINSF( void )
{
    // Подставляемый метод
    Del_all( ); // Удаление всего списка
    printf( "\n Destructor has completed operation \n" );
}

// Добавить элемент в конец списка
void Add_end( const char &c );

// Добавить элемент в начало списка: подставляемый метод - см.
// определение метода далее
void Add_beg( const char &c );

// Удалить первый элемент списка (подставляемый метод)
void Del_beg( void )
{
    ELEM
    *del; // Указатель на удаляемый элемент

    if( !start )
    {
        printf( "\n The list is empty. There is "
            "nothing to delete " );
        return;
    }
}
```

```

    }
    del = start; // Подготовка первого элемента для удаления
    // start сдвигается на второй элемент
    start = del->next;
    delete del; // Удаление первого элемента

    return;
}

// Печать содержимого списка на экран
void Print_ls( void );

// Удалить весь список
void Del_all( void );

// Удаление последнего элемента списка
void Del_end( void );

};

//*****
// Добавление элемента в конец списка
void LINSR :: Add_end(
    const char
        &c )    // Данные добавляемого элемента
{
    // Указатель на новый (добавляемый) элемент списка
    ELEM *temp,
        *cur;    // Указатель на текущий элемент

    temp = new ELEM; // Динамическое размещение элемента
    if( !temp )
    {
        printf( "\n The unit of the list is not placed " );
        exit( 1 );
    }
    temp->ch = c;    // Занесение данных
    // Новый элемент является последним
    temp->next = NULL;
    if( !start)
        // Новый список (пустой)
        start = temp; // Указатель на начало списка
    else
    {
        // Проходим весь список от начала, пока текущий элемент

```

```

        // не станет последним
        cur = start;
        while( cur->next )
            // Продвижение по списку
            cur = cur->next;
        // Ссылка последнего элемента на новый, добавляемый в конец
        // списка
        cur->next = temp;
    }

    return;
}

//*****
// Добавление элемента в начало списка (определение подставляемого
// метода вне определения класса)
inline void LINSР :: Add_beg(
    const char
        &c )        // Данные добавляемого элемента
{
    // Указатель на новый (добавляемый) элемент списка
    ELEM *temp;

    temp = new ELEM; // Динамическое размещение элемента
    if( !temp )
    {
        printf( "\n The unit of the list is not placed " );
        exit( 1 );
    }
    temp->ch = c;      // Занесение данных
    // Новый элемент ссылается на бывший первый
    temp->next = start;
    start = temp;      // Новый элемент становится первым

    return;
}

//*****
// Удаление последнего элемента списка
void LINSР :: Del_end( void )
{
    ELEM *prev,      // Указатель на предпоследний элемент
        *end;        // Указатель на последний элемент

    if( !start )
    {

```

```

    printf( "\n The list is empty. There is nothing to "
           "delete " );

    return;
}

if( !start->next )
{
    // В ЛС один элемент

    delete start; start = NULL;
    return;
}

end = start;      // Указатель на начало списка
// Поиск последнего (и предпоследнего) элемента
while( end->next )
{
    prev = end;
    // Продвижение по списку
    end = end->next;
}

delete end;       // Удаление последнего элемента
// Бывший предпоследний элемент становится последним
prev->next = NULL;

return;
}

//*****
// Печать содержимого списка на терминал
void LINSP :: Print_ls( void )
{
    ELEM *prn;     // Указатель на печатаемый элемент

    if( !start )
    {
        printf( "\n The list is empty. There is nothing to print "
               "out " );

        return;
    }

    prn = start;   // Указатель на начало списка
    printf( "\n" );
    while( prn )   // До конца списка
    {
        // Печать данных (символа) элемента
        printf( "%c", prn->ch );
        // Продвижение по списку
        prn = prn->next;
    }
}

```

```

    return;
}

//*****
// Удаление всего списка
void LINSР :: Del_all( void )
{
    if( !start )
    {
        printf( "\n The list is empty. There is nothing to"
                " delete " );
        return;
    }
    while( start )
        Del_beg( ); // Удаление первого элемента

    return;
}

#endif // Конец файла LINSР.H

```

Листинг 1.3. Файл LINSР.CPP

```

/*
    Работа с однонаправленным линейным списком с использованием класса,
    определенного в файле LINSР.H
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Стандартные заголовочные файлы и определение класса LINSР для работы
// с линейным списком
#include "linsр.h"

// Тестирование класса
int main( void ) // Возвращает 0 при успехе
{
    LINSР ls; // Определение ls классом LINSР

    ls.Add_end( 'A' ); // Добавление в конец списка 'A'
    ls.Add_beg( 'B' ); // Добавление в начало списка 'B'
    ls.Print_ls( ); // Печать содержимого списка
    ls.Del_all( ); // Удаление всего списка
    ls.Del_end( ); // Удаление последнего элемента
    ls.Del_beg( ); // Удаление первого элемента
    ls.Print_ls( ); // Печать содержимого списка
}

```

```

ls.Add_end( 'C' );    // Добавление в конец списка 'C'
ls.Add_end( 'D' );    // Добавление в конец списка 'D'
ls.Add_beg( 'F' );    // Добавление в начало списка 'F'
ls.Add_beg( 'G' );    // Добавление в начало списка 'G'
ls.Print_ls( );       // Печать содержимого списка
ls.Del_end( );        // Удаление последнего элемента
ls.Del_beg( );        // Удаление первого элемента
ls.Print_ls( );       // Печать содержимого списка

return 0;
}

```

Листинг 1.4. Результаты работы программы

```

The designer is called(caused)
БА
The list is empty. There is nothing to delete
The list is empty. There is nothing to delete
The list is empty. There is nothing to print out
GFCD
FC
Destructor has completed operation
Press any key to continue

```

Советуем внимательно изучить этот пример. К его обсуждению мы еще будем возвращаться.

1.2. Инициализация и разрушение объектов. Конструкторы и деструкторы

В языке C++ автоматически вызываемые при создании и разрушении объектов специальные методы классов, выполняющие задачи по инициализации и уничтожению объектов, называются соответственно *конструкторами* и *деструкторами* (см. листинги 1.2—1.4).

Как следует из приведенного примера, конструктор формально отличается от обычных методов класса тем, что его имя совпадает с именем класса. Имя деструктора образуется путем добавления символа "~" (тильда) в начало имени класса. Конструктор вызывается автоматически *только* при создании объекта соответствующего типа. Деструктор же автоматически вызывается при разрушении объектов, но при необходимости его можно вызывать явно, подобно обыкновенному методу класса. Конструктор/конструкторы (конструкторов, как мы узнаем далее, может быть несколько) и деструктор являются открытыми методами класса. Конструктор в общем случае может иметь параметры, а деструктор нет. Конструктор и деструктор не имеют возвращаемых значений, но в отличие от обычных функций перед их именами служебное слово `void` не используется. В конструкторе и деструкторе не следует использовать также оператор `return`.

Довольно "объемный" пример, приведенный в разд. 1.1, поможет читателям разобраться в материале, связанном с инициализацией и разрушением объектов класса, имеющего конструктор и деструктор. Это очень важный материал, поэтому для работы с примером не пожалейте времени и будьте особенно внимательны.

Сделаем несколько важных замечаний, которые будут полезны нам в будущем.

- ❑ В классе можно использовать несколько конструкторов — каждый конструктор со своим, отличным от других, списком параметров. Если конструктор не имеет параметров, как в рассмотренном ранее примере, то его называют *конструктором умолчания*. Деструктор же всегда один и не имеет параметров.
- ❑ Объект класса, который имеет только открытые члены и не имеет конструктора (и только в этом случае!), может быть инициализирован подобно обычной структурной переменной при помощи списка значений.
- ❑ Если ни один из конструкторов класса не является *открытой* функцией-членом класса, то объекты такого класса не могут быть созданы. Забегая вперед, скажем, что такие классы не бесполезны: они могут быть базовыми классами для других классов (наследование).

При создании объекта класса подходящий для этого конструктор вызывается автоматически. При этом выполнение конструктора происходит в два этапа. На первом этапе выполняется инициализация конструктора, а на втором — тело конструктора.

Инициализация членов-данных объекта класса может быть выполнена как во время инициализации конструктора, так и во время его выполнения. Проиллюстрируем сказанное простым примером (листинг 1.5).

Листинг 1.5. Инициализация членов-данных класса во время выполнения конструктора

```
class EMPLOYEE
{
    // Данные

private:

    unsigned int
        Age;           // Возраст
    unsigned int
        Salary;        // Зарплата

    // Методы

public:

    // Обычный конструктор - инициализация объекта: подставляемый метод
    EMPLOYEE( unsigned int age, unsigned int salary )
    {
        Age = age; Salary = salary;
```

```
    }

    // Получение значения возраста: подставляемый метод
    unsigned int GetAge( void ) const
    {
        return Age;
    }

    // Получение значения зарплаты: подставляемый метод
    unsigned int GetSalary( void ) const
    {
        return Salary;
    }

};
```

Но логически правильнее, а зачастую и эффективнее, инициализировать члены-данные класса во время инициализации конструктора (листинг 1.6).

Листинг 1.6. Инициализация членов-данных класса во время инициализации конструктора

```
class EMPLOYEE
{
    // Данные

private:

    unsigned int
        Age;          // Возраст
    unsigned int
        Salary;       // Зарплата

    // Методы

public:

    // Обычный конструктор - инициализация объекта: подставляемый метод
    EMPLOYEE( unsigned int age, unsigned int salary ):
        // Инициализация членов-данных параметрами конструктора
        Age( age ), Salary( salary )
    { }

    // Получение значения возраста: подставляемый метод
    unsigned int GetAge( void ) const
```

```

{
    return Age;
}

// Получение значения зарплаты: подставляемый метод
unsigned int GetSalary( void ) const
{
    return Salary;
}

};

```

Приведенный пример показывает, что если инициализация членов-данных класса выполняется при инициализации конструктора, то строение конструктора имеет следующий вид: после круглой скобки, закрывающей список параметров конструктора, ставится двоеточие; затем перечисляются имена членов-данных класса. Пара круглых скобок со значением за именем члена-данного используется для его инициализации. Если инициализируются сразу несколько членов-данных, то они должны быть разделены запятыми.

Аналогично, для примера, приведенного в *разд. 1.1*, конструктор умолчания можно оформить так, как показано в листинге 1.7.

Листинг 1.7. Конструктор умолчания

```

public:

    // Конструктор (имя совпадает с именем класса).
    // Конструктор всегда вызывается при создании объекта класса с
    // конструктором и не имеет возвращаемого значения
    // (подставляемый метод)
    /*LINSР( void )
    {
        start = NULL; // Вначале список пуст
        printf( "\n The designer is called(caused) " );
    }*/
    LINSР( void ):
    {
        start( NULL ) // Вначале список пуст
    {
        printf( "\n The designer is called(caused) " );
    }
}

```

В этом фрагменте конструктор, выполняющий инициализацию `start` во время своего выполнения, закомментирован. Оба варианта конструктора дают одинаковые конечные результаты.

1.3. Статические члены классов

Как сделать, чтобы некоторые члены класса были общими для всех одновременно существующих объектов данного класса?

Достаточно описать такой член класса как статический (**static**).

Особенности статических членов класса проиллюстрируем на примере (листинги 1.8, 1.9).

Листинг 1.8. Файл StaticMemberClass.cpp

```

/*
    Статические члены класса
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdio.h>          // Для ввода-вывода
#include <string.h>         // Для работы со строками

//*****
// Класс со статическими членами для обработки строк
class WSTRING              // Work STRINGS: обработка строк
{
    // Члены-данные класса (по умолчанию закрытые - private:)

    char    *s;            // Указатель на текстовую строку
    int     len;           // Длина строки
    int     my_number;     // Личный номер строки
    static int counter;    // Общее число строк (статический член-данные
                          // класса)

public:                  // Открытые методы

    // Подставляемый метод класса: увеличивает счетчик строк на единицу,
    // присваивает строке очередной номер, заполняет содержимым и
    // определяет ее длину
    void Assign(
        char    *String ) // Указатель заполняющей строки
    {
        my_number = ++counter; s = String; len = strlen( s );
        return;
    }

    // Подставляемый статический метод класса: возвращает число
    // созданных строк
    static int NumStrings( void )

```

```

{
    return counter;
}

/*----- Особенность 1 -----
    Статические методы могут работать только со статическими
    членами-данными (см. NumStrings). Чтобы сделать в них доступными
    другие члены-данные, нужно в статические методы с помощью
    параметра явно передавать адрес объекта
*/

void InfoString( void ) const;
// Модификатор const гарантирует, что данный метод не изменит
// значения членов-данных объекта, для которого он вызван
};

/*----- Особенность 2 -----
    Статические члены-данные и методы являются глобальными и, в отличие от
    обычных статических переменных и функций, доступны во всех файлах
    многофайлового проекта, содержащих описание данного класса.
----- Особенность 3 -----
    Статические члены-данные, как и обыкновенные глобальные переменные,
    обязательно необходимо определить в одном из файлов проекта. Обратите
    внимание (см. далее), что в определении слово static не используется
*/

// Определение и инициализация. В сегменте данных WSTRING :: counter
// размещается в одном экземпляре
int WSTRING :: counter = 0;

//*****
// Вывод на экран информации о строке
void WSTRING :: InfoString( void ) const
{
    const char *fmt = "\n I - string! My datas:"
                      "\n My personal number %d"
                      "\n My message %s"
                      "\n Length of the message "
                      "%d characters \n";
    printf( fmt, my_number, s, len );

    return;
}

//*****
// Тестирование класса

```

```

int main( void )           // Возвращает 0 при успехе
{

    /*----- Особенность 4 -----
        Статические члены-данные и функции-члены доступны и до создания
        хотя бы одного объекта. Обратиться к ним можно с помощью
        оператора ::
    */

    printf( "\n In total is created of strings(lines) - %d",
            WSTRING :: NumStrings( ) );

    // Создаем три строки
    WSTRING    str1, str2, str3;

    // Инициализируем созданные строки
    str1.Assign( "\n Hi!" );
    str2.Assign( "\n OOP - excellent(different) tool!" );
    str3.Assign( "\n We know how to consider(count) "
                "ourselves!" );

    // Печатаем информацию о созданных строках
    printf( "\n In total is created of strings(lines) - %d",
            WSTRING :: NumStrings( ) );
    // Можно было бы вызвать статическую функцию-член и так:
    //    str1.NumStrings( ), или str2.NumStrings( ), или
    //    str3.NumStrings( ). Все формы вызова эквивалентны
    str1.InfoString( );
    str2.InfoString( );
    str3.InfoString( );

    return 0;
}

```

Листинг 1.9. Результаты выполнения программы

```

In total is created of strings(lines) - 0
In total is created of strings(lines) - 3
I - string! My datas:
My personal number 1
My message
Hi!
Length of the message 5 characters

I - string! My datas:
My personal number 2

```

```

My message
OOP - excellent(different) tool!
Length of the message 34 characters

I - string! My datas:
My personal number 3
My message
We know how to consider(count) ourselves!
Length of the message 43 characters
Press any key to continue

```

Отметим, что этот пример также является очень важным и его также следует внимательно изучить, обратив внимание на описание особенностей работы со статическими членами класса, которые указаны в тексте примера в виде комментариев.

ЗАМЕЧАНИЕ

Конечно же, статические члены не могут принадлежать локальным классам, поскольку имя такого класса не видно за пределами блока и определение статических членов-данных класса становится невозможным. Это второе из ограничений, указанных ранее в *разд. 1.1* для локальных классов.

1.4. Друзья класса

В некоторых ситуациях желательно, чтобы функция, не являющаяся членом класса, тем не менее, имела доступ к закрытым членам этого класса. Язык C++ позволяет это сделать, если функцию объявить *другом класса* или если объявить класс, членом которого она является, другом класса с помощью служебного слова **friend**. Эту возможность иллюстрирует приводимый далее фрагмент программы, который следует внимательно изучить (листинг 1.10).

Листинг 1.10. Файл Friend.cpp

```

/*
    Дружественные функции и классы
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предварительное объявление класса. Используется для одного из двух
// классов, которые ссылаются друг на друга
class MYCLASS1;

//*****
// Класс, использующий дружественную функцию и дружественный класс
class MYCLASS
{
    // ...

```

```

private:

    int    j;           // Закрытый член-данное класса
    // ...

public:

    // Дружественная функция класса MYCLASS. В ней доступны закрытые
    // члены класса MYCLASS
    friend void IncJ( MYCLASS & );

    // Класс, дружественный классу MYCLASS. Благодаря
    // предварительному объявлению компилятор "знает", что MYCLASS1
    // - имя класса, который будет определен позднее. Методам
    // MYCLASS1 также доступны закрытые члены класса MYCLASS
    friend MYCLASS1;

    // Важное замечание! Спецификаторы доступа private: и public:
    // не оказывают влияния на доступность членов класса MYCLASS
    // в дружественной функции IncJ и в методах дружественного
    // класса MYCLASS1
    // ...
};

/*****
// Дружественный класс
class MYCLASS1
{
    // ...
    int    j;
    // ...

    public:

        void MakeJEqual( const MYCLASS & );
        // ...
};

/*****
// Определение дружественной функции IncJ (INcrement J). Все, к чему
// обязывает дружба MYCLASS и IncJ - это то, что класс разрешает
// функции пользоваться своими членами. Функция увеличивает значение
// член-данного j объекта obj на 1
void IncJ( MYCLASS &obj )
{

```

```

    obj.j++;

    return;
}

//*****
// MakeJEqual является методом дружественного класса MYCLASS1 и, поэтому,
// имеет доступ к закрытым членам объектов класса MYCLASS. Метод
// присваивает члену j объекта класса MYCLASS1 значение члена j объекта
// obj класса MYCLASS
void MYCLASS1 :: MakeJEqual( const MYCLASS &obj )
{
    j = obj.j;

    return;
}

//*****
// Тестирование
int main( void )          // Возвращает 0 при успехе
{
    // ...
    MYCLASS    ob;
    MYCLASS1   ob1;
    // ...
    // Функция IncJ не является членом класса MYCLASS и поэтому не может
    // быть вызвана как ob.IncJ( ). Скрытый аргумент - адрес объекта
    // - по этой причине ей не передается. В данном случае для
    // изменения члена j объекта ob функция в явном виде получает
    // ссылку на объект ob
    IncJ( ob );

    // Присвоим ob1.j значение ob.j
    ob1.MakeJEqual( ob );
    // ...

    return 0;
}

```

ЗАМЕЧАНИЕ

Не следует, без необходимости, стремиться к использованию дружественных функций и классов, так как это противоречит концепции инкапсуляции. Их следует использовать только тогда, когда без этого обойтись нельзя. В соответствии с ходом рассмотрения материала, такие ситуации будут рассмотрены далее.

Рассмотрим еще один пример, демонстрирующий ряд важных, интересных особенностей использования дружественных функций и классов (листинги 1.11, 1.12).

Листинг 1.11. Файл FRENД_STL.CPP

```

/*
    Демонстрация возможностей использования дружественных функций и
    классов.
    Т. Сидорина, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdio.h>          // Для ввода-вывода

// *****
// Предварительное объявление класса: если два класса ссылаются друг на
// друга, то один из классов надо предварительно кратко объявить.
// Тогда развернутое определение класса можно будет поместить позже, в
// подходящем месте
class CP;                  // Класс демонстрирует различные способы
                           // использования дружественных функций и
                           // классов

// *****
// Объявление класса, который ссылается на класс CP
class FFCP
{
    // Методы

public:

    // Этот метод в классе CP объявлен как дружественный
    void crff( CP &op );
};

// *****
// Развернутое объявление класса, демонстрирующего различные способы
// использования дружественных функций и классов
class CP
{
    // Данные

private:

    int      x;

    // Дружественные функции и классы - для них спецификатор доступа
    // значения не имеет

    // Дружественной функцией может быть и главная функция приложения

```

```

friend int main( void );
// Дружественной функцией может быть обычная внешняя функция
friend void f( CP &op );
// Дружественным может быть и класс. Тогда все его методы будут
// дружественными
friend class FCP;
// Дружественным может быть и отдельный метод класса
friend void FFCP :: cpff( CP &op );
};

// *****
// Определение класса, дружественного классу CP
class FCP
{
    // Методы

public:

    // Этот метод, как член дружественного класса, является дружественным
    // классу CP
    void cpf( CP &op )
    {
        op.x = 9;

        return;
    }
};

// *****
// Реализация класса FFCP
// Определение метода, дружественного классу CP
void FFCP :: cpff( CP &op )
{
    op.x = 11;

    return;
}

// *****
// Определение обычной внешней функции, дружественной классу CP
void f( CP &op )
{
    op.x = 7;

    return;
}

```

```

}

// *****
int main( void )           // Возвращает 0 при успехе
{
    // Главная функция является дружественной классу CP, поэтому в ней
    // доступны все члены объекта c1
    CP          c1;

    c1.x = 5;
    printf( "c1.x=%i \n", c1.x );

    // Объект c1 доступен в дружественной внешней функции
    //   f( )
    f( c1 );
    printf( "c1.x=%i \n", c1.x );

    // Класс FCP является дружественным классу CP. Поэтому методам
    //   объекта c2 доступны все члены объекта c1
    FCP          c2;

    c2.cpf( c1 );
    printf( "c1.x=%i \n", c1.x );

    // Класс FFCP содержит метод cpff(), дружественный классу CP.
    //   Поэтому метод cpff() имеет полный доступ к объекту c1
    FFCP          c3;

    c3.cpff( c1 );
    printf( "c1.x=%i \n", c1.x );

    return 0;
}

```

Листинг 1.12. Информация, выдающаяся на экран при выполнении программы

```

c1.x=5
c1.x=7
c1.x=9
c1.x=11
Press any key to continue

```

Внимательно изучите этот пример — это очень важно. Еще раз обратите внимание на то, что дружественная функция не получает автоматически скрытый указатель **this** на объект класса. Поэтому в ее вызове нужно обязательно указывать в качестве аргумента объект класса, с которым она должна работать. Обратите также внимание на способ работы с дружественной главной функцией консольного приложения.

1.5. Перегрузка операций (операторов) для классов

В [3], в *разд. 12.9*, говорилось, что большинство операций @ языка C++ может быть перегружено для работы с объектами пользовательских типов. Такая возможность существует и для объектов-классов. Если некоторый оператор @ языка C++ перегружается для работы с объектами каких-либо классов, то разумно и целесообразно сделать соответствующую функцию-операцию `operator@` либо членом класса, либо дружественной функцией (не желательно), либо обычной функцией. В двух последних случаях функция-операция должна принимать хотя бы один аргумент, имеющий тип класса, указателя или ссылки на класс. *Если внешняя функция-операция не является дружественной классу, то следует учитывать доступность членов соответствующего класса.* Название функции-операции состоит из служебного слова `operator`, за которым следует знак переопределяемой операции.

Действуют следующие правила перегрузки операций:

1. Можно перегружать любые операции, существующие в языке C++, за исключением `(".", ".*", "?:", ":", "sizeof`. При этом действие большинства операций переопределяется так, чтобы при использовании с объектами конкретного класса они выполняли заданные функции. В результате этого появляется возможность использовать собственные типы данных точно так же, как стандартные. Обозначения собственных операций вводить *нельзя*.
2. Не допускается перегрузка операций для встроенных (стандартных) типов данных.
3. Нельзя изменять синтаксис операции в выражении. При перегрузке операций сохраняются количество операций, приоритеты операций и правила ассоциативности (справа налево или слева направо), используемые для стандартных типов данных. В частности, унарную операцию нельзя перегружать как бинарную операцию. Так, операции `--`, `++`, `~` могут быть перегружены только как унарные. Операции `+`, `-`, `*`, `&` могут перегружаться как унарные и как бинарные.

ЗАМЕЧАНИЕ

Перечисленные правила 1—3 справедливы не только для объектов-классов, но и для других объектов пользовательского типа.

4. Функции-операции не могут иметь параметров с умалчиваемыми значениями.
5. Функции-операции наследуются, за исключением метода, перегружающего операцию присваивания (подробную информацию о наследовании см. в *гл. 2*).
6. Функции-операции не могут определяться как статические (`static`).
7. Так как для объектов с типом класс перегружать можно только операции, для которых, по крайней мере, хотя бы один операнд имеет тип класса, определенный пользователем, то функция-операция должна быть определена либо как метод этого класса, либо как внешняя функция, дружественная этому классу (напомним, что использование дружественной функции нежелательно).
8. При перегрузке унарной операции перегружающая функция не должна иметь параметров, если она является членом класса, и должна иметь один параметр (ссылку на объект), если является дружественной внешней функцией. Последний вариант не желателен. Объясняется это тем, что при перегрузке унарной операции как функции-члена ей передается неявный аргумент — указатель `this` на текущий объект (операнд).

9. При перегрузке бинарной операции перегружающая функция должна иметь один параметр (ссылку на объект), если она является членом класса, и два параметра (ссылки на объекты), если является дружественной внешней функцией. Последний вариант также не желателен. Объясняется это тем, что при перегрузке бинарной операции как функции-члена класса ей передается один неявный аргумент — указатель `this` на текущий объект (левый операнд).

10. При перегрузке бинарных арифметических операций

+ - * / % ++ --

перегружающие функции *в большинстве случаев* возвращают объект класса, для которого они используются, а не ссылку на объект. Вместе с тем отметим, что для бинарных арифметических операций возможна также реализация перегрузки, при которой возвращается ссылка на объект. Сравнительный анализ этих вариантов будет проведен далее. При перегрузке остальных бинарных операций, напротив, должна возвращаться ссылка на объект. Это обеспечивает возможность использования цепочек одинаковых операций.

11. Если *левый операнд* перегружаемой бинарной операции представляет собой не пользовательский тип, а один из *встроенных* типов, то тогда такая операция не может быть перегружена как функция-член. В этом случае единственной возможностью перегрузки подобной операции является использование дружественной внешней функции, у которой первый параметр имеет один из встроенных типов, а второй — пользовательский тип.

12. Операции

= [] () ->

должны перегружаться только как *члены класса*.

13. Стандарт языка C++ позволяет определять по две различных функции `operator@` для унарных операций `"++"` и `"--"`, допускающих их префиксное и постфиксное использование:

```
// Префиксный
class :: operator@( void );
// Постфиксный: параметр целого типа роли не играет (в качестве
// соответствующего аргумента обычно используется целая константа)
class :: operator@( int );
```

14. *Операция присваивания* определена в любом классе по умолчанию как поэлементное копирование. Эта операция вызывается каждый раз, когда одному существующему объекту присваивается значение другого. Если класс содержит поля (члены-данные), память под которые выделяется динамически, то нужно определить *собственную* операцию присваивания. Чтобы сохранить семантику присваивания, присущую языку C++, метод должен возвращать ссылку на объект класса, для которого он вызван, и принимать в качестве параметра единственный аргумент — ссылку на присваиваемый объект.

15. Операция индексирования `"[]"` обычно перегружается, когда тип класса представляет множество значений, для которого индексирование имеет смысл. Операция индексирования должна возвращать ссылку на элемент, содержащийся во множестве. В этом случае ее можно использовать и в левой, и в правой частях

выражения присваивания. Метод, перегружающий операцию индексирования, получает целый аргумент — индекс элемента множества. При реализации метода целесообразно проверить, лежит ли значение аргумента в пределах допустимого диапазона.

16. Особенности перегрузки операций **new**, **delete**, приведения типа, вызова функции изложены в международном стандарте ISO/IEC 14882 и в [9].

Проиллюстрируем все сказанное содержательным примером (листинги 1.13, 1.14).

Листинг 1.13. Файл **Overload.cpp**

```
/*
    Перегрузка операторов (операций) для классов.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdio.h>          // Для ввода-вывода

//*****
// Для работы с геометрическими векторами
class VECTOR
{
    // Используется умалчиваемый модификатор доступа
    // private

    // ***** Данные *****

    int      x, y;          // Координаты конца вектора

    // ***** Методы *****

public:

    // Присвоить значения координатам вектора: подставляемый метод.
    // Вместо него можно было бы использовать конструктор с двумя
    // параметрами - рекомендуем вам это сделать
    void Assign( const int &x1, const int &y1 )
    {
        x = x1; y = y1;

        return;
    }

    // Перегрузка префиксного унарного оператора "++" с помощью метода
    // класса: прибавляет к вектору единичный вектор и возвращает сам
    // вектор (не ссылку!). Метод не имеет явно передаваемых
```

```

// аргументов, так как при вызове неявно получает адрес объекта
// (this), для которого он вызван. Подставляемый метод
VECTOR operator++( void )
{
    x++; y++;

    return *this;    // Возвращается сам вектор после
                    // увеличения
}

// Перегрузка постфиксного унарного оператора "++" с помощью метода
// класса: возвращает значение вектора до прибавления единичного
// вектора. В методе "operator++( int )" параметр метода
// игнорируется. Метод является подставляемым
VECTOR operator++( int )
{
    VECTOR vec_tmp = *this;

    x++; y++;

    return vec_tmp; // Возвращает значение вектора до
                    // увеличения
}

// Перегрузка бинарного оператора "+" с помощью метода класса:
// суммирует два вектора и возвращает их сумму (не ссылку).
// Отмечаем, что модификатор const гарантирует, что данный метод не
// изменит значений членов объекта, для которого он вызван
VECTOR operator+( const VECTOR & ) const;

// Печать значений координат вектора. Подставляемый метод класса
void print( void ) const
{
    printf( "\n Coordinates of a vector x=%d y=%d", x, y );

    return;
}

// Перегрузка бинарной операции "=": подставляемый метод (при его
// наличии можно корректно использовать цепочки присваиваний).
// В нашем примере этот метод можно было не записывать, так как в нем
// не используются цепочки присваиваний и нет полей указателей на
// области динамической памяти
VECTOR & operator=( // Возвращает ссылку на объект
const VECTOR

```

```

        &v )           // Правый операнд
    {
        this->x = v.x; this->y = v.y;
        // А можно и так: x = v.x; y = v.y;

        return *this;
    }

// Дружественные функции класса: для них спецификатор доступа роли не
//  играет

// Дружественная функция для перегрузки префиксного унарного
//  оператора: вычитает из вектора единичный вектор и возвращает сам
//  вектор. Этот оператор можно было перегрузить и с помощью метода
//  класса
friend VECTOR operator--( VECTOR & );

// Дружественная функция для перегрузки бинарного оператора "-": ищет
//  разность двух векторов и возвращает вектор разности (не
//  ссылку!). Этот оператор также можно было перегрузить и с помощью
//  метода класса
friend VECTOR operator-( const VECTOR &,
                        const VECTOR & );

};

//*****
// Оператор "+" бинарный, поэтому метод "operator+" должен иметь помимо
//  скрытого (this) еще один явно передаваемый параметр. Обратите
//  внимание на то, как надо определять метод класса вне блока класса
VECTOR VECTOR :: operator+( const VECTOR &vec ) const
{
    // Создать вектор для получения суммы и вначале присвоить ему
    //  значение первого слагаемого
    VECTOR    vec_tmp = *this;

    // Сложить векторы
    vec_tmp.x += vec.x; vec_tmp.y += vec.y;

    return vec_tmp;           // и вернуть результат
}

//*****
// Функции-друзья класса не являются членами этого класса и не получают
//  скрытого параметра this. Поэтому унарные операции перегружаются

```

```
// дружественными функциями с одним аргументом, а бинарные - с двумя

// В дружественную функцию "operator--" передается ссылка, так как
// префиксная операция уменьшения должна изменить сам операнд
VECTOR operator--( VECTOR &vec )
{
    vec.x--; vec.y--;

    return vec;        // Возвращается сам объект vec после уменьшения
}

//*****
// Дружественная функция для перегрузки бинарного оператора "-": ищет
// разность двух векторов (vec1-vec2) и возвращает вектор разности (не
// ссылку!)
VECTOR operator-( const VECTOR &vec1, const VECTOR &vec2 )
{
    VECTOR    vec_tmp = vec1;
    vec_tmp.x -= vec2.x; vec_tmp.y -= vec2.y;

    return vec_tmp;    // Возвращает результат вычитания векторов
}

//*****
// Тестирование класса

const int      N = 5;    // Число векторов в массиве

int main( void )        // Возвращает 0 при успехе
{
    // Создаем массив векторов и еще 1 вектор
    VECTOR      v_a[ N ], v1;

    // Инициализируем и печатаем массив векторов
    for( int i = 0; i < N; i++ )
    {
        v_a[i].Assign( i, i+1 );
        printf( "\n Have assigned value to a vector # %d", i );
        v_a[i].print( );
    }

    // Из элементов массива векторов вычитаем единичные вектора и
    // печатаем массив
    for( i = 0; i < N; i++ )
    {
        --v_a[i];
    }
}
```

```

        printf( "\n Have deducted from a vector #%d unit vector.", i );
        v_a[i].print( );
    }

    // Вычисляем сумму элементов массива векторов и печатаем ее
    v1.Assign( 0, 0 );
    printf( "\n The sum of all vectors of the array: " );
    for( i = 0; i < N; i++ )
    {
        v1 = v1 + v_a[i];
    }
    v1.print( );
    printf( "\n" );

    return 0;
}

```

Листинг 1.14. Результат работы программы

```

Have assigned value to a vector # 0
Coordinates of a vector x=0 y=1
Have assigned value to a vector # 1
Coordinates of a vector x=1 y=2
Have assigned value to a vector # 2
Coordinates of a vector x=2 y=3
Have assigned value to a vector # 3
Coordinates of a vector x=3 y=4
Have assigned value to a vector # 4
Coordinates of a vector x=4 y=5
Have deducted from a vector #0 unit vector.
Coordinates of a vector x=-1 y=0
Have deducted from a vector #1 unit vector.
Coordinates of a vector x=0 y=1
Have deducted from a vector #2 unit vector.
Coordinates of a vector x=1 y=2
Have deducted from a vector #3 unit vector.
Coordinates of a vector x=2 y=3
Have deducted from a vector #4 unit vector.
Coordinates of a vector x=3 y=4
The sum of all vectors of the array:
Coordinates of a vector x=5 y=10
Press any key to continue

```

В качестве упражнения попробуйте "поиграть" с этим примером, модифицируйте его. С этой целью перегрузите префиксные и постфиксные операции "++" и "--" различными способами и протестируйте все перегруженные операции.

В качестве еще одного упражнения модифицируйте программу, для чего определение класса и определение дружественных функций поместите в заголовочный файл. С помощью директив условной трансляции исключите возможность многократного подключения заголовочного файла.

1.6. Шаблоны классов

В учебном пособии [3] в *разд. 12.8* рассмотрены шаблоны функций, с помощью которых можно отделить алгоритм работы шаблона функции от конкретных типов данных, с которыми он работает. Это обеспечивается тем, что конкретный тип данных передается шаблону в качестве параметра настройки. *Шаблоны классов* предоставляют аналогичную возможность, позволяя создавать параметризованные классы.

Шаблоны классов называют также "генераторами классов". Шаблоны позволяют определить структуру семейства классов, по которой компилятор создает конкретные классы в дальнейшем. Шаблонами классов настолько удобно пользоваться, что *стандартная библиотека шаблонов классов* (Standard Template Library, STL) была включена в состав языка C++.

Преимущество использования шаблона классов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода. Рассмотрим процесс создания и использования шаблона класса на примере шаблона класса для работы с массивами-векторами (листинги 1.15, 1.16).

Листинг 1.15. Файл TemplateVector.cpp

```
/*
    Шаблон класса для работы с массивами-векторами.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdlib.h>          // Для функции exit()
#include <iostream.h>        // Для потокового ввода-вывода (подробнее об
                             // этом см. в гл. 5)

//*****
// Шаблон классов: T, size - параметры настройки: в данном случае T есть
// тип размещаемых элементов, а size - их максимальное количество
// (обратите внимание, что в качестве size можно использовать только
// константное выражение). Обратите внимание, как следует оформлять
// заголовок шаблона класса
template< class T, int size >
class TVECTOR
{
    // ***** Данные *****

    // Используется умалчиваемый модификатор доступа private:
```

```

T          *elements; // Указатель на массив элементов типа T

// ***** Методы *****

public:

TVECTOR( void );      // Конструктор
~TVECTOR( void )      // Деструктор: подставляемый метод
{
    if( elements )
    {
        delete [ ] elements; elements = NULL;
    }
}

// Перегрузка оператора [] выбора элемента вектора с помощью метода
// класса (подставляемый метод)
T & operator[ ](      // Возвращает ссылку на элемент массива, поэтому
                    // имя объекта с [] можно использовать и в
                    // левой и в правой частях выражения
                    // присваивания
    int    i )        // Индекс элемента массива
{
    if( i<0 || i>=size )
    {
        cout << "Incorrect value of an index of a unit" << endl;
        // Вывод на экран с помощью перегруженного оператора <<
        // (используется умалчиваемый формат вывода). Манипулятор
        // endl выполняет переход на новую строку и очистку буфера
        exit( 1 );
    }
    return elements[ i ];
}

// Печать содержимого вектора
void print_contents( void );

};

// Определения методов шаблона классов
//*****

// Конструктор. Обратите внимание, как следует определять метод шаблона
// класса вне блока шаблона класса
template< class T, int size >

```

```

TVECTOR< T, size > :: TVECTOR( void )
{
    // Размещение массива в динамической памяти
    elements = new T[ size ];
    if( !elements )
    {
        cout << " The array in DM to place it was not possible " << endl;
        exit( 1 );
    }

    // Инициализация массива
    for( int i = 0; i < size; elements[ i ] = ( T )0, i++ );
}

//*****
// Метод для печати содержимого вектора
template< class T, int size >
void TVECTOR< T, size > :: print_contents( void )
{
    cout << " In total of units: " << size << ". Units:";
    for( int i = 0; i < size; i++ )
    {
        if( i%5 == 0 )
            cout << endl;
        cout.width( 10 );// Задаёт ширину поля вывода 10
        cout << elements[ i ];
    }
    cout << endl;

    return;
}

//*****
// Тестирование шаблона классов
int main( void )        // Возвращает 0 при успехе
{
    // Создаем векторы с элементами типа int, double и char, включающие
    // по 10 элементов путем вызова конструктора шаблона классов
    TVECTOR< int, 10 >
        i;
    TVECTOR< double, 10 >
        d;
    TVECTOR< char, 10 >
        ch;

    // Печатаем содержимое векторов
    i.print_contents();
    d.print_contents();
}

```

```

ch.print_contents( );

// Присваиваем значения векторам (используется перегруженная операция
// [ ])
for( int count = 0; count < 10; count++ )
{
    i[ count ] = count; d[ count ] = count + 0.1;
    ch[ count ] = count + 'a';
}

// Печатаем содержимое векторов
i.print_contents( ); d.print_contents( );
ch.print_contents( );

return 0;
}

```

Листинг 1.16. Информация, выводимая на экран после выполнения программы

```

In total of units: 10. Units:
    0      0      0      0      0
    0      0      0      0      0
In total of units: 10. Units:
    0      0      0      0      0
    0      0      0      0      0
In total of units: 10. Units:

In total of units: 10. Units:
    0      1      2      3      4
    5      6      7      8      9
In total of units: 10. Units:
    0.1     1.1     2.1     3.1     4.1
    5.1     6.1     7.1     8.1     9.1
In total of units: 10. Units:
    a      b      c      d      e
    f      g      h      i      j
Press any key to continue

```

По результатам рассмотрения данного примера подведем некоторые итоги. В общем случае *синтаксис описания шаблона класса* имеет следующий вид:

```
template< описание_параметров_шаблона > определение_класса
```

Параметры шаблона перечисляются через запятую. В качестве параметров могут применяться типы, шаблоны и типизированные константы. В рассмотренном нами примере в качестве параметров были использованы тип элементов вектора и типизированная константа, задающая размер массива. Типы могут быть как стандартными,

так и определенными пользователем. Для их описания используется служебное слово `class`. Внутри шаблона класса параметр типа может применяться в любом месте, где допустимо использовать спецификацию типа. Для любых параметров шаблона могут быть заданы значения по умолчанию. Область действия параметра шаблона — от точки описания параметра шаблона до конца шаблона класса.

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона класса, то заголовок метода должен иметь следующую структуру:

```
template< описание_параметров_шаблона>
    возвр_тип имя_класса< параметры_шаблона > ::
...

```

Описание параметров шаблона в заголовке метода должно соответствовать шаблону класса, причем для `описания_параметров_шаблона` и `параметров_шаблона` должно соблюдаться количественное и позиционное соответствие.

Перечислим основные правила описания шаблонов:

1. Если в многофайловом программном проекте требуется работать с обычным классом во многих файлах проекта, то определение класса можно выполнить двумя разными способами:
 - и объявление класса, и реализация методов класса помещаются в один общий заголовочный файл, подключаемый к соответствующим файлам проекта;
 - объявление класса помещается в заголовочный файл, подключаемый к соответствующим файлам программного проекта, а реализацию методов класса в файл с расширением `cpp`, который также включается в программный проект.

ЗАМЕЧАНИЕ

Для шаблона класса доступен только первый способ.

2. Локальные классы не могут содержать шаблоны в качестве своих элементов.
3. Шаблоны методов не могут быть виртуальными (о виртуальных методах см. гл. 3).
4. Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
5. Шаблоны классов могут быть производными как от шаблонных, так и от обычных классов, а также являться базовыми как для шаблонов, так и для обычных классов (о наследовании см. гл. 2).
6. Внутри шаблона класса нельзя определять шаблоны дружественных функций.

Если требуется создать вектор с размером, задаваемым переменной, а не константой, следует использовать конструктор с параметром (листинг 1.17).

Листинг 1.17. Файл `TemplateVector1.cpp`

```
/*
```

```
    Шаблон класса для работы с массивами-векторами.
```

```
    Консольное приложение, Microsoft Visual Studio C++ 6.0
```

```

*/

#include <stdlib.h>          // Для функции exit()
#include <iostream.h>        // Для потокового ввода-вывода (подробнее об
                             // этом см. в гл. 5)

/*****
// Шаблон классов: Т - параметры настройки: в данном случае Т есть тип
// размещаемых элементов. Обратите внимание на то, как следует
// оформлять заголовок шаблона класса
template< class T >
class TVECTOR1
{
    // ***** Данные *****

    // Используется умалчиваемый модификатор доступа private:

    T          *elements; // Указатель на массив элементов
                             // типа Т
    int         l;         // Длина вектора

    // ***** Методы *****

public:

    TVECTOR1( int size ); // Конструктор
    // ...

    // Перегрузка оператора [] выбора элемента вектора с помощью метода
    // класса (подставляемый метод)
    T & operator[ ] (      // Возвращает ссылку на элемент массива, поэтому
                             // имя объекта с [] можно использовать и в
                             // левой и в правой частях выражения
                             // присваивания
        int i )           // Индекс элемента массива
    {
        if( i<0 || i>=l )
        {
            cout << "Incorrect value of an index of a unit" << endl;
            // Вывод на экран с помощью перегруженного оператора <<
            // (используется умалчиваемый формат вывода). Манипулятор
            // endl выполняет переход на новую строку и очистку буфера
            exit( 1 );
        }
        return elements[ i ];
    }
}

```

```

    // Печать содержимого вектора
    void print_contents( void );

};

// Определения методов шаблона классов
//*****

// Конструктор. Обратите внимание, как следует определять метод шаблона
// класса вне блока шаблона класса
template< class T >
TVECTOR1< T > :: TVECTOR1(
    int      size )      // Размер массива
{
    // Размещение массива в динамической памяти
    elements = new T[ size ];
    if( !elements )
    {
        cout << " The array in DM to place it was not possible " << endl;
        exit( 1 );
    }

    // Инициализация массива
    for( int i = 0; i < size; elements[ i ] = ( T )0, i++ );

    l = size;
}

//*****
// Метод для печати содержимого вектора
template< class T >
void TVECTOR1< T > :: print_contents( void )
{
    cout << " In total of units: " << l << ". Units:";
    for( int i = 0; i < l; i++ )
    {
        if( i%5 == 0 )
            cout << endl;
        cout.width( 10 ); // Задаёт ширину поля вывода 10
        cout << elements[ i ];
    }
    cout << endl;

    return;
}

```

```

//*****
// Тестирование шаблона классов
int main( void )           // Возвращает 0 при успехе
{
    // Создаем векторы с элементами типа int, double и char, включающие
    // по 10 элементов путем вызова конструктора шаблона классов
    TVECTOR1< int >
        i( 10 );
    TVECTOR1< double >
        d( 10 );
    TVECTOR1< char >
        ch( 10 );

    // Печатаем содержимое векторов
    i.print_contents( );
    d.print_contents( );
    ch.print_contents( );

    // Присваиваем значения векторам (используется перегруженная операция
    // [])
    for( int count = 0; count < 10; count++ )
    {
        i[ count ] = count; d[ count ] = count + 0.1;
        ch[ count ] = count + 'a';
    }

    // Печатаем содержимое векторов
    i.print_contents( ); d.print_contents( );
    ch.print_contents( );

    return 0;
}

```

1.7. Конструктор копирования.

Поверхностное и глубинное копирование

Теперь хотим обратить внимание читателей на наличие в классах специального вида конструктора, называемого *конструктором копирования*. Конструктор копирования получает в качестве единственного параметра ссылку на объект этого же класса:

```

C :: C( const C &ObjC )
{
    ... /* тело-конструктора */
}

```

Конструктор копирования вызывается в тех случаях, когда новый объект с типом класс создается путем копирования существующего:

- ❑ при определении нового объекта с типом класс с инициализацией его другим существующим объектом того же типа;
- ❑ при передаче в метод класса параметра-объекта с типом класса по значению;
- ❑ при возврате из метода класса значения объекта с типом класса с помощью оператора `return`.

Если программист не создал ни одного конструктора копирования, то компилятор создаст автоматически *умалчиваемый конструктор копирования*. Такой конструктор выполняет поэлементное копирование полей данных класса (поверхностное копирование). Если же какое-либо из полей является указателем на некоторую область динамической памяти, зарезервированную, например, с помощью операции `new`, то использование умалчиваемого конструктора копирования приведет к неверной работе программы. В этом случае следует, вместо умалчиваемого конструктора копирования, спроектировать свой конструктор копирования:

- ❑ конструктор, выполняющий *поверхностное копирование*;
- ❑ конструктор с *глубинным копированием*.

Особенности использования *конструктора с поверхностным копированием* иллюстрируют листинги 1.18, 1.19.

Листинг 1.18. Файл CMP0.CPP

```
/*
   Работа с комплексными данными с применением класса CMP, используется
   динамическое размещение комплексного объекта и конструктор копирования
   с "поверхностным" копированием.
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdlib.h>          // Для функции exit
#include <iostream.h>        // Для потокового ввода-вывода

//*****
// Класс для работы с комплексными объектами
class CMP
{
    // Данные

private:

    struct C                // Для комплексного данного
    {
        int    r,           // Действительная часть
            i,             // Мнимая часть
            cc;             // Counter Copies: счетчик копий объекта
    };
};
```

```

    }                *pc;          // Указатель на комплексное данные

    // Методы

public:

    // Конструктор по умолчанию - вызывается при создании объектов с
    // типом CMP (подставляемый метод)
    CMP( void )
    {
        // Размещение объекта в динамической памяти
        if( !( pc = new C ) )
        {
            cout << "\n The dynamic memory does not suffice"
                 " - the allocation of the complex "
                 "object is not fulfilled " << endl;
            exit( 1 );
        }

        pc->r = pc->i = 0;          // Начальная инициализация
        pc->cc = 0;                 // Пока копий объекта нет
    }

    // Конструктор копирования - вызывается при создании нового объекта с
    // инициализацией другим объектом, при передаче параметра с типом
    // класса по значению и при возврате значения объекта с типом
    // класса из метода класса по return.
    // Поверхностное копирование с подсчетом числа копий объекта -
    // копируется только указатель на объект, а копия объекта в
    // динамической памяти не создается. Подставляемый метод
    CMP(
        const CMP &copy )// Источник копирования
    {
        pc = copy.pc; pc->cc++;
    }

    // Деструктор: подставляемый метод
    ~CMP( void )
    {
        // Если копии объекта имеются, то занятая ранее динамическая
        // память пока не освобождается, а счетчик копий уменьшается на
        // единицу
        if( !pc->cc )
            delete pc;
        else
            pc->cc--;
    }

```

```

}

// Определение значения: подставляемый метод
void read_cmp(
    const int
        &re,          // Действительная часть
    const int
        &im )        // Мнимая часть
{
    pc->r = re; pc->i = im;

    return;
}

// Печать значения: подставляемый метод
void print_cmp( void )
{
    cout.width( 15 ); cout << pc->r << " i ";
    cout.width( 15 ); cout << pc->i << endl;

    return;
}

// Перегрузка оператора "-" с применением метода класса
CMP operator-(          // Возвращает разность: уменьшаемое задается с
                        // помощью this
    const CMP
        &c2 )          // Вычитаемое
{
    // Временное комплексное данные - здесь вызывается конструктор по
    // умолчанию
    CMP    tmp;

    // Вычисление разности
    tmp.pc->r = pc->r - c2.pc->r;
    tmp.pc->i = pc->i - c2.pc->i;

    // Здесь вызывается конструктор копирования и, после него,
    // деструктор
    return tmp;
}

// Перегрузка оператора "=" с применением метода класса (этот
// оператор нельзя перегрузить с помощью дружественной функции)
CMP & operator=(        // Возвращает ссылку (для левой части)
    const CMP

```

```

        &c )          // Правая часть
    {
        pc->r = c.pc->r; pc->i = c.pc->i;

        return *this;
    }

};                                // Конец определения класса

//*****
// Тестирование класса
int main( void )                // Возвращает 0 при успехе
{
    // Пять комплексных объектов: последовательно вызывается конструктор
    // по умолчанию для cu, cv1, cv2, cr1, cr2
    CMP      cu, cv1, cv2, cr1, cr2;
    cout << " Values of objects after creation:" << endl;
    cout << "cu: "; cu.print_cmp( ); cout << "cv1:";
    cv1.print_cmp( ); cout << "cv2:"; cv2.print_cmp( );
    cout << "cr1:"; cr1.print_cmp( ); cout << "cr2:";
    cr2.print_cmp( );

    // Чтение значений
    cu.read_cmp( 10, 10 ); cv1.read_cmp( 1, 1 );
    cv2.read_cmp( 2, 2 );
    cout << " Values of objects after reading:" << endl;
    cout << "cu: "; cu.print_cmp( ); cout << "cv1:";
    cv1.print_cmp( ); cout << "cv2:"; cv2.print_cmp( );
    cout << "cr1:"; cr1.print_cmp( ); cout << "cr2:";
    cr2.print_cmp( );

    // Цепочки вычитаний и присваиваний
    // !!! Здесь дважды вызывается деструктор - по одному разу для
    // каждого из перегруженных операторов "-" для разрушения копий
    // возвращаемых значений - см. текст деструктора
    cr1 = cr2 = cu - cv1 - cv2;
    cout << "cr1 = cr2 = cu - cv1 - cv2;" << endl;
    cout << "cu: "; cu.print_cmp( ); cout << "cv1:";
    cv1.print_cmp( ); cout << "cv2:"; cv2.print_cmp( );
    cout << "cr1:"; cr1.print_cmp( ); cout << "cr2:";
    cr2.print_cmp( );

    cr1 = cr2 = cu - ( cv1 = cv2 ) - cv1;
    // !!! Здесь дважды вызывается деструктор - по одному разу для
    // каждого из перегруженных операторов "-" для разрушения копий
    // возвращаемых значений - см. текст деструктора

```

```

cout << "cr1 = cr2 = cu - ( cv1 = cv2 ) - cv1;" << endl;
cout << "cu: "; cu.print_cmp( ); cout << "cv1:";
cv1.print_cmp( ); cout << "cv2:"; cv2.print_cmp( );
cout << "cr1:"; cr1.print_cmp( ); cout << "cr2:";
cr2.print_cmp( );

// Здесь последовательно вызываются деструкторы для cr2, cr1, cv2,
//   cv1, cu
return 0;
}

```

Листинг 1.19. Информация, выводимая на экран в результате выполнения программы

```

Values of objects after creation:
cu:          0 i          0
cv1:         0 i          0
cv2:         0 i          0
cr1:         0 i          0
cr2:         0 i          0

Values of objects after reading:
cu:          10 i         10
cv1:         1 i          1
cv2:         2 i          2
cr1:         0 i          0
cr2:         0 i          0

cr1 = cr2 = cu - cv1 - cv2;
cu:          10 i         10
cv1:         1 i          1
cv2:         2 i          2
cr1:         7 i          7
cr2:         7 i          7

cr1 = cr2 = cu - ( cv1 = cv2 ) - cv1;
cu:          10 i         10
cv1:         2 i          2
cv2:         2 i          2
cr1:         6 i          6
cr2:         6 i          6

Press any key to continue

```

ВНИМАНИЕ!

В принципе, конструктор копирования в классе можно было не определять. В подобной ситуации будет вызываться конструктор копирования по умолчанию, который, в отличие от использованного в примере, не будет вести счетчик копий. Это, в свою очередь, приведет к некорректной работе программы.

Как и ранее, в качестве упражнения для самостоятельной работы рекомендуем модифицировать только что рассмотренную программу: оформить определение класса и его методов в виде включаемого файла с предотвращением возможности его многократного включения.

Хотя рассмотренный ранее конструктор с поверхностным копированием и позволяет добиться корректной работы программы, все же предпочтительнее использование *конструктора с глубинным копированием*. Основные аспекты его проектирования и использования демонстрирует приводимый далее листинг 1.20.

Листинг 1.20. Файл CMP1.CPP

```
/*
   Работа с комплексными данными с использованием класса CMP1, используется
   динамическое размещение комплексного объекта и конструктор копирования
   с "глубинным" копированием.
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdlib.h>          // Для функции exit
#include <iostream.h>        // Для потокового ввода-вывода

//*****
// Класс для работы с комплексными объектами
class CMP1
{
    // Данные

private:

    struct C                // Для комплексного данного
    {
        int    r,           // Действительная часть
              i;            // Мнимая часть
    }          *pc;         // Указатель на комплексное данное

    // Методы

public:

    // Конструктор умолчания - вызывается при создании объектов с типом
    //   CMP1 (подставляемый метод)
    CMP1( void )
    {
        // Размещение в динамической памяти
        if( !( pc = new C ) )
        {
```

```

        cout << " The dynamic memory - does not "
              "suffice - the allocation complex given is"
              " not fulfilled " << endl;
        exit( 1 );
    }

    pc->r = pc->i = 0;          // Начальная инициализация
}

// Конструктор копирования - вызывается при определении объекта
// класса с его инициализацией другим объектом, при передаче в
// метод параметра с типом класса по значению и при передаче
// значения объекта из метода по return
CMP1(
    const CMP1
        &copy )    // Источник копирования
{
    // Глубинное копирование! При его использовании не требуется
    // подсчитывать копии и анализировать их наличие при
    // освобождении динамической памяти в деструкторе.
    // Подставляемый метод
    if( !( pc = new C ) )
    {
        cout << " The dynamic memory - allocation "
              "complex given does not suffice is not "
              "fulfilled " << endl;
        exit( 1 );
    }

    pc->r = copy.pc->r; pc->i = copy.pc->i;
}

// Деструктор: подставляемый метод
~CMP1( void )
{
    if( pc )
    {
        delete pc; pc = NULL;
    }
}

// Последующая часть примера совпадает с предыдущим примером, но
// вместо имени класса CMP используется имя CMP1. Результаты
// выполнения программы выводятся на экран и имеют такой же вид,
// как в предыдущем примере
// ...

```

В заключение повторим еще раз сделанное ранее важное замечание. Если в классе не производится работа с динамической памятью и, следовательно, деструктор не нужен, то можно не определять конструктор копирования, пользуясь конструктором копирования по умолчанию. Программы, использующие подобные классы, будут работать правильно.

Из сказанного следует, что надо, по возможности, избегать передачи в метод класса параметра с типом класса по значению и возвращать из метода класса не значение объекта с типом класса, а ссылку на него. При выполнении данной рекомендации становится ненужным конструктор копирования и быстродействие приложения повышается. Для достижения указанной цели достаточно придерживаться следующих правил:

- ❑ вместо передачи в метод класса параметра с типом класса по значению следует этот параметр передавать по ссылке, а для предотвращения модификации соответствующего аргумента в вызове функции параметр надо снабдить модификатором `const`;
- ❑ при перегрузке арифметических операций для объектов с типом класса нужно так спроектировать перегружающий метод, чтобы он возвращал ссылку на объект (это можно сделать всегда).

Приведем иллюстрирующий пример (листинги 1.21, 1.22).

Листинг 1.21. Файл CMP2.CPP

```
/*
    Работа с комплексными данными с использованием класса CMP2, используется
    динамическое размещение комплексного данного - конструктор копирования не
    требуется.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdlib.h>          // Для функции exit
#include <iostream.h>        // Для потокового ввода-вывода

//*****
// Класс для работы с комплексными объектами
class CMP2
{
    // Данные

private:

    struct C                // Для комплексного данного
    {
        float  r,           // Действительная часть
              i;            // Мнимая часть
    } *pc;                 // Указатель на комплексное данное
```

```

CMP2      *pCMP;      // Указатель на временный объект (используется в
                      // методе operator-)

// Методы

public:

// Конструктор - вызывается при создании объектов с типом CMP2 и
// выполняет инициализацию объекта (подставляемый метод)
CMP2(
    const float
        &Re,          // Для инициализации вещественной части
    const float
        &Im )         // Для инициализации мнимой части
{
// Размещение в динамической памяти
    if( !( pc = new C ) )
    {
        cout << " The dynamic memory - allocation "
            "complex given does not suffice is not "
            "fulfilled " << endl;
        exit( 1 );
    }

    // Инициализация
    pc->r = Re; pc->i = Im; pCMP = NULL;

    cout << " The constructor CMP2 is called " << endl;
}

// Деструктор (подставляемый метод)
~CMP2( void )
{
    if( pc )
    {
        delete pc; pc = NULL;
    }

    if( pCMP )
    {
        delete pCMP; pCMP = NULL;
    }

    cout << " Is called destructor CMP2" << endl;
}

```

```

}

// Печать значения (подставляемый метод)
void print_cmp( void )
{
    cout.width( 15 ); cout << pc->r << " +i ";
    cout.width( 15 ); cout << pc->i << endl;

    return;
}

// Перегрузка оператора "-" с применением метода класса
//   (подставляемый метод)
CMP2 & operator-(      // Возвращает ссылку на разность: уменьшаемое
                        //   задается с помощью this

    const CMP2
        &c2 )          // Вычитаемое
{
    // Временное комплексное данное размещается в динамической памяти
    //   для объекта класса один раз
    if( !pCMP )
    {
        // Здесь будет вызван конструктор
        pCMP = new CMP2( 0, 0 );
        if( !pCMP )
        {
            cout << " Error of allocation in dynamic memory" << endl;
            exit( 2 );
        }
    }

    // Вычисляем разность
    pCMP->pc->r = pc->r - c2.pc->r;
    pCMP->pc->i = pc->i - c2.pc->i;

    return *pCMP;
}

// Перегрузка оператора "=" с применением метода класса (этот
//   оператор нельзя перегрузить с помощью дружественной функции) -
//   подставляемый метод
CMP2 & operator=(      // Возвращает ссылку (для левой части)

    const CMP2
        &c )           // Правая часть
{

```

```

        pc->r = c.pc->r; pc->i = c.pc->i;

        return *this;
    }

}; // Конец определения класса

//*****
// Тестирование класса
int main( void )
{
    // Пять комплексных объектов: последовательно вызываются конструкторы
    // для cu, cv1, cv2, cr1, cr2
    CMP2      cu( 10, 10 ), cv1( 1, 1 ), cv2( 2, 2 ),
              cr1( 0, 0 ), cr2( 0, 0 );

    // Печать значений объектов после создания
    cout << " Values of objects after creation:" << endl;
    cout << "cu: "; cu.print_cmp( );
    cout << "cv1:"; cv1.print_cmp( );
    cout << "cv2:"; cv2.print_cmp( );
    cout << "cr1:"; cr1.print_cmp( );
    cout << "cr2:"; cr2.print_cmp( );

    // Цепочки вычитаний и присваиваний - здесь два раза будет вызван
    // конструктор (по одному разу для каждой перегруженной операции
    // "-" объектов cv2 и cv1)
    cr1 = cr2 = cu - cv1 - cv2;
    // Печать значений
    cout << "cr1 = cr2 = cu - cv1 - cv2;" << endl;
    cout << "cu: "; cu.print_cmp( );
    cout << "cv1:"; cv1.print_cmp( );
    cout << "cv2:"; cv2.print_cmp( );
    cout << "cr1:"; cr1.print_cmp( );
    cout << "cr2:"; cr2.print_cmp( );

    // Более сложные цепочки вычитаний и присваиваний - здесь конструктор
    // уже не будет вызываться
    cr1 = cr2 = cu - ( cv1 = cv2 ) - cv1;
    // Печать значений
    cout << "cr1 = cr2 = cu - ( cv1 = cv2 ) - cv1;" << endl;
    cout << "cu: "; cu.print_cmp( );
    cout << "cv1:"; cv1.print_cmp( );
    cout << "cv2:"; cv2.print_cmp( );
    cout << "cr1:"; cr1.print_cmp( );

```

```

cout << "cr2:"; cr2.print_cmp( );

// Здесь последовательно вызываются деструкторы для cr2, cr1, cv2,
//   cv1, cu и для двух временных объектов
return 0;
}

```

Листинг 1.22. Результаты выполнения программы

```

The constructor CMP2 is called
The constructor CMP2 is called
The constructor CMP2 is called
The constructor CMP2 is called
The constructor CMP2 is called
Values of objects after creation:
cu:          10 +i          10
cv1:          1 +i           1
cv2:          2 +i           2
cr1:          0 +i           0
cr2:          0 +i           0
The constructor CMP2 is called
The constructor CMP2 is called
cr1 = cr2 = cu - cv1 - cv2;
cu:          10 +i          10
cv1:          1 +i           1
cv2:          2 +i           2
cr1:          7 +i           7
cr2:          7 +i           7
cr1 = cr2 = cu - ( cv1 = cv2 ) - cv1;
cu:          10 +i          10
cv1:          2 +i           2
cv2:          2 +i           2
cr1:          6 +i           6
cr2:          6 +i           6
Is called destructor CMP2
Is called destructor CMP2
Is called destructor CMP2
Is called destructor CMP2
Is called destructor CMP2
Is called destructor CMP2
Is called destructor CMP2
Press any key to continue

```

В соответствии с известной житейской мудростью "за удовольствия надо платить" приведенный пример, наряду с отмеченными ранее достоинствами, не лишен и некоторых недостатков. Таким недостатком, в частности, является то, что под временный

объект с типом класс при использовании перегруженных арифметических операций занимается динамическая память. Правда, динамическая память *автоматически* резервируется только при использовании перегруженных арифметических операций и только в одном экземпляре для каждого операнда-объекта. Освобождается дополнительная динамическая память *автоматически*, одновременно с разрушением объекта в деструкторе класса. Поэтому указанный недостаток не является существенным.

ПРИМЕЧАНИЕ

Как же быть с перегрузкой арифметических операций? Должны ли соответствующие перегружающие функции возвращать значение объекта или ссылку на объект? Для решения этого вопроса нужно руководствоваться следующим правилом. Если важнее быстрое действие приложения, то перегружающие арифметические операции функции должны возвращать ссылку на объект. И наоборот, если требуется минимизировать память, требуемую приложению, то перегружающие арифметические операции функции должны возвращать значение объекта.

1.8. Вопросы и упражнения для самопроверки [4]

1. Чем определяется размер объекта класса?
2. Почему не следует делать члены-данные класса открытыми?
3. Имеет ли смысл использовать структуры в программах на языке C++?
4. Что представляет собой оператор прямого доступа и для чего он используется с объектом класса?
5. Что резервирует память — определение класса или определение объекта с типом класса?
6. Объявление класса является его интерфейсом или реализацией класса?
7. Какова разница между открытыми (`public:`) и закрытыми (`private:`) членами-данными класса?
8. Могут ли методы класса быть закрытыми?
9. Могут ли данные-члены класса быть открытыми?
10. Если определить два объекта некоторого класса, то могут ли они иметь различные значения своих членов-данных?
11. Нужно ли объявление класса завершать точкой с запятой? А определение метода класса?
12. Как бы выглядел заголовок определения метода `GetX()` класса `ANYCLASS`, который не принимает никаких параметров и возвращает целое значение? Считайте, что определение метода помещается вне блока класса.
13. Какой метод вызывается для инициализации объекта с типом класса?
14. Напишите объявление класса с именем `EMPLOYEE` (служащие) с открытыми членами-данными целого типа `Age` (возраст) и `Salary` (зарплата).
15. Перепишите класс `EMPLOYEE` из п. 14 таким образом, чтобы сделать члены-данные класса закрытыми и обеспечить открытые методы доступа для чтения и установки значений всех членов-данных.
16. Напишите программу с использованием класса `EMPLOYEE` из п. 15, которая создает объекты `Ivanov` и `Petrov` этого класса, задает значения членов-данных этих объектов, а затем выводит их на печать.

17. В класс из п. 15 добавьте метод, который сообщает, сколько тысяч рублей зарабатывает служащий, округляя ответ до тысячи рублей. Напишите определение этого метода, поместив его вне блока класса.
18. Измените класс `EMPLOYEE` из п. 14 так, чтобы можно было инициализировать члены-данные класса в процессе "создания" служащего.
19. Что неправильно в следующем объявлении класса?

```
class Square
{
public:

    int    Side;
}
```

20. Что весьма полезное отсутствует в следующем объявлении класса?

```
class Square
{
    int    Side;
    void SetSide( int side );
};
```

21. Какие три ошибки обнаружит компилятор в этом программном коде?

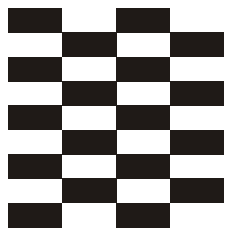
```
class STATION
{
private:
    int    Station;
public:
    void SetStation( int station );
    int GetStation( void ) const;
};
// ...
int main( void )
{
    STATION mySTATION;
    mySTATION.Station = 9;
    STATION.SetStation( 10 );
    STATION myOtherSTATION( 2 );
    return 0;
}
```

22. Какие члены-данные класса следует инициализировать одновременно с инициализацией конструктора, а какие инициализировать в теле конструктора?
23. Почему одни методы класса определяются в блоке класса, а другие — за его пределами?
24. Какая разница между определением класса и определением объекта класса?

25. Когда вызывается конструктор копирования?
26. Когда вызывается деструктор?
27. Чем отличается конструктор копирования от оператора присваивания?
28. Что представляет собой указатель `this`?
29. Как различить перегрузку префиксных и постфиксных операторов приращения?
30. Можно ли перегрузить операцию суммирования для операндов типа `short int`?
31. Допускается ли в C++ перегрузка оператора `++` таким образом, чтобы он выполнял операцию декремента?
32. Напишите объявление класса `SimpleCircle` (простая окружность) с единственным членом-данным `Radius` (радиус). В классе должны использоваться конструктор умолчания и деструктор, а также методы для чтения и записи значения `Radius`.
33. Для класса `SimpleCircle`, созданного в п. 32, напишите определение конструктора умолчания, инициализирующего `Radius` значением 5. Инициализацию выполните во время инициализации конструктора, а определение конструктора умолчания поместите вне блока класса.
34. Для класса `SimpleCircle`, созданного в п. 32, напишите определение обычного конструктора, который присваивает значение своего параметра члену-данному `Radius`. Инициализацию выполните в блоке конструктора.
35. Для класса `SimpleCircle`, созданного в пп. 32—34, перегрузите операции преинкремента и постинкремента для члена-данного `Radius`.
36. Измените класс `SimpleCircle`, созданный в пп. 32—35, таким образом, чтобы член-данное `Radius` размещался в динамической памяти. Зафиксируйте все созданные методы класса.
37. В класс `SimpleCircle`, созданный в п. 36, добавьте конструктор копирования.
38. В класс `SimpleCircle`, созданный в п. 37, добавьте метод, перегружающий операцию присваивания.
39. На базе класса `SimpleCircle`, созданного в пп. 37—38, напишите программу, которая создает два объекта этого класса. Для создания одного объекта используйте конструктор умолчания, а второму объекту при его определении присвойте значение 9. С каждым из объектов используйте оператор инкремента и выведите полученные значения на печать. И, наконец, присвойте значение одного объекта другому объекту и выведите результат на печать.

Ответы на вопросы и решения для упражнений можно проверить — они приведены в разд. П1.1 приложения 1.

Глава 2



Наследование. Иерархия классов

Наследование — один из главных механизмов объектно-ориентированного программирования языка C++. С его помощью можно разрабатывать очень сложные классы, продвигаясь от общего к частному, а также наращивать уже созданные классы, получая от них новые классы, отличающиеся от исходных классов.

Проектируя новый класс, необходимо, прежде всего, уяснить, какими наиболее общими свойствами должны обладать его объекты и нет ли похожего готового класса. Иными словами, следует вначале набросать "крупными мазками" план вновь разрабатываемого класса, а затем переходить к постепенной детализации, создавая на основе уже построенных классов новые, которые наследуют от них свойства и поведение (т. е. члены-данные и методы класса), приобретая в то же время новые качества.

2.1. Иерархия наследования классов

Чтобы указать, по отношению к каким базовым классам данный определяемый класс является производным, в определении класса перед списком его членов (перед открывающей фигурной скобкой) приводится список базовых классов.

Класс, определенный при помощи служебного слова `class`, может быть как базовым, так и производным. В этом случае по умолчанию члены класса являются закрытыми. Класс, определенный при помощи служебного слова `struct`, также может быть как базовым, так и производным, и по умолчанию члены класса являются открытыми. Класс, определенный при помощи служебного слова `union`, не может быть ни базовым, ни производным по отношению к какому бы то ни было другому классу.

Рассмотрим иллюстрирующий пример (листинги 2.1—2.3).

Листинг 2.1. Файл LINSР_N.H

```
/*  
    Включаемый файл для LINSР_N.CPP.  
    Содержит определение класса LINSР_N для работы с однонаправленным линейным списком.  
    Класс LINSР_N является производным от класса LINSР, определенного в файле LINSР.H  
    (см. разд. 1.1). Алгоритмы операций над линейным списком, реализованных в  
    производном классе, подробно рассмотрены в [3].  
*/
```

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предотвращение многократного включения файла
#ifndef __LINSN_N_H
#define __LINSN_N_H

// Подключение базового класса для работы с однонаправленным
//   неколецевым линейным списком
#include "linsp.h"

//*****
// Объявление производного класса для работы с однонаправленным
//   неколецевым линейным списком. Обратите внимание на оформление
//   заголовка объявления производного класса
class LINSN_N : public LINSN
{
    // Класс LINSN_N, производный от класса LINSN, наследует от него
    //   все его данные и методы (включая деструктор). Конструктор
    //   базового класса не наследуется, но при создании класса,
    //   наследуемого от другого класса, сначала вызывается
    //   конструктор для базового класса, а затем конструктор,
    //   определенный в производном классе (если он есть).
    // Компилятор ищет конструктор базового класса по умолчанию
    //   (т. е. без параметров), что подходит в нашем случае
    //   (см. конструктор базового класса LINSN). Если конструктор
    //   базового класса требует параметров, то конструктор
    //   производного класса должен вызывать базовый конструктор,
    //   используя список инициализации элементов. При разрушении
    //   объекта деструкторы вызываются в обратном порядке

    // Методы

public:

    // Добавить элемент add после элемента find
    void After_Add( char find, char add );

    // Добавить элемент add перед элементом find
    void Before_Add( char find, char add );

}; // Конец объявления производного класса

// ***** Реализация методов производного класса *****

//*****

```

```

// Добавление элемента add после элемента find
void LINSN_N :: After_Add( char find, char add )
{
    ELEM *temp,      // Указатель на добавляемый элемент
          *cur;       // Указатель на текущий элемент

    // Линейный список пуст?
    if( !start )
    {
        printf( "\n The list is empty. It is impossible"
                " to find the necessary unit " );
        return;
    }

    // Поиск элементов, содержащих символ find, с добавлением после
    // них элемента с символом add
    cur = start;
    while( cur )
    {
        if( cur->ch == find )
        {
            // Нужный элемент найден (он является текущим)
            // Динамическое размещение элемента
            temp = new ELEM;
            if( !temp )
            {
                printf( "\n The unit of the list is not placed " );
                exit( 1 );
            }
            // Занесение данных
            temp->ch = add;
            // Ссылка на элемент, который стоял за текущим
            temp->next = cur->next;
            // Текущий элемент указывает на новый
            cur->next = temp;
            // Новый элемент становится текущим
            cur = temp;
        }
        // Продвижение по списку
        cur = cur->next;
    }

    return;
}

//*****

```

```

// Добавление элемента add перед элементом find
void LINSF_N :: Before_Add( char find, char add )
{
    ELEM *temp,      // Указатель на добавляемый элемент
          *cur,      // Указатель на текущий элемент
          *prev;     // Указатель на элемент, стоящий перед текущим

    // Линейный список пуст?
    if( !start )
    {
        printf( "\n The list is empty. It is impossible"
                " to find the necessary unit " );
        return;
    }

    // Поиск элементов, содержащих символ find, с добавлением перед
    // ними элемента с символом add
    cur = start;
    while( cur )
    {
        // Нужный элемент найден (он является текущим)
        if( cur->ch == find )
        {
            // Динамическое размещение элемента
            temp = new ELEM;
            if( !temp )
            {
                printf( "\n The unit of the list is not placed " );
                exit( 1 );
            }
            // Занесение данных
            temp->ch = add; // Занесение данных
            // Новый элемент указывает на элемент с символом find
            temp->next = cur;
            // Если элемент с символом find был первым
            if( cur == start )
                // start смещается влево (на новый элемент)
                start = temp;
            else
                // Элемент, стоящий перед cur, указывает на новый
                prev->next = temp;
        }
        // Продвижение текущего и предыдущего элементов по списку
        prev = cur; cur = cur->next;
    }
}

```

```

    }

    return;
}

#endif

```

Листинг 2.2. Файл LINSР_N.CPP

```

/*
    Тестирование иерархии классов для работы с однонаправленным линейным списком.
    Базовый класс LINSР определен в файле LINSР.H, и производный от него класс LINSР_N
    определен в файле LINSР_N.H.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Подключение производного и базового классов
#include "linsp_n.h"

// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    LINSР_N    ls;          // Определение ls классом типа LINSР_N

    // Использование функций, унаследованных от базового класса LINSР
    ls.Add_end( 'C' );      // Добавление в конец списка 'C'
    ls.Add_end( 'D' );      // Добавление в конец списка 'D'
    ls.Add_beg( 'B' );      // Добавление в начало списка 'B'
    ls.Add_beg( 'A' );      // Добавление в начало списка 'A'
    ls.Print_ls( );         // Печать содержимого списка
    ls.Del_end( );          // Удаление последнего элемента
    ls.Del_beg( );          // Удаление первого элемента
    ls.Print_ls( );
    ls.Del_all( );          // Удаление всего списка
    // Заполнение списка пятью двойками
    for( int i=1; i<=5; i++ )
        ls.Add_end( '2' );
    ls.Print_ls( );

    // Использование функций производного класса LINSР_N
    // (Befor_Add, After_Add)
    // Добавление '1' перед '2'
    ls.Before_Add( '2', '1' );
    ls.Print_ls( );
    // Добавление '3' после '2'
    ls.After_Add( '2', '3' );
}

```

```

ls.Print_ls( );
ls.Before_Add( '1', '1' );
ls.Print_ls( );
ls.After_Add( '2', '2' );
ls.Print_ls( );
ls.After_Add( '2', '2' );
ls.Print_ls( );
ls.After_Add( '3', '3' );
ls.Print_ls( );

return 0;
}

```

Листинг 2.3. Результаты выполнения программы

```

The designer is called(caused)
ABCD
BC
22222
1212121212
123123123123123
11231123112311231123
1122311223112231122311223
11222231122223112222311222231122223
1122223311222233112222331122223311222233
Destructor has completed operation
Press any key to continue

```

Внимательно проанализируйте листинги 2.1—2.3.

2.2. Доступ к членам базовых классов

К открытому члену базового класса можно обращаться из производного класса точно так же, как если бы он был просто членом производного класса. Как получить доступ к открытому члену базового класса из производного класса, если в производном классе также определен член с тем же именем? Для этой цели необходимо использовать оператор разрешения области видимости "::", перед которым следует указать имя базового класса.

Рассмотрим и проанализируем следующий пример (листинг 2.4).

Листинг 2.4. Файл ACCESS.CPP

```

/*
Доступ к членам базовых классов.
Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

```

```

//*****
// Внешняя функция, не являющаяся членом класса
void f( void )
{
    // ...
    return;
}

//*****
// Определение базового класса. Класс можно определять и с помощью
// служебного слова struct
struct BASE1
{
    // В этом случае по умолчанию члены класса являются общедоступными

    // Данные

    int          a;

    // Методы

    // Обратите внимание, что используется то же самое имя метода
    void f( int i )
    {
        a += i;

        return;
    }

    // ...
};

//*****
// Определение еще одного базового класса
struct BASE2
{
    // Данные

    // !!! Используется то же самое имя данного
    int          a;

    // Методы

    // !!! Используется то же самое имя метода

```

```

void f( char c )
{
    a += ( int )c;

    return;
}

// ...
};

/*****
// Определение производного класса - обратите внимание, как определяется
// заголовок
// Эквивалентно struct DERIV: public BASE1, public BASE2
struct DERIV: BASE1, BASE2
{
    // При использовании вместо служебного слова struct слова class
    // запись class DERIV: BASE1, BASE2 эквивалентна записи
    // class DERIV: private BASE1, private BASE2

    // Данные

    // !!! Обратите внимание на совпадение имени данного
    int      a;

    // Методы

    // !!! Обратите внимание на имя метода
    void f( void )
    {
        :: f( );          // Это вызов внешней функции, не являющейся
                           // членом класса

        // ...
        return;
    }

    // ...
};

/*****
// Тестирование классов
int main( void )          // Возвращает 0 при успехе
{
    DERIV      obj;        // Объект производного класса
    // ...
    obj.BASE1 :: a =        // Член-данное BASE1

```

```
obj.BASE2 :: a =      // Член-данные BASE2
obj.a = 1;           // Член-данные DERIV
obj.BASE1 :: f( 2 ); // Метод BASE1
// Метод BASE2
obj.BASE2 :: f( '\x1' );
obj.f( );           // Метод DERIV
:: f( );           // Обычная внешняя функция
// ...
return 0;
}
```

Закljučая рассмотрение примера, отмечаем, что для объектов производного класса доступны все открытые члены-данные и все открытые методы базовых классов, а также функции, не являющиеся членами классов и глобальные переменные.

Несколько забегаая вперед, отметим также, что *внутри* производного класса доступны все защищенные (`protected:`) члены базового класса.

В объявлении производного класса в списке базовых классов в общем случае могут использоваться модификаторы `public:`, `protected:` и `private:` (в рассмотренном ранее примере по умолчанию использовался модификатор `private:`). С помощью этих модификаторов доступ к членам базовых классов может быть еще ограничен (табл. 2.1). Это является ценным свойством языка C++ — признаком хорошего стиля ООП.

Таблица 2.1. Доступ к членам базового класса из производного класса

Доступ к членам базового класса в нем самом	Модификатор доступа в заголовке определения производного класса	Доступ к членам базового класса из производного класса	Эквивалентный модификатор доступа к членам базового класса для производного класса
<code>public:</code> (открытый)	<code>public:</code> (см. пример в файле ACCESS1.CPP)	Доступен	<code>public:</code>
<code>protected:</code> (защищенный)		Доступен	<code>protected:</code>
<code>private:</code> (закрытый)		Недоступен	<code>private:</code>
<code>public:</code>	<code>protected:</code> (см. пример в файле ACCESS2.CPP)	Доступен	<code>protected:</code>
<code>protected:</code>		Доступен	<code>protected:</code>
<code>private:</code>		Недоступен	<code>private:</code>
<code>public:</code>	<code>private:</code> (см. пример в файле ACCESS3.CPP)	Доступен	<code>private:</code>
<code>protected:</code>		Доступен	<code>private:</code>
<code>private:</code>		Недоступен	<code>private:</code>

Для иллюстрации материала, приведенного в табл. 2.1, рассмотрим еще три примера (листинг 2.5 иллюстрирует три первых строки таблицы, листинг 2.6. — три следующих

строки и листинг 2.7 — три последних строки). Эти примеры являются весьма важными и их нужно внимательно изучить вместе с табл. 2.1.

Листинг 2.5. Файл ACCESS1.CPP

```
/*
    Доступ из производного класса к членам базового класса (в заголовке производного
    класса для базового класса используется спецификатор доступа public:).
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

//*****
// Базовый класс
class B
{
    // Данные

private:
    int     privi;

protected:
    int     proti;

public:
    int     publi;

    // Методы

public:

    // Конструктор
    B( void )
    {
        privi = 10; proti = 20; publi = 30;
    }

};

//*****
// Производный класс для класса B
class D: public B
{
    // Методы

public:
```

```

void fprot( void )
{
    prot1++;

    return;
}

void fpub( void )
{
    pub1++;

    return;
}

};

//*****
// Производный класс для D и B
class DD: public D
{
    // Методы

public:
    void fprot( void )
    {
        prot1++;

        return;
    }

    void fpub( void )
    {
        pub1++;

        return;
    }
};

//*****
// Тестирование
int main( void )          // Возвращает 0 при успехе
{
    D          d;
    // Здесь d.B::priv1 = 10, d.B::prot1 = 20, d.B::pub1 = 30

```

```

d.fprot( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
// доступен B::proti, имеющий спецификатор доступа protected:.
// Получаем d.B::proti = 21

d.fpub( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
// доступен B::pubi, имеющий спецификатор доступа public:. Получаем
// d.B::pubi = 31

//d.B::proti++;
// Эквивалентный спецификатор доступа из класса D к B::proti -
// protected:. Поэтому из произвольной программной среды d.B::proti
// недоступен

d.B::pubi++;          // Эквивалентно d.pubi++; - спускаемся вниз по
                      // иерархии классов
// Эквивалентный спецификатор доступа из класса D к B::pubi -
// public:. Поэтому из произвольной программной среды d.B::pubi:
// доступен. Получаем d.B::pubi = 32

DD      dd;
// Здесь dd.B::privi = 10, dd.B::proti = 20, dd.B::pubi = 30

dd.fprot( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
// доступен B::proti, имеющий эквивалентный спецификатор доступа
// protected:. Получаем dd.B::proti = 21

dd.fpub( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
// доступен B::pubi, имеющий эквивалентный спецификатор доступа
// public:. Получаем dd.B::pubi = 31

//dd.B::proti++;
// Эквивалентный спецификатор доступа к B::proti - protected:.
// Поэтому из произвольной программной среды dd.B::proti недоступен

dd.B::pubi++;          // Эквивалентно dd.pubi++; - спускаемся вниз по
                      // иерархии классов
// Эквивалентный спецификатор доступа к B::pubi - public:. Поэтому из
// произвольной программной среды dd.B::pubi: доступен. Получаем
// dd.B::pubi = 32

return 0;
}

```

Листинг 2.6. Файл ACCESS2.CPP

```

/*
    Доступ из производного класса к членам базового класса (в заголовке производного
    класса для базового класса используется спецификатор доступа protected:).
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

//*****
// Базовый класс
class B
{
    // Данные

private:
    int     privi;

protected:
    int     proti;

public:
    int     publi;

    // Методы

public:

    // Конструктор
    B( void )
    {
        privi = 10; proti = 20; publi = 30;
    }

};

//*****
// Производный класс для класса B
class D: protected B
{
    // Методы

public:

    void fprot( void )
    {

```

```

        proti++;

        return;
    }

    void fpub( void )
    {
        publi++;

        return;
    }

};

//*****
// Производный класс для D и B
class DD: public D
{
    // Методы

public:

    void fprot( void )
    {
        proti++;

        return;
    }

    void fpub( void )
    {
        publi++;

        return;
    }

};

//*****
// Тестирование
int main( void )        // Возвращает 0 при успехе
{
    D        d;
    // Здесь d.B::privi = 10, d.B::proti = 20, d.B::pubi = 30

    d.fprot( );

```

```

// Метод имеет спецификатор public: и поэтому доступен. Из него
//  доступен B::proti, имеющий спецификатор доступа protected:.
//  Получаем d.B::proti = 21

d.fpub( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
//  доступен B::pubi, имеющий спецификатор доступа public:. Получаем
//  d.B::pubi = 31

//d.B::proti++;
// Эквивалентный спецификатор доступа из класса D к B::proti -
//  protected:. Поэтому из произвольной программной среды d.B::proti
//  недоступен

//d.B::pubi++;
// Эквивалентный спецификатор доступа из класса D к B::pubi -
//  protected:. Поэтому из произвольной программной среды d.B::pubi
//  недоступен.

DD      dd;
// Здесь dd.B::privi = 10, dd.B::proti = 20, dd.B::pubi = 30

dd.fprot( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
//  доступен B::proti, имеющий эквивалентный спецификатор доступа
//  protected:. Получаем dd.B::proti = 21

dd.fpub( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
//  доступен B::pubi, имеющий эквивалентный спецификатор доступа
//  protected:. Получаем dd.B::pubi = 31

//dd.B::proti++;
// Эквивалентный спецификатор доступа к B::proti - protected:.
//  Поэтому из произвольной программной среды dd.B::proti недоступен

//dd.B::pubi++;
// Эквивалентный спецификатор доступа к B::pubi - protected:. Поэтому
//  из произвольной программной среды dd.B::pubi недоступен.

return 0;
}

```

Листинг 2.7. Файл ACCESS3.CPP

```

/*
    Доступ из производного класса к членам базового класса (в заголовке производного
    класса для базового класса используется спецификатор доступа private:).
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

//*****
// Базовый класс
class B
{
    // Данные

private:
    int     privi;

protected:
    int     proti;

public:
    int     publi;

    // Методы

public:

    // Конструктор
    B( void )
    {
        privi = 10; proti = 20; publi = 30;
    }

};

//*****
// Производный класс для класса B
class D: private B
{
    // Методы

public:

    void fprot( void )
    {

```

```

        proti++;

        return;
    }

    void fpub( void )
    {
        publi++;

        return;
    }

};

//*****
// Производный класс для D и B
class DD: public D
{
    // Методы

public:

/*    void fprot( void ) // Метод недоступен из-за недоступности proti
    {
        proti++;          // Недоступен, так как имеет эквивалентный
                           // спецификатор доступа private:

        return;
    }

    void fpub( void )      // Метод недоступен из-за недоступности publi
    {
        publi++;          // Недоступен, так как имеет эквивалентный
                           // спецификатор доступа private:

        return;
    } */

};

//*****
// Тестирование
int main( void )          // Возвращает 0 при успехе
{
    D          d;

```

```

// Здесь d.B::privi = 10, d.B::proti = 20, d.B::pubi = 30

d.fprot( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
// доступен B::proti, имеющий спецификатор доступа protected:.
// Получаем d.B::proti = 21

d.fpub( );
// Метод имеет спецификатор public: и поэтому доступен. Из него
// доступен B::pubi, имеющий спецификатор доступа public:. Получаем
// d.B::pubi = 31

//d.B::proti++;
// Эквивалентный спецификатор доступа из класса D к B::proti -
// private:. Поэтому из произвольной программной среды d.B::proti
// недоступен

//d.B::pubi++;
// Эквивалентный спецификатор доступа из класса D к B::pubi -
// private:. Поэтому из произвольной программной среды d.B::pubi
// недоступен.

DD      dd;
// Здесь dd.B::privi = 10, dd.B::proti = 20, dd.B::pubi = 30

dd.fprot( );
// Спускаясь по иерархии классов вниз, здесь вызывается метод класса
// D, имеющий спецификатор public: и поэтому доступный. Из него
// доступен B::proti, имеющий спецификатор доступа protected:.
// Получаем dd.B::proti = 21

dd.fpub( );
// Спускаясь по иерархии классов вниз, здесь вызывается метод класса
// D, имеющий спецификатор public: и поэтому доступный. Из него
// доступен B::pubi, имеющий спецификатор доступа public:. Получаем
// dd.B::pubi = 31

//dd.B::proti++;
// Эквивалентный спецификатор доступа к B::proti - private:. Поэтому
// из произвольной программной среды dd.B::proti недоступен

//dd.B::pubi++;
// Эквивалентный спецификатор доступа к B::pubi - private:. Поэтому
// из произвольной программной среды dd.B::pubi недоступен

return 0;
}

```

2.3. Виртуальные базовые классы

Синтаксис языка C++ запрещает непосредственную передачу базового класса в производный более одного раза:

```
class A
{
    // ...
};

class B : A, A           // Ошибка
{
    // ...
};
```

Если повторная передача базового класса в производный класс выполняется *косвенно*, то это также недопустимо:

```
class A
{
    // ...
};

class B : public A
{
    // ...
};

// Ошибка
class C : public A, public B
{
    // ...
};
```

В подобных случаях следует использовать *виртуальные базовые классы*. При "обычном" наследовании объект производного класса содержит в своем составе *подобъект* базового класса, тогда как при виртуальном наследовании объект производного класса содержит скрытый *указатель на подобъект* виртуального базового класса. Этот указатель компилятор неявно использует при работе с объектом для доступа к членам, унаследованным от виртуального базового класса (рис. 2.1).

```
class A
{
    // ...
};

class B : virtual public A
{
    // ...
};

// Так можно
class C : virtual public A, public B
{
    // ...
};
```

Приведем иллюстрирующий пример (листинги 2.8, 2.9).

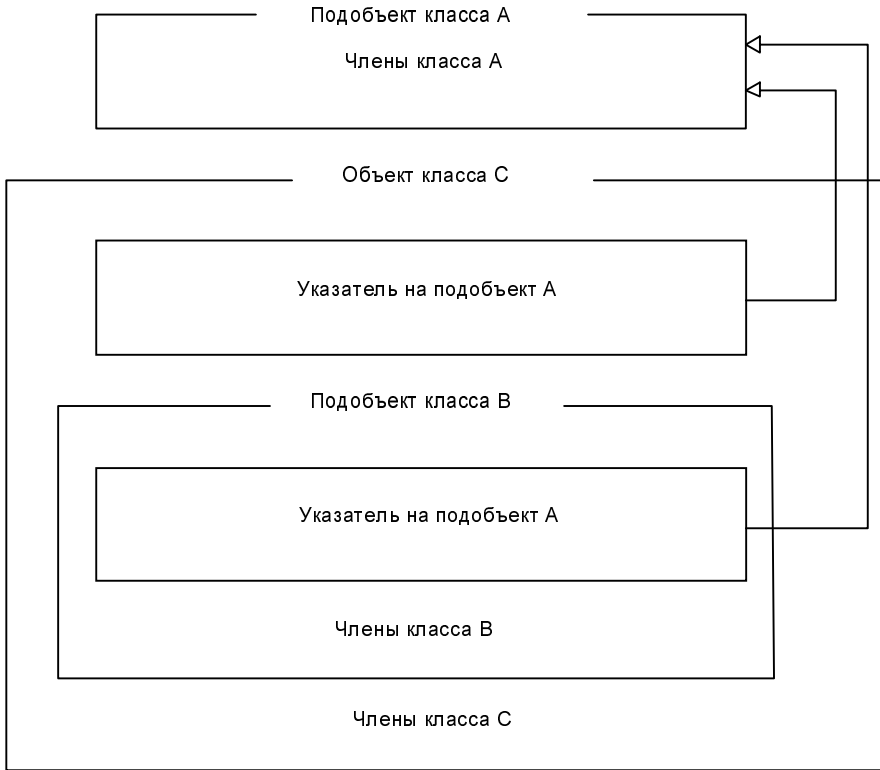


Рис. 2.1. Виртуальное наследование

Листинг 2.8. Файл VIRTUAL.CPP

```

/*
    Виртуальные базовые классы.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream.h>    // Для потокового ввода-вывода

//*****
// Базовый класс для классов В, В1 и С
class A
{
    // Данные

private:
    long    i;

    // Методы

public:

```

```

// Конструктор: подставляемый метод
A( long init_i )
{
    cout << " The constructor A is called " << endl;
    i = init_i;
}

// Деструктор: подставляемый метод
~A( void )
{
    cout << " Is called destructor A" << endl;
}

// Получение значения i: подставляемый метод
long get_i( void )
{
    return i;
}

};

//*****
// Базовый класс для C и производный для A
class B: virtual public A
{
    // Методы

public:

    // Конструктор: подставляемый метод. !!! Обратите внимание на
    // то, как конструктор производного класса инициализирует
    // конструктор базового класса
    B( long i ):
        A( i )           // Конструктор базового класса
    {
        cout << " Is called constructor B" << endl;
    }

    // Деструктор: подставляемый метод.
    ~B( void )
    {
        cout << " Is called destructor B" << endl;
    }

};

```

```
    //*****
// Производный класс для А
class B1: public A
{
    // Методы

public:

    // Конструктор: подставляемый метод
    B1( long i ):
        A( i )           // Вызов конструктора базового класса
    {
        cout << " The constructor B1 is called" << endl;
    }

    // Деструктор: подставляемый метод
    ~B1( void )
    {
        cout << " Is called destructor B1" << endl;
    }

};

//*****
// Производный класс для В и производный для А
class C: public B, virtual public A
{
    // Методы

public:

    // Конструктор: подставляемый метод. !!! Обратите внимание на
    // то, как инициализируются конструкторы А( ) и В( )
    C( long j ):
        B( j ), A( j )    // Вызов конструкторов базовых классов
    {
        cout << " The constructor C is called " << endl;
    }

    // Деструктор: подставляемый метод
    ~C( void )
    {
        cout << " Is called destructor C" << endl;
    }

};
```

```

//*****
// Тестирование
int main( void )          // Возвращает 0 при успехе
{
    cout << " We create the object of the class A " << endl;
    A      a( 1 );
    cout << " Size of the object a=" << sizeof( a ) << " byte"
        << endl;

    cout << " We create the object of the class B1" << endl;
    B1     b1( 1 );
    cout << " Size of the object b1=" << sizeof( b1 ) << " byte"
        << endl;

    cout << " We create the object of the class B" << endl;
    B      b( 1 );
    cout << " Size of the object b=" << sizeof( b ) << " byte"
        << endl;

    cout << " We create the object of the class C" << endl;
    C      c( 1 );
    cout << " Size of the object c=" << sizeof( c ) << " byte"
        << endl;

    return 0;
}

```

Листинг 2.9. Результаты выполнения программы

```

We create the object of the class A
The constructor A is called
Size of the object a=4 byte
We create the object of the class B1
The constructor A is called
The constructor A1 is called
Size of the object b1=4 byte
We create the object of the class B
The constructor A is called
Is called constructor A
Size of the object b=8 byte
We create the object of the class C
The constructor A is called
Is called constructor A
The constructor N is called
Size of the object c=8 byte

```

```

Is called destructor N
Is called destructor B
Is called destructor A
Is called destructor B
Is called destructor A
Is called destructor B1
Is called destructor A
Is called destructor A
Press any key to continue

```

Сделаем несколько замечаний относительно данной программы:

1. В Size of the object b=8 byte входят 4 байта на указатель на подобъект виртуального базового класса A и 4 байта занимает член-данное базового класса A.
2. В Size of the object c=8 byte входят 4 байта на указатель на подобъект виртуального базового класса A (указатель хранится в одном экземпляре) и 4 байта занимает член-данное базового класса A.
3. Обратите внимание на порядок вызова конструкторов и деструкторов производных и базовых классов.
4. Советуем внимательно проанализировать работу этой программы.

2.4. Рекомендации по составу класса.

Отличия структур и объединений от классов

Обычно, класс как пользовательский тип должен содержать некоторое количество скрытых (**private:**) членов-данных и следующие методы:

- ☐ *конструкторы*, определяющие способы инициализации объектов класса и, при необходимости, обеспечивающие резервирование динамической памяти;
- ☐ *конструктор копирования* (необходим, если класс использует динамическую память);
- ☐ *деструктор* (необходим, если класс использует динамическую память);
- ☐ *набор методов, реализующих свойства класса* (при этом методы, возвращающие значения скрытых членов-данных класса, должны описываться с модификатором **const**, указывающим, что они не должны изменять значения полей);
- ☐ *набор операций*, позволяющий присваивать, сравнивать объекты, выполнять над объектами арифметические и другие действия, требующиеся в классе;
- ☐ *средства обработки исключений*, используемые для сообщений об ошибках.

Очень важными при проектировании класса являются следующие вопросы. Как определить состав членов-данных класса и их доступность? Как определить состав методов класса и их доступность?

Если некоторое данное используется более чем в одном методе класса, или даже в единственном методе, который при решении задачи часто вызывается, то такое данное следует сделать членом класса. При этом если такой член-данное используется только методами данного класса, то его следует закрыть (снабдить спецификатором

доступа `private:`). Если член-данное должен быть доступен в этом классе и производных от него классах, то его следует защитить (снабдить спецификатором доступа `protected:`). Открывать члены-данные класса не следует практически никогда, так как при этом класс становится уязвимым, что делает программный код ненадежным.

При проектировании методов класса руководствуются следующими соображениями. В качестве метода класса следует использовать функцию, реализующую решение некоторой функционально-законченной задачи. Для обеспечения разумного размера метода можно использовать "правило 7 ± 2 ", смысл которого рассмотрен в [3]. В соответствии с этим правилом размер метода не должен превышать 25—80 строк исходного текста, а количество параметров не должно превышать 5—7. Выполнение этих требований можно обеспечить путем надлежащей иерархической декомпозиции "больших" методов. Все разновидности конструкторов, деструкторы и интерфейсные методы класса, с помощью которых пользователь взаимодействует с классом, нужно сделать доступными из любой программной среды (`public:`). Вспомогательные, не интерфейсные методы следует либо закрыть (`private:`), если они нужны только в этом классе, либо защитить (`protected:`), если они нужны в этом и производных от него классах.

В гл. 2 указывалось, что наряду с классом, в качестве альтернативы, можно использовать структуры и объединения. *Чем же отличаются структуры и объединения от классов?* Структуры и объединения, по своей сути, представляют частные случаи классов.

Структуры отличаются от классов тем, что в них доступ к членам структуры по умолчанию считается `public:`. Аналогично, при объявлении производной структуры в ее заголовке, по умолчанию, перед именем базового класса используется спецификатор доступа `public:`. Структуры предпочтительнее использовать для классов, все или большинство членов которых доступны из произвольной программной среды. Вообще говоря, без структур вполне можно было бы обойтись. Они были введены в язык C++ с единственной целью — обеспечить совместимость программного кода между языками C и C++ снизу вверх.

Основные отличия объединений от классов состоят в следующем:

- ☐ доступ к членам объединения по умолчанию `public:`, а явное использование спецификатора доступа не разрешено;
- ☐ объединение не может участвовать в иерархии классов;
- ☐ членами объединения не могут быть объекты с конструкторами и деструкторами;
- ☐ объединение может иметь конструктор и другие методы, но не может содержать статических членов.

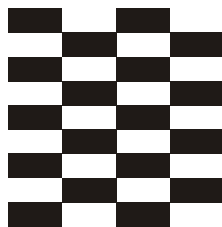
2.5. Вопросы и упражнения для самопроверки [4]

1. Методы базового и производных классов могут иметь одинаковые имена. Как при этом можно будет различать разные варианты этих методов?
2. Наследуются ли данные и методы базового класса в последующие поколения производных классов?

3. Пусть класс `Three` произведен от класса `Two`, а класс `Two` произведен от класса `One`. В классе `Two` замещен метод, описанный в классе `One`. Какой вариант этого метода получит класс `Three`?
4. Можно ли в производном классе описать как `private`: метод, который перед этим был описан в базовом классе как `public`:?
5. С какой целью используется служебное слово `protected`:?
6. Запишите объявление класса `Square` (квадрат), произведенного от класса `Rectangle` (прямоугольник), который, в свою очередь, произведен от класса `Form` (форма).
7. Класс `Square` (квадрат) произведен от класса `Rectangle` (прямоугольник), который, в свою очередь, произведен от класса `Form` (форма). Объект класса `Form` не использует параметры, объект класса `Rectangle` принимает два параметра целого типа (`length` и `width`), а объект класса `Square` — один параметр (`length`). Напишите конструктор класса `Square`, поместив его определение вне блока класса.

Ответы на вопросы и решения для упражнений приведены в *разд. П1.2 приложения 1*.

Глава 3



Полиморфизм

Третьим принципом ООП является *полиморфизм*. Слово это можно перевести с греческого как "многоформенность". Полиморфизм (перегрузка операций, перегрузка функций, шаблоны функций и классов) означает, что одно и то же имя операции, функции или класса может использоваться для различных типов данных.

В языке C++ полиморфизм имеет две формы.

- ❑ Перегрузка операций и функций, реализуемая через механизм *статического связывания*. Сюда же относятся шаблоны функций и классов. Эти средства языка C++ были рассмотрены в [3].

Статическое связывание предполагает, что определение конкретного экземпляра операции, функции или класса (это и называется связыванием) выполняется *на этапе компиляции* программы.

- ❑ Использование так называемых *виртуальных функций*, что реализуется через механизм *динамического связывания*. Эта форма является основной формой полиморфизма как одного из важнейших принципов ООП.

Динамическое связывание, в отличие от статического, означает, что определение конкретного экземпляра виртуальной функции производится не на этапе компиляции, а непосредственно *во время выполнения* программы.

Рассмотрим подробнее вторую форму полиморфизма.

3.1. Виртуальные методы классов

При вызове метода некоторого класса у компилятора есть несколько рассмотренных нами способов связать имя вызываемой функции с соответствующим ей машинным кодом на стадии компиляции.

Однако если для вызова метода класса используется указатель на класс, у которого есть производные и базовые классы, имеющие методы с тем же именем и одинаковым набором аргументов (с той же *сигатурой*), то компилятор *не в состоянии* определить, к какому конкретно классу относится указываемый объект и какой метод следует для него вызвать.

Для разрешения этой коллизии язык C++ предусматривает возможность использования *виртуальных функций*. Если в некотором классе имеется метод, описанный как `virtual`, то в такой класс добавляется скрытый член-указатель на таблицу виртуальных функций,

а также генерируется специальный код, позволяющий осуществить выбор виртуального метода, подходящего для объекта данного типа, *во время работы программы, а не во время компиляции* (позднее связывание метода, динамическое связывание метода).

Если некоторый метод объявлен в базовом классе как виртуальный, то метод с тем же именем, с таким же списком параметров и тем же типом возвращаемого значения (метод с такой же сигнатурой), переопределенный в производном классе, автоматически становится виртуальным. Ключевое слово `virtual`: при его описании в производном классе уже можно не использовать.

Приведем еще ряд важных правил работы с виртуальными методами.

- ❑ В производном классе нельзя переопределять метод, отличающийся от виртуального метода базового класса только типом возвращаемого значения.
- ❑ Виртуальный метод должен быть методом некоторого класса, но не может быть статическим методом. Но виртуальный метод может быть объявлен как дружественный (лучше, без необходимости, не пользоваться последней возможностью).
- ❑ Если виртуальный метод не был переопределен в производном классе, то при его вызове для объекта этого производного класса будет происходить обращение к соответствующему виртуальному методу из ближайшего по иерархической лестнице базового класса, в котором он определен.
- ❑ Членом производного класса вполне может быть метод с именем, совпадающим с именем виртуального метода базового класса, но с другой сигнатурой. Тогда он будет другим, не виртуальным методом.
- ❑ Виртуальные методы *наследуются*, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменять *нельзя*.
- ❑ Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к виртуальному методу базового класса с помощью операции доступа к области видимости.

Далее представлен пример программы, позволяющей сравнить виртуальные и не виртуальные методы, вызываемые различными способами (листинг 3.1). *Это важный пример и его следует внимательно изучить.*

Листинг 3.1. Файл VIRTUALF.CPP

```
/*
    Обычные и виртуальные методы класса.
    В. Давыдов, Т. Сидорина, консольное приложение,
    Microsoft Visual Studio C++ 6.0
*/

#include <iostream.h>      // Для потокового ввода-вывода

// *****
// Базовый класс для Deriv1, Deriv2, Deriv2
class Base
```

```

{
    // Данные

private:

    int      d;

    // Методы

public:

    // Каждый метод этого и остальных классов при вызове сообщает
    // информацию о себе: обычный метод
    void f( void )
    {
        cout << "The function is called: " << "Base::f( )" << endl;

        return;
    }

    // Виртуальный метод с тем же именем: перед типом возвращаемого
    // значения использовано служебное слово virtual
    virtual void f( int i )
    {
        cout << "The virtual function is called: Base::f( int )" << endl;

        return;
    }

    // Перегрузка обычного метода внутри класса - сигнатуры методов
    // различны, имена совпадают
    void f( int i, int j )
    {
        cout << "The function is called: Base::f( int, int )" << endl;

        return;
    }

};

// *****
// Производный класс для Base и базовый для Deriv2
class Deriv1: public Base
{
    // Данные

```

```
private:
```

```
    char    d;
```

```
    // Методы
```

```
public:
```

```
    // Переопределение обычного метода в производном классе: сигнатуры
    // методов в базовом и производном классах одинаковы
```

```
    void f( void )
```

```
    {
```

```
        cout << "The function is called: Deriv1::f( )" << endl;
```

```
        return;
```

```
    }
```

```
    // Переопределение виртуального метода в производном классе:
    // сигнатуры совпадают, virtual можно опустить
```

```
    virtual void f( int i )
```

```
    {
```

```
        cout << "The virtual function is called: Deriv1::f( int )"
            << endl;
```

```
        return;
```

```
    }
```

```
};
```

```
// *****
```

```
// Производный класс для Base
```

```
class Deriv2: public Base
```

```
{
```

```
    // Методы
```

```
public:
```

```
    // Переопределение обычного метода в производном классе: сигнатуры
    // методов в базовом и производном классах одинаковы
```

```
    void f( void )
```

```
    {
```

```
        cout << "The function is called: Deriv2::f( )" << endl;
```

```
        return;
```

```
    }
```

```
    // Переопределение виртуального метода в производном классе:
```

```

// сигнатуры совпадают, virtual можно опустить
virtual void f( int i )
{
    cout << "The virtual function is called: Deriv2::f( int )"
        << endl;

    return;
}

};

// *****
// Производный класс для Deriv1 и Base
class Deriv12: public Deriv1
{
    // Данные

private:

    char    d;

    // В этом классе обычный метод f( ) и виртуальный метод f( int ) не
    // переопределены

};

// *****
// Тестирование работы с обычными и виртуальными методами с
// использованием объектов или указателей на них

typedef Deriv1* PDeriv1;

int main( void )        // Возвращает 0 при успехе
{
    // Создаем объекты различных классов
    Base    b_obj;
    Deriv1   d1_obj;
    Deriv2   d2_obj;
    Deriv12  d12_obj;

    cout << "Call of methods with usage of objects" << endl;
    cout << "*****" << endl;
    // Для объекта класса Base вызываем обычный и виртуальный методы
    b_obj.f( );
    b_obj.f( 1 );
    // Для объекта класса Deriv2 вызываем его виртуальный метод

```

```

d2_obj.f( 1 );
// Для объекта класса Deriv2 вызываем виртуальный метод из класса
//   Base
d2_obj.Base::f( 1 );
// Для объекта класса Deriv2 вызываем обычный метод из класса Base
// d2_obj.f( 1, 1 ); // Ошибка! Такой функции в классе Deriv2 нет
// Так правильно!
d2_obj.Base::f( 1, 1 );
/* Вывод на экран имеет вид:
Call of methods with usage of objects
*****
The function is called: Base::f( )
The virtual function is called: Base::f( int )
The virtual function is called: Deriv2::f( int )
The virtual function is called: Base::f( int )
The function is called: Base::f( int, int )
*/

// Массив из 4 указателей на Base и его инициализация адресами
//   созданных объектов
Base      *b_ptr[ 4 ];
b_ptr[ 0 ] = &b_obj;
// Неявное преобразование Deriv1 * в Base * (так делать можно - адрес
//   объекта производного класса преобразуется к адресу базового
//   класса)
b_ptr[ 1 ] = &d1_obj;
b_ptr[ 2 ] = &d2_obj; // Неявное преобразование Deriv2 * в Base *
// Неявное преобразование Deriv12 * в Base *
b_ptr[ 3 ] = &d12_obj;

// Имена классов
char      *types[ 4 ] = { "Base.", "Deriv1.", "Deriv2.",
                          "Deriv12." };

cout << endl << "Call of methods with usage of pointers on the class"
    << endl;
cout << "*****" << endl
    << endl;

cout << "The not virtual methods f( ) *****"
    << endl;

cout << "      The type of the pointer Base *" << endl;

// Программа при работе будет сообщать тип указателя, тип
//   указываемого объекта и к какому классу принадлежит вызванная

```

```
// функция
// Какая функция будет вызвана для b_ptr[ 3 ]?
for( int count = 0; count < 4; count++ )
{
    cout << "The type of the directed object: " << types[ count ]
        << endl;
    b_ptr[ count ]->f( );
}
/* Далее на экран будет выведено:
```

Call of methods with usage of pointers on the class

The not virtual methods f() *****

The type of the pointer Base *

The type of the directed object: Base.

The function is called: Base::f()

The type of the directed object: Deriv1.

The function is called: Base::f()

The type of the directed object: Deriv2.

The function is called: Base::f()

The type of the directed object: Deriv12.

The function is called: Base::f()

*/

```
cout << "          The type of the pointer Deriv1 *" << endl;
```

```
// Какая функция будет вызвана для b_ptr[ 3 ]?
```

```
for( count = 0; count < 4; count++ )
```

```
{
```

```
    cout << "The type of the directed object: " << types[ count ]
        << endl;
```

```
    // Для приведения типа базового класса к типу производного
```

```
    // требуется явное приведение типа: PDeriv1( b_ptr[ count ] )
```

```
    PDeriv1( b_ptr[ count ] )->f( );
```

```
}
```

```
/* Далее на экран будет выведено:
```

```
The type of the pointer Deriv1 *
```

The type of the directed object: Base.

The function is called: Deriv1::f()

The type of the directed object: Deriv1.

The function is called: Deriv1::f()

The type of the directed object: Deriv2.

The function is called: Deriv1::f()

The type of the directed object: Deriv12.

The function is called: Deriv1::f()

```

*/

cout << endl << "The virtual methods f( ) *****"
    << endl;

cout << "          The type of the pointer Base *" << endl;

// Какая функция будет вызвана для b_ptr[ 3 ]?
for( count = 0; count < 4; count++ )
{
    cout << "The type of the directed object: " << types[ count ]
        << endl;
    b_ptr[ count ]->f( 1 );
}

/* Далее на экран будет выведено:

The virtual methods f( ) *****
    The type of the pointer Base *
The type of the directed object: Base.
The virtual function is called: Base::f( int )
The type of the directed object: Deriv1.
The virtual function is called: Deriv1::f( int )
The type of the directed object: Deriv2.
The virtual function is called: Deriv2::f( int )
The type of the directed object: Deriv12.
The virtual function is called: Deriv1::f( int )
*/

cout << "          The type of the pointer Deriv1 *" << endl;

// Какая функция будет вызвана для b_ptr[ 3 ]?
for( count = 0; count < 4; count++ )
{
    cout << "The type of the directed object: " << types[ count ]
        << endl;
    PDeriv1( b_ptr[ count ] )->f( 1 );
}

/* Далее на экран будет выведено:
    The type of the pointer Deriv1 *
The type of the directed object: Base.
The virtual function is called: Base::f( int )
The type of the directed object: Deriv1.
The virtual function is called: Deriv1::f( int )
The type of the directed object: Deriv2.
The virtual function is called: Deriv2::f( int )

```

```

The type of the directed object: Derivl2.
The virtual function is called: Derivl::f( int )
*/

cout << endl;

// Вот как можно заставить компилятор вызвать виртуальную функцию, не
// соответствующую типу объекта, на который указывает указатель
PDerivl( b_ptr[ 2 ] )->Derivl::f( 1 );
/* Далее на экран будет выведено:

The virtual function is called: Derivl::f( int )
Press any key to continue
*/

return 0;
}

```

Рассмотренный пример позволяет сделать следующие важные выводы.

- ❑ Если вызов обычного или виртуального метода выполняется для объекта класса с использованием операции ".", то вызывается метод соответствующего класса. Можно для объекта производного класса вызвать метод какого-либо из базовых классов. Для этого достаточно между операцией "." и именем функции указать имя класса и операцию разрешения области видимости "::". Из сказанного можно заключить, что в такой ситуации использование виртуальных методов ничего не дает и лучше использовать обычные методы.
- ❑ Если вызов обычных, не виртуальных методов выполняется с использованием указателя на класс посредством операции ">", то независимо от того, адрес какого объекта присвоен указателю, всегда вызывается обычный метод из класса, соответствующего типу указателя. Чтобы этого избежать, следует использовать виртуальные методы.
- ❑ Указатель на базовый класс может указывать на объект базового или производного класса. Выбор виртуального метода зависит от класса, на объект которого указывается, а не от типа указателя! При отсутствии переопределенного виртуального метода в производном классе берется виртуальный метод базового класса. Следовательно, виртуальные методы в иерархии классов нужны для выполнения в классах различной работы, если работа ведется с использованием указателей.
- ❑ Чтобы вызвать виртуальный метод, не соответствующий типу объекта, на который указывает указатель, достаточно привести указатель к нужному типу и между операцией ">" и именем метода указать имя класса, к которому выполнено приведение типа, и операцию разрешения области видимости "::" (см. в примере конец функции main).

При внимательном рассмотрении примера возникает также вопрос — а как реализуется механизм виртуального вызова, если виртуальные функции являются подставляемыми (они же определены внутри блока класса)?

Компиляторы языка C++ поступают весьма разумно. Подстановка тела подставляемой виртуальной функции осуществляется только при тех вызовах, где механизм виртуального вызова игнорируется.

Как же реализуется *механизм виртуального вызова*? Для каждого класса (но не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает *таблицу виртуальных функций* (v-таблицу), в которой для каждого виртуального метода записан его адрес в памяти. Адреса виртуальных методов содержатся в v-таблице в том же порядке, что и в блоке класса.

Каждый объект класса с виртуальными методами содержит скрытый дополнительный член-ссылку на v-таблицу, называемый `vptr`. Этот указатель инициализируется конструктором при создании объекта. Для этого компилятор добавляет в начало тела конструктора соответствующие инструкции.

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к v-таблице через `vptr` объекта. На этапе же выполнения, в момент обращения к виртуальному методу, его адрес выбирается из v-таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, сопряжен с накладными расходами — он выполняется через дополнительный этап получения адреса метода из таблицы, а на хранение v-таблицы расходуется дополнительная память. Это замедляет выполнение программы.

3.2. Виртуальные деструкторы

Конструктор класса *не может* быть виртуальным, поскольку он вызывается только при создании объекта и при этом, конечно же, тип создаваемого объекта известен.

Может ли деструктор быть виртуальным? Да, конечно. Необходимость этого связана с тем, что в программах может использоваться оператор `delete`, разрушающий объект, адресуемый указателем на базовый класс. А вдруг этот указатель на самом деле указывает на объект производного класса, имеющий свой собственный деструктор? В подобных случаях проблема корректного разрушения указываемых объектов решается при помощи *виртуальных деструкторов*, которые аналогичны виртуальным методам. Если в подобном случае деструктор будет объявлен как виртуальный, то все произойдет правильно — будет вызван деструктор соответствующего производного класса. Затем деструктор производного класса автоматически вызовет деструктор базового класса и указанный объект будет удален целиком.

Отсюда следует *правило*: если в классе объявлены виртуальные методы, то и деструктор должен быть виртуальным.

Рассмотрим иллюстрирующий пример (листинги 3.2, 3.3).

Листинг 3.2. Файл VIRTDEST.CPP

```
/*  
    Виртуальные деструкторы.  
    Консольное приложение, Microsoft Visual Studio C++ 6.0  
*/  
  
#include <iostream.h>    // Для потокового ввода/вывода
```

```
// *****
// Базовый класс
class Figure
{
    // Методы

public:

    Figure( void )          // Конструктор умолчания
    {
        cout << "The constructor Figure" << endl;
    }

    // Виртуальный деструктор
    virtual ~Figure( void )
    {
        cout << "The destructor Figure" << endl;
    }

    // ...
};

// *****
// Производный класс для Figure
class Circle: public Figure
{
    // Данные

private:

    int      x,          // x-координата центра
            y,          // y-координата центра
            r;          // Радиус окружности

    // ...

    // Методы

public:

    // Конструктор
    Circle( int CenterX, int CenterY, int Radius )
    {
        cout << "The constructor Circle" << endl;
        x = CenterX; y = CenterY; r = Radius;
    }
}
```

```

    }

    // Виртуальный деструктор
    virtual ~Circle( void )
    {
        cout << "The destructor Circle" << endl;
    }
    // !!! Деструктор производного класса при виртуальном деструкторе
    // базового класса является также виртуальным. Поэтому здесь
    // virtual избыточно

    // ...
};

// *****
// Производный класс для Figure
class Rectangle: public Figure
{
    // Данные

private:

    int      l,          // Левый верхний
            t,          //   угол
            r,          // Правый нижний угол
            b;          //   прямоугольника

    // ...

    // Методы

public:

    // Конструктор
    Rectangle( int L, int T, int R, int B )
    {
        cout << "The constructor Rectangle" << endl;
        l = L; t = T; r = R; b = B;
    }

    // Виртуальный деструктор
    virtual ~Rectangle( void )
    {
        cout << "The destructor Rectangle" << endl;
    }

    // ...

```

```

};

// *****
// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    // ...

    // Создается массив указателей на базовый класс, который
    // инициализируется адресами объектов классов Circle и Rectangle
    Figure    *figures[ 2 ];
    figures[ 0 ] = new Circle( 100, 100, 10 );
    figures[ 1 ] = new Rectangle( 100, 100, 200, 250 );

    // ...

    // Уничтожаются созданные объекты
    delete figures[ 0 ]; delete figures[ 1 ];
    // Если бы деструкторы не были виртуальными, то при уничтожении
    // объектов вызывался бы ~Figure( ) - была бы ошибка

    // ...

    return 0;
}

```

Листинг 3.3. Результаты выполнения программы

```

The constructor Figure
The constructor Circle
The constructor Figure
The constructor Rectangle
The destructor Circle
The destructor Figure
The destructor Rectangle
The destructor Figure
Press any key to continue

```

3.3. Абстрактные методы и классы

Если при определении базового класса некоторый виртуальный метод объявлен членом этого класса, то он должен быть определен подобно обычным методам либо же объявлен как *абстрактный* или "чистый" метод при помощи спецификатора "=0":

```

class Base
{
    // ...
    // Абстрактный виртуальный метод класса

```

```

    virtual void f( int ) = 0;
    // ...
};

```

Класс, содержащий хотя бы один абстрактный метод, называется *абстрактным классом*. Абстрактный класс может служить только в качестве базового для других классов — объект абстрактного класса создать невозможно. В классах, производных от абстрактного класса, абстрактные методы должны быть либо определены, либо вновь объявлены как абстрактные.

Сформулируем наиболее важные правила работы с абстрактными методами и классами.

- ❑ Тип параметра метода не может быть абстрактным классом.
- ❑ Тип возвращаемого методом значения не может быть абстрактным классом.
- ❑ Разрешено (и это часто используется) создавать указатель на абстрактный базовый класс, а также ссылку на такой класс, если для ее инициализации не требуется создания временного объекта.
- ❑ Методы абстрактного класса могут вызывать абстрактные методы этого же класса. В подобном случае будет вызван соответствующий типу объекта метод, определенный в производном классе.
- ❑ Абстрактный класс может иметь конструкторы и деструктор. Конструктор абстрактного класса будет вызываться при создании объектов производных классов, а деструктор — при их разрушении. Следует иметь в виду, что деструктор базового класса вызывается при разрушении объекта в последнюю очередь, когда уже разрушены "подобъекты", определенные в производных классах. Поэтому деструктор абстрактного базового класса не должен вызывать абстрактные методы своего класса, так как такой вызов приведет к ошибке во время выполнения программы.
- ❑ Если в классе, производном от абстрактного класса, определены не все абстрактные функции, то производный класс также является абстрактным.
- ❑ Любой класс, произведенный от абстрактного класса, унаследует от него абстрактные методы. Чтобы получить возможность создания объекта производного класса, в нем нужно переопределить все абстрактные методы.
- ❑ Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться *только* в качестве базового класса для других классов.

Рассмотрим иллюстрирующий пример (листинг 3.4).

Листинг 3.4. Файл VRTFUN1.CPP

```

/*
    Виртуальные и абстрактные виртуальные методы.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdio.h>           // Для ввода-вывода

// *****

```

```
// Абстрактный базовый класс
class FIGURE
{
    // Методы

public:

    // Виртуальный метод может иметь возвращаемый тип и параметры.
    // Виртуальный метод, переопределяемый в производном классе, должен
    // иметь те же свойства - ту же сигнатуру. В этом примере метод
    // areal() нужен только для того, чтобы показать, что будет, если
    // это не выполнено

    // Виртуальный метод без аргументов (подставляемый метод)
    virtual double area( void )
    {
        return 0.0;
    }

    // Виртуальный метод с аргументом (подставляемый метод)
    virtual float areal( float tmp = 9.0f )
    {
        return tmp;
    }

    // Виртуальный метод ДОЛЖЕН БЫТЬ определен ( см. area( void ) ) или
    // описываться как чистый, т. е. как виртуальный метод, тело
    // которого не определено. Такой метод называется абстрактным, а
    // класс, его содержащий, - абстрактным классом. Пример чистого
    // метода:
    virtual void prn1( void ) = 0;

    // Не виртуальная подставляемая функция
    void prn2( void )
    {
        printf( "\n The base class " );

        return;
    }

    // Виртуальный подставляемый метод вычисления и печати значения
    // площади фигуры
    virtual void prn3( void )
    {
        printf( "\n Square of a figure area = %g", area( ) );
        printf( "\n Square of a figure areal( 5.0f ) = %g",
```

```
        areal( 5.0f ) );

    return;
}

};

// *****
// Класс CIRCLE, производный от абстрактного класса FIGURE
class CIRCLE : public FIGURE
{
    // Данные

private:

    double    radius;    // Радиус

    // Методы

public:

    // Конструктор с параметром: подставляемая функция
    CIRCLE ( double rad )
    {
        radius = rad;
    }

    // Вычисление площади фигуры. При переопределении виртуального метода
    // в производном классе служебное слово virtual не обязательно.
    // Подставляемый метод
    virtual double area( void )
    {
        return 3.14 * radius * radius;
    }

    // Вычисление площади фигуры. Подставляемый метод (не виртуальный -
    // другая сигнатура)
    double areal( void )
    {
        return 3.14 * radius * radius;
    }

    // Переопределение абстрактного метода (фактически его первое
    // определение)
    virtual void prn1( void )
    {
```

```

    printf( "\n Example of operation of the virtual function " );

    return;
}

// Не виртуальная функция (см. базовый класс )
void prn2( void )
{
    printf( "\n The derivative class " );

    return;
}

// Виртуальный метод вычисления и печати значения площади круга
virtual void prn3( void )
{
    printf( "\n Square of a circle area( ) = %g", area( ) );
    printf( "\n Square of a circle areal( ) = %g", areal( ) );

    return;
}

};

// *****
// Тестирование
int main( void )          // Возвращает 0 при успехе
{
    // Создать объект абстрактного класса нельзя, т. е. запись вида
    //  FIGURE figur; является ошибочной

    // Создаем объект производного класса
    CIRCLE    circl( 2.0 );

    // Вызываем все методы производного класса
    circl.prn2( ); circl.prn1( ); circl.prn3( );
    /* Вывод на экран:

The derivative class
Example of operation of the virtual function
Square of a circle area( ) = 12.56
Square of a circle areal( ) = 12.56
    */

    // Теперь еще раз о "тонкостях", связанных с использованием
    //  виртуальных методов. Чтобы использовать механизм виртуальных

```

```

// методов для абстрактного класса и производных от него классов,
// необходимо применять указатели или ссылки. Указатель на базовый
// класс может указывать на объект базового или производного
// класса. Выбор метода зависит от класса, на объект которого
// указывается, а не от типа указателя! При отсутствии
// переопределенного метода в производном классе берется
// виртуальный метод базового класса

// Базовый указатель на объект производного класса
FIGURE      *pfigure = new CIRCLE( 1.0 );

// Использование виртуального механизма - вызываются виртуальные
// методы производного класса
pfigure->prn1( ); pfigure->prn3( );
/* Продолжение вывода на экран:
Example of operation of the virtual function
Square of a circle area( ) = 3.14
Square of a circle areal( ) = 3.14
/*

// Вызов не виртуального метода (будет вызван метод базового класса)
pfigure->prn2( );
/* Продолжение вывода на экран:
The base class
/*

// Обход виртуального механизма (вызов виртуального метода базового
// класса)
pfigure->FIGURE :: prn3( );
/* Продолжение вывода на экран:
Square of a figure area = 3.14
Square of a figure areal( 5.0f ) = 5
/*

delete pfigure;
printf( "\n" );

return 0;
}

```

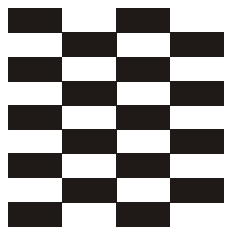
3.4. Вопросы для самопроверки [4]

1. Зачем в методе класса используют значения параметров, заданные по умолчанию, если можно его перегрузить?
2. Почему бы тогда всегда не использовать только методы с умалчиваемыми значениями параметров?

3. Может ли перегруженный метод содержать параметры, заданные по умолчанию?
4. Если вы перегрузили метод класса, то как потом можно будет различать разные варианты методов?
5. В каких случаях не следует делать методы класса виртуальными?
6. Предположим, что некоторый метод без параметров был описан в базовом классе как виртуальный, а затем перегружен таким образом, чтобы принимать один или два целочисленных параметра. Затем в производном классе был замещен вариант метода с одним целочисленным параметром. Что произойдет, если с помощью указателя, связанного с объектом производного класса, вызвать вариант метода с двумя параметрами?
7. Для чего возиться с абстрактным классом? Не проще ли создать обычный базовый класс, для которого в программе не создавать объекты?
8. Что такое `vptr`?

Ответы на вопросы можно проверить — они приведены в *разд. П1.3 приложения 1*.

Глава 4



Области действия и пространства имен [9]

В источнике [3] в *гл. 3 (разд. 3.3—3.6)* были рассмотрены области действия программных объектов и время их жизни применительно к возможностям языков С и С++. Однако возможности языка С++ в этой части гораздо богаче. Теперь самое время поговорить об этом подробнее.

4.1. Области действия и время жизни

Каждый программный объект имеет область действия и время жизни, которые определяются видом и местом его определения. Существуют следующие разновидности областей действия:

- ☐ блок;
- ☐ прототип функции;
- ☐ функция;
- ☐ файл;
- ☐ группа файлов, в пределе включающая все файлы программного проекта (глобальная область действия);
- ☐ класс;
- ☐ пространство имен (часть глобальной области действия).

Первые пять категорий областей действия программных объектов и время их жизни были довольно подробно рассмотрены в [3]. Кратко их напомним и обсудим две новые категории областей действия программных объектов — класс и пространство имен.

Блок. Напомним, что программный объект, определенный внутри блока, по области действия является локальным. *Область действия* такого объекта начинается в точке определения и заканчивается в конце блока. В область действия включаются вложенные (внутренние) блоки, если в них не содержится переопределение программного объекта с тем же идентификатором. *Время жизни* такого программного объекта, если он имеет описатель класса хранения `auto` (автоматический), начинается с момента его определения и заканчивается после выхода из блока (после завершения работы блока). При этом память, занятая программным объектом, освобождается. Если же программный объект определен внутри блока с описателем класса хранения `static` (статический), то время его жизни максимально и совпадает со временем выполнения программы.

Прототип функции. Идентификаторы, указанные в списке параметров прототипа (объявления) функции, имеют *область действия* только прототип функции. По этой причине можно использовать произвольные идентификаторы параметров функции и даже вообще их опускать. При этом все же отметим, что если в прототипе функции идентификаторы параметров используются, то их целесообразно выбрать такими же, как в заголовке определения функции. При этом прототип функции будет более ясным.

Функция. Программные объекты, определенные в блоке функции, имеют область действия и время жизни точно такие же, как и в обычном блоке (см. ранее абзац "Блок"). Дополнительно к сказанному отметим следующее. *Параметры функции, передаваемые по значению*, имеют область действия всю функцию и временем жизни — время выполнения функции. После завершения работы функции память, занятая такими параметрами, освобождается. *Параметры функции, передаваемые по ссылке*, имеют область действия и время жизни, определяемые соответствующими аргументами в вызове функции. При этом в область действия, конечно же, входит и блок функции. И, наконец, *метки операторов*, используемые внутри функции, имеют область действия блок функции. Это означает, что в одной функции идентификаторы меток должны различаться, но могут совпадать с идентификаторами меток в других функциях.

Файл. Программный объект, определенный с использованием описателя класса хранения `static` (статический) вне любого блока, функции, класса или пространства имен (о пространстве имен см. далее), имеет *область действия*, начинающуюся в точке определения и заканчивается в конце файла. В область действия включаются вложенные (внутренние) блоки, если в них не содержится переопределение программного объекта с тем же идентификатором. Если же во вложенном блоке переопределен объект с тем же идентификатором, то в этом случае внешний объект во вложенном блоке не видим, но к нему, если он глобален (определен вне блока, функции, класса или пространства имен), можно обратиться с помощью операции доступа к области видимости `::`. *Время жизни* такого программного объекта максимально и совпадает со временем выполнения программы.

Группа файлов, в пределе включающая все файлы программного проекта (глобальная область действия). Программный объект, определенный только в одном из файлов программного проекта вне любого блока, функции, класса или пространства имен и объявленный в других файлах с использованием описателя класса хранения `extern` (внешний) также вне любого блока, функции, класса или пространства имен, имеет *область действия*, начинающуюся в каждом из таких файлов в точке определения или объявления, и заканчивается в конце файла. В область действия включаются вложенные (внутренние) блоки файлов, если в них не содержится переопределение программного объекта с тем же идентификатором. Часто говорят, что такой программный объект имеет глобальную область действия. *Время жизни* такого программного объекта максимально и совпадает со временем выполнения программы.

Теперь будьте более внимательными — далее излагается новая информация.

Класс. Члены объектов-классов, за исключением статических членов классов, имеют *область действия* класс. Это означает, что они являются видимыми лишь в пределах класса. *Время жизни* членов объекта-класса определяется промежутком времени от момента создания объекта-класса до момента его разрушения. Область действия статических членов класса и время их жизни рассмотрены ранее в *разд. 1.3*.

Пространство имен. Язык C++ позволяет явным образом задать область действия имен как часть глобальной области с помощью оператора `namespace`. В каждой

области действия различают так называемые *пространства имен*. Пространство имен — область, в пределах которой идентификатор должен быть уникальным. В разных пространствах имен идентификаторы могут совпадать, поскольку разрешение ссылок осуществляется по контексту идентификатора в программе, например:

```
struct Node
{
    int      Node;
    int      i;
} Node;
```

В данном случае противоречия нет, поскольку имена типа, переменной и поля структуры относятся к разным пространствам имен.

В языке C++ определено четыре разных пространства имен, в пределах каждого из которых идентификаторы должны быть уникальными.

- ❑ К одному пространству имен относятся идентификаторы переменных (объектов); функций; типов, определенных пользователем (**typedef**) и констант перечислений в пределах одной области действия. Все они, кроме идентификаторов функций, могут быть переопределены во вложенных блоках.
- ❑ Другое пространство имен образуют имена типов перечислений, структур, классов и объединений. В этом пространстве имен каждый такой идентификатор должен быть уникальным в пределах одной области действия.
- ❑ Отдельное пространство имен составляют члены каждого класса. Имя члена класса должно быть уникально внутри класса, но может совпадать с именами членов других классов.
- ❑ Метки образуют отдельное пространство имен.

4.2. Пространство имен

Пространство имен (именованная область) служит для логического группирования определений, объявлений и ограничения доступа к ним. Чем больше размер программы, тем более актуально использование именованных областей. Типичным примером, когда целесообразно использовать именованные области, является разработка сложного, большого программного продукта коллективом программистов. С помощью различных пространств имен можно отделить код, написанный одним программистом, от кода, написанного другим. Без использования же именованных областей формировать программу из отдельных частей очень сложно из-за возможного совпадения и конфликта имен. Таким образом, применение пространств имен препятствует доступу к ненужным средствам.

Объявление пространства имен (именованной области) имеет следующий формат:

```
namespace [имя_области] { /*Определения и объявления*/ ... }
```

Одно и то же пространство имен может объявляться неоднократно, причем последующие объявления рассматриваются как расширения предыдущих. Таким образом, пространство имен может объявляться и изменяться за рамками одного файла (листинг 4.1).

Листинг 4.1. Объявление пространства имен demo

```

namespace demo
{
    int      i = 1;      // Определение объекта
    int      k = 0;      // Определение объекта
    void func1( int );    // Прототип (объявление) функции
    // Определение функции
    void func2( int r )
    {
        /*...*/
    }
}

...
// Расширение пространства имен demo
namespace demo
{
    //int      i = 2;      Неверно - двойное определение
    void func1( double ); // Верно - прототип функции (перегрузка)
    void func2( int );    // Верно - прототип функции
}

```

Если необязательное имя области не задано, компилятор определяет его самостоятельно с помощью уникального идентификатора.

Определение или объявление объекта в пространстве имен без указания имени области *равнозначно* его описанию как глобального объекта с описателем класса хранения **static** (такой объект имеет описатель класса хранения "внешний статический").

Помещать определения и объявления объектов в такую область полезно для того, чтобы сохранить локальность кода. Нельзя получить доступ из именованной области одного файла к неименованной области другого файла.

Как следует из приведенного ранее примера, в объявлении пространства имен могут присутствовать как определения объектов программы, так и их объявления. Логично же помещать в объявление пространства имен *только* объявления, а определять их *позднее* с помощью имени области и оператора доступа к области видимости "::", например (см. также пример, приведенный ранее):

```

void demo :: func1( int n )
{
    /* ... */
}

```

Такой прием обеспечивает разделение интерфейса и реализации. Обратите внимание, что таким образом *нельзя объявить новый элемент* пространства имен.

Объекты программы, определенные внутри пространства имен, являются доступными с момента объявления пространства имен. К ним можно явно обращаться с помощью имени области и оператора доступа к области видимости "::", например (см. также пример, приведенный ранее):

```

demo :: i = 100;
demo :: func2( 10 );

```

Если имя часто используется вне своего пространства, то его можно сделать доступным с помощью оператора `using`:

```
using demo :: i;
```

После этого можно использовать имя `i` без явного указания области.

Если требуется сделать доступными все имена из какой-либо области, используется оператор `using namespace`:

```
using namespace demo;
```

Операторы `using` и `using namespace` можно использовать и внутри объявления именованной области, чтобы сделать в ней доступными объявления и определения из другой области:

```
namespace demol
{
    using demo :: i;
    // ...
}
```

Имена, объявленные или определенные в именованной области явно или с помощью оператора `using`, имеют приоритет по отношению к именам, введенным с помощью оператора `using namespace`. Это имеет важное значение при включении нескольких именованных областей, содержащих совпадающие имена.

В заключение рассмотрим пример, иллюстрирующий использование именованных областей (пространств имен) (листинги 4.2—4.7). Этот пример очень важен для практического освоения материала, и поэтому не пожалейте времени на его внимательное изучение и на эксперименты с ним.

Листинг 4.2. Файл File1.cpp

```
/*
    Пространство имен.
    Проект NameSpace.dsp содержит файлы File1.cpp, File2.cpp и File3.cpp.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Включаемые файлы проекта
#include "namespace1.h"
#include "namespace2.h"

// Определение глобального объекта с инициализацией
int i = 1;

// Определение функции из глобального пространства
void fun1( int i )
{
    cout << "Значение глобального объекта ::i = " << ::i << endl;
    cout << "Значение локального объекта i = " << i << endl;
    cout << "Значение объекта именованной области demo::i = " << demo::i
```

```

        << endl;
    cout << "Значение объекта именованной области demo1::i = "
        << demo1::i << endl;

    return;
}

// Определение объекта из пространства имен demo
int demo::i = 3;

// Определение функции из пространства имен demo
void demo::fun1( int i )
{
    cout << "Значение глобального объекта ::i = " << ::i << endl;
    cout << "Значение локального объекта i = " << i << endl;
    cout << "Значение объекта именованной области demo::i = " << demo::i
        << endl;
    cout << "Значение объекта именованной области demo1::i = "
        << demo1::i << endl;

    return;
}

// Главная функция проекта
int main( void ) // Возвращает 0 при успехе
{
    // Определение локального объекта с инициализацией и с тем же именем
    int i = 2;

    cout << "Вывод из функции main" << endl;
    cout << "Значение глобального объекта ::i = " << ::i << endl;
    cout << "Значение локального объекта i = " << i << endl;
    cout << "Значение объекта именованной области demo::i = " << demo::i
        << endl;
    cout << "Значение объекта именованной области demo1::i = "
        << demo1::i << endl;
    // cout << demo::i1 << endl;
    // Недоступно, так как данный объект в этом файле не определен и не
    // объявлен

    cout << endl << "Вывод из функции fun1( int ) из глобальной области"
        << endl;
    fun1( i );

    cout << endl << "Вывод из функции demo::fun1( int )" << endl;

```

```

demo::fun1( i );

cout << endl << "Вывод из функции demo::fun2( int )" << endl;
demo::fun2( i );

cout << endl << "Вывод из функции demo::fun3( )" << endl;
demo::fun3( );

return 0;
}

```

Листинг 4.3. Файл Namespace1.h

```

/*
    Используется в программном проекте Namespace.dsp
*/

// Предотвращение многократного включения данного файла
#ifndef __NAMESPACE1_H
#define __NAMESPACE1_H

// Для потокового ввода-вывода
#include <iostream.h>

// Объявление глобального объекта
extern int i;

// Прототип функции глобальной области
void fun1( int );

// Объявление пространства имен
namespace demo
{
    // Объявление объекта
    extern int
        i;

    // Прототипы функции
    void fun1( int );
    void fun2( int );
    void fun3( void );
}

#endif

```

Листинг 4.4. Файл Namespace2.h

```
/*
    Используется в программном проекте Namespace.dsp
*/

// Предотвращение многократного включения данного файла
#ifndef __NAMESPACE2_H
#define __NAMESPACE2_H

    // Объявление пространства имен
    namespace demo1
    {
        // Объявление объекта
        extern int
            i;
    }

#endif
```

Листинг 4.5. Файл File2.cpp

```
/*
    Используется в программном проекте Namespace.dsp
*/

#include "namespace1.h" // Включаемый файл проекта

// Расширение пространства имен demo, объявленного в namespace.h
namespace demo
{
    // Объявление объекта
    extern int i1;
}

// Определение того же объекта
int demo::i1 = 5;

// В этом файле доступны все объекты пространства имен demo, в том числе
// объект i1

// Определение функции из пространства имен demo
void demo::fun2( int i )
{
    cout << "Значение глобального объекта ::i = " << ::i << endl;
    cout << "Значение локального объекта i = " << i << endl;
```

```

cout << "Значение объекта именованной области demo::i = " << demo::i
    << endl;
//cout << "Значение объекта именованной области "
//      "demo1::i = " << demo1::i << endl;
// Недопустимая запись - пространство имен demo1 в этом
// файле не объявлено
cout << "Значение объекта именованной области demo::i1 = "
    << demo::i1 << endl;

return;
}

```

Листинг 4.6. Файл File3.cpp

```

/*
Используется в программном проекте Namespace.dsp
*/

// Включаемые файлы проекта
#include "namespace1.h"
#include "namespace2.h"

using namespace demo;    // Теперь имена из demo можно записывать без
                        // префиксов

// Определение объекта пространства имен demo1
int demo1::i = 4;

// В этом файле доступны все объекты пространства имен demo, кроме
// объекта i1, и все объекты пространства имен demo1

// Определение функции из пространства имен demo
void demo :: fun3( void )
{
    cout << "Значение глобального объекта ::i = " << ::i << endl;
    // Обратите внимание, что вместо demo :: i используется просто i
    cout << "Значение объекта именованной области demo::i = " << i
        << endl;
    cout << "Значение объекта именованной области demo1::i = "
        << demo1::i << endl << endl;
    //cout << "Значение объекта именованной области "
    //      "demo::i1 =" << demo::i1 << endl;
    // Объект i1 пространства имен demo в этом файле
    // недоступен

return;
}

```

Листинг 4.7. Результаты выполнения программного проекта

```
Вывод из функции main
Значение глобального объекта ::i = 1
Значение локального объекта i = 2
Значение объекта именованной области demo::i = 3
Значение объекта именованной области demo1::i = 4

Вывод из функции fun1( int ) из глобальной области
Значение глобального объекта ::i = 1
Значение локального объекта i = 2
Значение объекта именованной области demo::i = 3
Значение объекта именованной области demo1::i = 4

Вывод из функции demo::fun1( int )
Значение глобального объекта ::i = 1
Значение локального объекта i = 2
Значение объекта именованной области demo::i = 3
Значение объекта именованной области demo1::i = 4

Вывод из функции demo::fun2( int )
Значение глобального объекта ::i = 1
Значение локального объекта i = 2
Значение объекта именованной области demo::i = 3
Значение объекта именованной области demo::i1 = 5

Вывод из функции demo::fun3( )
Значение глобального объекта ::i = 1
Значение объекта именованной области demo::i = 3
Значение объекта именованной области demo1::i = 4

Press any key to continue
```

Анализ приведенного примера позволяет сделать несколько важных выводов.

- ❑ Пространство имен можно сделать доступным в нескольких файлах (в пределе — во всех файлах) программного проекта. Для этого необходимо и достаточно:
 - в объявление пространства имен поместить *только объявления объектов и функций*;
 - объявление пространства имен включить в заголовочный файл программного проекта;
 - с помощью директивы `#include` подключить заголовочный файл в те файлы программного проекта (в пределе — во все файлы), где будет использоваться пространство имен;
 - в *одном* из файлов программного проекта, безразлично в каком, *однократно* поместить определение каждого из объектов и функций, объявление которых содержится в пространстве имен.

В рассмотренном примере пространство имен `demo` доступно во всех файлах программного проекта, а пространство имен `demo1` доступно только в файлах `File1.cpp` и `File3.cpp`.

- ❑ Пример демонстрирует способы одновременного доступа к локальным объектам, глобальным объектам и объектам разных пространств имен с одинаковыми именами. В примере такими объектами являются объекты с именем `i`.
- ❑ Пространство имен можно расширять. В рассмотренном примере в файле `File2.cpp` использовано расширение пространства имен `demo`.
- ❑ Использование оператора `using namespace` демонстрирует фрагмент программного проекта, содержащийся в файле `File3.cpp`.

4.3. Пространство имен стандартной библиотеки

Для использования средств стандартной библиотеки в программу на языке C++ требуется включить с помощью директивы `#include` соответствующие заголовочные файлы.

Элементы заголовочных файлов стандартной библиотеки могут быть определены в пространстве имен `std` или в глобальном пространстве имен. Заголовочные файлы с расширением `h` всегда используют глобальное пространство имен. Для использования стандартного пространства имен `std` следует подключать одноименные заголовочные файлы с расширениями, отличающимися от `h`. Например, в интегрированной среде программирования Microsoft Visual C++ 6.0 такими файлами являются заголовочные файлы без расширений, а в среде Borland C++ Builder 5 — файлы с расширениями `ss`.

Если в программу на языке C++ включен стандартный заголовочный файл с расширением, отличным от `h`, то обращаться к элементу такого заголовочного файла следует с префиксом `std::`. Если в программном коде таких обращений много, то, как указывалось ранее, целесообразно после заголовочного файла с расширением, отличным от `h`, поместить оператор

```
using namespace std;
```

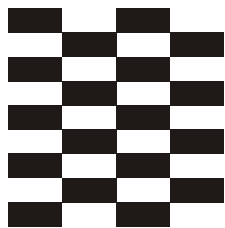
В этом случае можно обращаться к элементу этого заголовочного файла обычным образом — без использования префикса `std::`.

4.4. Вопросы для самопроверки [4]

1. Обязательно ли использовать пространства имен?
2. Каковы отличия между использованием `using namespace` и `using`?
3. Что такое неименованные пространства имен и зачем они нужны?
4. Можно ли использовать идентификаторы, объявленные в пространстве имен без применения служебного слова `using`?
5. Что такое стандартное пространство имен `std`?

Ответы на вопросы можно проверить — они приведены в разд. П1.4 приложения 1.

Глава 5



Ввод-вывод в языке C++ средствами стандартной библиотеки. Потокосые классы

В любой программе, кроме операторов языка, используются средства различных библиотек, включаемых в интегрированную среду разработки (Integrated Development Environment, IDE). Примерами такого рода являются библиотеки классов для проектирования Windows-приложений (библиотека классов MFC — Microsoft Foundation Classes — для IDE Microsoft Visual Studio C++ или библиотека классов OWL — Object Windows Library — для IDE Borland C++). Часть используемых библиотек стандартизована и должна поставляться вместе с любым компилятором языка.

Сказанное в полной мере относится и к языку программирования C++. Стандартную библиотеку языка C++ можно разделить на две части:

- функции, макросы, типы и константы, унаследованные из библиотеки языка C;
- стандартные классы и другие средства языка C++.

Функции, унаследованные из стандартной библиотеки языка C, в соответствии с их назначением, можно разделить на функции ввода-вывода, обработки строк, математические функции, функции для работы с динамической памятью, для поиска, сортировки и т. п. Описание средств, унаследованных из стандартной библиотеки языка C, приведены на компакт-диске в *приложениях 6* (константы, макросы и типы данных) и *7* (функции). Вторая часть стандартной библиотеки языка C++ содержит классы, шаблоны и другие средства для ввода, вывода, хранения и обработки данных как стандартных, так и пользовательских типов.

В соответствии с их назначением стандартные классы языка C++ можно разбить на следующие группы.

- *Потокосые классы* для управления потоками данных между оперативной памятью и внешними устройствами (дисками, клавиатурой, экраном монитора), а также в пределах оперативной памяти.
- *Строковый класс* для удобной и защищенной от ошибок работы с символьными строками.
- *Контейнерные классы, алгоритмы и итераторы*. Контейнерные классы реализуют наиболее распространенные структуры для хранения данных — списки, вектора, множества и др. В состав стандартной библиотеки входят также *алгоритмы*, использующие и преобразующие контейнеры. *Итераторы* обеспечивают унифицированный доступ к элементам контейнерных классов.

- *Математические классы* для эффективной обработки массивов с плавающей точкой и комплексных данных.
- *Диагностические классы* для динамической идентификации типов и объектно-ориентированной обработки ошибок.
- *Остальные классы* — для динамического распределения памяти, адаптации к локальным особенностям и т. п.

Часть стандартной библиотеки, в которую входят контейнерные классы, алгоритмы и итераторы, называют *стандартной библиотекой шаблонов* (Standard Template Library, STL).

Для использования средств стандартной библиотеки в программу с помощью директивы `#include` следует подключить соответствующие стандартные заголовочные файлы. Элементы заголовочных файлов без расширения `h` определены в *стандартном пространстве имен* `std`, а одноименные файлы с расширением `h` — в *глобальном пространстве имен*. Список заголовочных файлов стандартной библиотеки приведен на компакт-диске в *приложении 5*.

И еще одно *важное* замечание. Имена стандартных заголовочных файлов языка C++ для функций языка C, определенные в пространстве имен `std`, начинаются с буквы "с" и не имеют расширения, например, `<cstdio>`, `<cstring>`, `<cstdlib>`. Для каждого заголовочного файла вида `<cx>` (например, `<cstdio>`) существует файл `<x.h>` (например, `<stdio.h>`), определяющий те же имена в глобальном пространстве имен.

Далее в этом разделе рассматриваются средства ввода-вывода языка C++, основанные на использовании потоковых классов стандартной библиотеки языка C++ [5, 9]. Остальные средства стандартной библиотеки рассматриваются далее в третьей части учебного пособия.

Поток — понятие, относящееся к переносу данных от источника к приемнику. Потоки языка C++, в отличие от стандартных функций ввода-вывода в стиле языка C, обеспечивают надежную работу со стандартными и определенными пользователем типами данных, а также единообразный и понятный синтаксис.

Чтение данных из потока называется *извлечением* их из потока, а вывод в поток — *помещением* или *включением* данных в поток. Поток определяется как *последовательность байтов* и не зависит от конкретного устройства, с которым выполняется обмен (оперативная память, файл на диске, экран, клавиатура или принтер). Обмен с потоком для увеличения скорости передачи данных производится, как правило, через специально организованную область оперативной памяти, называемую *буфером*. Фактическая передача данных выполняется при выводе после заполнения буфера, а при вводе — если буфер исчерпан.

Дадим классификацию потоков. *По направлению обмена* потоки можно разделить на *входные* (данные вводятся в оперативную память), *выходные* (данные выводятся из оперативной памяти) и *двунаправленные* (допускается как извлечение, так и включение). *По виду устройств*, с которыми работает поток, потоки можно разделить на *стандартные*, *файловые* и *строковые*.

Стандартные потоки предназначены для передачи данных от клавиатуры и на экран дисплея, *файловые потоки* — для обмена информацией с файлами на магнитном диске, а *строковые потоки* — для работы с массивами символов в оперативной памяти.

Для поддержки потоков библиотека языка C++ содержит иерархию классов, построенную на основе двух базовых классов `ios` и `streambuf`. Класс `ios` содержит общие для ввода-вывода поля и методы, класс `streambuf` обеспечивает буферизацию потоков и их взаимодействие с физическими устройствами. От этих классов наследуется класс `istream` для входных потоков и класс `ostream` — для выходных. Два последних класса являются базовыми для класса `iostream`, реализующего двунаправленные потоки. Далее в иерархии классов располагаются файловые и строковые потоки. К наиболее часто используемым потоковым классам относятся следующие классы:

- `ios` — базовый класс потоков;
- `istream` — класс входных потоков;
- `ostream` — класс выходных потоков;
- `iostream` — класс двунаправленных потоков;
- `istringstream` — класс входных строковых потоков;
- `ostringstream` — класс выходных строковых потоков;
- `stringstream` — класс двунаправленных строковых потоков;
- `ifstream` — класс входных файловых потоков;
- `ofstream` — класс выходных файловых потоков;
- `fstream` — класс двунаправленных файловых потоков.

Описания этих классов находятся в следующих заголовочных файлах:

- `<ios>` — базовый класс потоков ввода-вывода;
- `<istream>` — шаблон потока ввода;
- `<ostream>` — шаблон потока вывода;
- `<iostream>` — стандартные объекты и операции с потоками ввода-вывода;
- `<fstream>` — потоки ввода-вывода в файлы;
- `<sstream>` — потоки ввода-вывода в строки;
- `<streambuf>` — буферизация потоков ввода-вывода;
- `<iomanip>` — манипуляторы.

Подключение к программе файлов `<fstream>` и `<sstream>` автоматически подключает и файл `<iostream>`, так как он является для них базовым.

Повторно напоминаем, что основным преимуществом потоков по сравнению со стандартными функциями ввода/вывода, унаследованными из библиотеки языка C, являются *контроль типов*, а также *расширяемость*, то есть возможность работать с типами, определенными пользователем. Для этого достаточно переопределить операции потоков. Кроме того, потоки могут работать с расширенным набором символов `wchar_t`.

В заключение отметим, что язык C++ не определяет ввод-вывод, но предоставляет широкий набор средств для реализации ввода-вывода. Ввод-вывод в языке C++ — это широкое использование перегрузки. Перегрузка операций `<<` и `>>` в классе `iostream`, определенном во включаемом файле `iostream.h`, дает богатые возможности

ввода и вывода данных. Можно вводить-выводить данные как встроенных (предопределенных) типов, так и типов, определенных пользователем, не применяя функции `scanf()`-`fscanf()`, `printf()`-`fprintf()`-`sprintf()` с их сложными форматами. Хотя в языке C++ можно использовать как новые средства ввода-вывода, так и старые средства из языка C, все же предпочтительнее применять новые средства ввода-вывода из класса `iostream`. Они проще и элегантнее.

5.1. Ввод-вывод встроенных (стандартных) типов

Для организации ввода-вывода в программах на языке C++ используются включаемые файлы `iostream` или `iostream.h`, которые содержат определения нескольких классов и объектов, обеспечивающих методы ввода-вывода. В частности, язык C++ поддерживает четыре предопределенных объекта, выполняющих ввод-вывод.

- ❑ Объект `cin` — стандартный ввод, соответствующий `stdin` в языке C. Этот объект связывается с клавиатурой (стандартным буферизированным вводом) и является объектом класса `istream`.
- ❑ Объект `cout` — стандартный вывод, соответствующий `stdout` в языке C. Объект `cout` связывается с экраном (стандартным буферизированным выводом) и является объектом класса `ostream`.
- ❑ Объект `cerr` — стандартный вывод ошибок, соответствующий `stderr` в языке C (экран). Этот объект связывается с экраном (стандартным *не буферизированным* выводом), куда направляются сообщения об ошибках. Объект `cerr` является объектом класса `ostream`.
- ❑ Объект `clog` не имеет эквивалента в языке C и связывается с экраном (стандартным *буферизированным* выводом), куда направляются сообщения об ошибках. Объект `clog` также является объектом класса `ostream`.

Эти стандартные объекты создаются при включении в программу заголовочного файла `<iostream>`, при этом становятся доступными связанные с ними средства ввода-вывода. Имена стандартных объектов можно переназначать на другие устройства или в файлы. Рассмотрим некоторые из них подробнее.

Предопределенный объект `cout` является объектом класса `ostream`, определение которого содержится во включаемом файле `ostream.h`. Класс `ostream` используется для управления форматированным и неформатированным выводом. Здесь переопределяется операция сдвига "<<" для вывода данных встроенных типов.

Например, оператор

```
cout << x;
```

посылает значение переменной `x` в объект `cout` класса `ostream` для вывода. Перегруженный метод **operator<<** возвращает ссылку на объект `ostream`, для которого он был вызван. Поэтому к результату операции вывода можно еще раз применить операцию вывода, т. е. несколько операций вывода можно объединить в цепочку:

```
#include <iostream.h>
int          x;
cout << "x = " << x << "\n";
```

Учитывая, что операция "<<" имеет порядок вычислений слева направо, такая запись будет интерпретироваться как

```
((cout.operator<<("x = ")).operator<<(x)).operator<<("\n");
```

В зависимости от типа переданного аргумента будет вызываться тот или иной экземпляр метода `operator<<`.

Аналогично выводу в языке C++ определяется и ввод. С вводом связан класс `istream`, в котором определена перегруженная операция ">>" для стандартных типов. Объект `cin` класса `istream` используется для ввода с назначенного устройства. Например,

```
#include <iostream.h>

int          x;
cin >> x;
```

Метод `operator>>` также возвращает ссылку на объект класса `istream`, и, значит, операции ввода можно тоже связывать в цепочку. Например,

```
#include <iostream.h>

int          x;
long         d;
float        c;
cin >> x >> d >> c;
```

Это будет проинтерпретировано следующим образом:

```
(( cin.operator>>( x ) ).operator>>( d ) ).operator>>( c );
```

ЗАМЕЧАНИЕ

Вводимые значения должны соответствовать объявленному типу и должны разделяться не менее чем одним пробелом, табулятором или символом новой линии.

Как и для других перегруженных операций, для вставки (вывода) и извлечения (ввода) нельзя изменять приоритеты. Поэтому в необходимых случаях следует использовать круглые скобки:

```
// Круглые скобки здесь не требуются, так как приоритет
// сложения выше, чем приоритет операции <<
int          i, j;
cout << i+j;

// А здесь круглые скобки необходимы - приоритет операции
// отношения меньше, чем приоритет операции <<
cout << ( i<j );
```

5.2. Состояния предопределенных объектов (поток). Ошибки потоков

Любой предопределенный поток-объект в каждый момент времени находится в определенном состоянии. С учетом этого состояния и осуществляется обработка ошибок ввода-вывода. Проще всего состояние потока можно проверить как обычное

логическое выражение. В этом проявляется красота и элегантность языка C++. Например,

```
#include <stdlib.h>
#include <iostream.h>
// ...
float          f;
cin >> f;
if( !cin )
{
    cout << "\n Ошибка 1. Ошибка ввода " << endl;
    // О манипуляторе endl см. ниже
    exit( 1 );
}
```

Аналогичным образом можно проверить и ошибку вывода для потока cout, хотя это обычно не делается.

ЗАМЕЧАНИЕ

Состояние потока cin "конец файла (потока)" при вводе не трактуется как ошибка и поэтому не может быть проверено таким способом. Для отслеживания состояния конца файла (потока) можно применить метод eof(), который возвращает ненулевое значение, если конец файла достигнут:

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

float          f;
cin >> f;
if( cin.eof( ) )
    cout << "\n Конец файла ввода " << endl;
if( !cin )
{
    cout << "\n Ошибка ввода " << endl;
    exit( 1 );
}
```

В базовом классе ios определено битовое поле, элементы которого представляют возможное состояние потока:

```
static const iostate goodbit;    // Нет ошибок
static const iostate eofbit;     // Достигнут конец файла
static const iostate failbit;    // Ошибка форматирования или
                                // преобразования
static const iostate badbit;     // Серьезная ошибка, после
                                // которой пользоваться потоком
                                // невозможно
```

Состоянием потока можно управлять с помощью следующих методов и операций класса ios:

```
// Возвращает текущее состояние потока (поразрядное объединение всех
// установленных флагов состояния): iostate - тип поля флагов
```

```

//    состояния
iostate rdstate( ) const;
// Возвращает ненулевое значение, если сброшены все флаги (биты)
//    ошибок
bool good( ) const;
// Возвращает ненулевое значение, если установлен флаг eofbit
bool eof( ) const;
// Возвращает ненулевое значение, если установлен один из флагов
//    (битов) failbit или badbit
bool fail( ) const;
// Возвращает ненулевое значение, если установлен флаг badbit
bool bad( ) const;
// Сбрасывает все флаги и устанавливает флаги, содержащиеся в
//    iostate. При отсутствии аргумента все флаги (биты) состояния
//    потока сбрасываются. Обратите внимание на этот метод - его
//    часто используют с пустым списком аргументов для очистки флагов
//    (битов) состояния, чтобы можно было продолжить ввод-вывод
void clear( iostate state = goodbit );
// Только устанавливает флаги, содержащиеся в state. Остальные
//    флаги не изменяются
void setstate( iostate state );
// Возвращает нулевой указатель, если установлен хотя бы один бит
//    ошибки
operator void * ( ) const;
// Возвращает true, если установлен хотя бы один бит ошибки
bool operator! ( ) const;

```

Приведем несколько примеров работы с флагами состояния потока.

```

// Проверить, установлен ли флаг flag (в качестве flag можно
//    использовать eofbit, failbit или badbit)
if( stream_obj.rdstate( ) & ios::flag )
{
    // Установлен
}
// Сбросить флаг flag
stream_obj.clear( stream_obj.rdstate( ) & ~ios::flag );
// Установить флаг flag
stream_obj.setstate( ios::flag );
// Сбросить все флаги
stream_obj.clear( );

```

Операция `void *` неявно вызывается всякий раз, когда поток сравнивается с нулем:

```

while( stream_obj )
{
    // Можно продолжать ввод-вывод
}

```

```

if( !stream_obj )
{
    // Ошибка ввода-вывода
}

```

5.3. Ввод-вывод типов, определенных пользователем

Еще раз напомним, что для ввода и вывода в потоках используются перегруженные для всех стандартных типов операции ">>" и "<<". Операции ввода ">>" перегружены в классе `istream`, определение которого содержится в заголовочном файле `istream`. Аналогично, операции вывода "<<" перегружены в классе `ostream`, определение которого содержится в заголовочном файле `ostream`. Чтобы воспользоваться операциями ввода-вывода, достаточно включить в программу заголовочный файл `iostream`, который, в свою очередь, подключает заголовочные файлы `istream` и `ostream`. При этом выбор конкретной операции ввода или вывода определяется типом вводимого или выводимого данного.

Но язык C++ позволяет определять новые типы, столь же эффективные и удобные, как и встроенные типы. Для того чтобы аналогичным образом вводить или выводить значения объектов с типами, определенными пользователем, требуется перегрузить операции ">>" и "<<" для каждого пользовательского типа. Операции ">>" и "<<" бинарные, у них левым операндом является объект-поток, а правым — объект, который требуется извлечь или поместить в этот поток. Значение, возвращаемое методом, перегружающим операции ">>" и "<<", должно быть ссылкой на поток, чтобы можно было организовать цепочки операций, как и в случае стандартных типов.

Детали перегрузки операций ввода-вывода для пользовательских типов иллюстрирует листинг 5.1.

Листинг 5.1. Файл CMP.CPP

```

/*
    Работа с комплексными данными с использованием класса CMP, применяется статиче-
    ское размещение комплексного данного и ввод-вывод на основе классов с перегрузкой
    операторов << и >>.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET)
*/

#include <iostream>           // Для потокового ввода-вывода с использованием
                             // классов

using namespace std;         // Используем стандартное
                             // пространство имен

//*****
// Для работы с комплексными данными - объявление класса
class CMP

```

```

{
    // Данные

private:

    int      r,          // Действительная часть
           i;          // Мнимая часть

    // Методы

public:

    // Конструктор: подставляемая функция
    CMP(
        int    re,        // Действительная часть
        int    im )      // Мнимая часть
    {
        r = re; i = im;
    }

    // Перегрузка операции "<<" для вывода с применением дружественной
    // функции. Перегружающая функция обязательно должна быть
    // дружественной классу CMP, а не членом класса, потому что
    // операнды операции разных классов
    friend ostream & operator<<( ostream &, const CMP & );

    // Перегрузка операции ">>" для ввода с применением дружественной
    // функции. Перегружающая функция обязательно должна быть
    // дружественной классу CMP, а не членом класса, потому что
    // операнды операции разных классов
    friend istream & operator>>( istream &, CMP & );

};                                     // Конец объявления класса

//*****
// Реализация методов класса

//*****
// Перегрузка операции "<<" для вывода с применением дружественной
// функции
ostream & operator<<(          // Возвращает ссылку (можно применять цепочки
                           // операций)
    ostream    &o,           // Объект вывода
    const CMP  &c )          // Выводимое значение
{
    return o << c.r << " i " << c.i << endl;
}

```

```

}

//*****
// Перегрузка операции ">>" для ввода с применением дружественной функции
// класса
istream & operator>>(    // Возвращает ссылку (можно применять цепочки
                        // операций)
    istream    &o,      // Объект ввода
    CMP        &c )    // Вводимое значение
{
    o >> c.r >> c.i;
    if( !o )
    {
        cout << "\n Input error " << endl;
        exit( 1 );
    }

    return o;
}

//*****
// Тестирование
int main( void )        // Возвращает 0 при успехе
{
    // Определение комплексного объекта: вызывается конструктор
    CMP        c( 12, 21 );

    // Печать значения: используется перегруженная операция <<
    cout << " Value of the object c: " << c;
    cout << " Enter complex value in the format int int: ";

    // Ввод нового значения и его печать
    cin >> c;
    cout << " Value of the object c: " << c;

    return 0;
}

```

ЗАМЕЧАНИЕ

Еще раз отмечаем две важные особенности. Мы можем сцепить нашу новую пользовательскую операцию "<<" или операцию ">>" в цепочку с уже существующими операциями вывода или ввода, потому что обе функции-операции возвращают ссылку на объект класса `ostream` или `istream`. Перегружающий метод для операции "<<" или ">>" обязательно должен быть дружественным классу `CMP`, а не членом класса, потому что операнды операции различных классов.

Как следует из приведенного примера, ввод пользовательского типа определяется точно так же, как и вывод. Единственное отличие состоит в том, что для операции ввода важно, чтобы в перегружающем методе *второй параметр был ссылкой*. Кроме того, в перегружающем методе нужно выполнить контроль успешности операции ввода.

5.4. Форматированный ввод-вывод. Манипуляторы

В потоковых классах форматирование можно выполнять тремя способами — с помощью *манипуляторов* (удобнее всего), *флагов форматирования* и *форматирующих методов*.

Потоковые классы содержат средства, позволяющие ввести или вывести данные в той или иной системе счисления с различной точностью. Выводимые данные можно прижимать к левому или правому концу поля вывода. Средства форматированного ввода-вывода позволяют управлять и другими деталями формата ввода-вывода. Одним из наиболее удобных способов реализации указанных возможностей является использование *манипуляторов*.

Манипуляторы и флаги форматирования. Манипуляторы используют перегружаемые методы, что упрощает кодирование форматированного ввода-вывода. Манипуляторы можно вставлять в цепочки ввода-вывода с помощью операций ">>" и "<<". Манипуляторы бывают *параметризованными* (записываются в виде вызова функции с одним аргументом) и *простыми* (после имени манипулятора круглые скобки с аргументом отсутствуют). Для использования параметризованных манипуляторов в программу необходимо включить соответствующий заголовочный файл:

```
#include <iomanip>
```

Например, для переустановки системы счисления можно использовать следующие методы-манипуляторы:

- ☐ `dec` — использовать десятичную систему счисления;
- ☐ `hex` — использовать шестнадцатеричную систему счисления;
- ☐ `oct` — использовать восьмеричную систему счисления.

Изменение системы счисления действует до следующего явного изменения.

```
int a=256;  
cout << a << hex << a << oct << a << endl;
```

В этом случае значение переменной `a` будет последовательно выдаваться вначале в десятичной системе счисления (по умолчанию), потом в шестнадцатеричной и, в заключение, в восьмеричной системах счисления. Здесь `dec`, `hex` и `oct` представляют собой методы-манипуляторы без параметров и для их использования включать в программу файл `iomanip` не нужно.

Методы-манипуляторы и их действие приведены далее в табл. 5.1. В этой таблице сначала перечислены простые манипуляторы, а затем — манипуляторы с параметрами. В параметризованных манипуляторах `resetiosflags(long)` и `setiosflags(long)` можно использовать биты формата (флаги форматирования), которые определены в классе `ios_base`. Перечень флагов форматирования и их действие приведены в листинге 5.2.

Обратите внимание на две важных особенности флагов форматирования.

В записи аргументов манипуляторов `resetiosflags(long)` и `setiosflags(long)` биты форматов (флаги форматирования) можно комбинировать, используя побитовую операцию ИЛИ "|", например:

```
ios :: left | ios :: fixed
```

ЗАМЕЧАНИЕ

Флаги (`left`, `right` и `internal`), (`dec`, `oct` и `hex`), а также (`scientific` и `fixed`) взаимно исключают друг друга. Это означает, что в каждый момент может быть установлен только один флаг из каждой группы.

Таблица 5.1. Методы-манипуляторы и их действие

Имя	Действие	Примечание
<code>boolalpha</code>	Устанавливает текстовый формат ввода и вывода логических значений (<code>true</code> или <code>false</code>)	Используется для ввода-вывода
<code>noboolalpha</code>	Устанавливает числовой формат ввода и вывода логических значений (1 или 0). Используется по умолчанию	Используется для ввода-вывода
<code>showbase</code>	Устанавливает вывод восьмеричных значений с префиксом 0, а шестнадцатеричных — с префиксом 0x или 0X	Используется для вывода
<code>noshowbase</code>	Устанавливает вывод восьмеричных значений без префикса 0, а шестнадцатеричных — без префикса 0x или 0X	Используется для вывода
<code>showpoint</code>	Устанавливает отображение десятичной точки и дробной части при выводе вещественных значений	Используется для вывода
<code>noshowpoint</code>	Отменяет отображение десятичной точки и дробной части при выводе вещественных значений	Используется для вывода
<code>showpos</code>	Устанавливает вывод знака для положительного числа	Используется для вывода
<code>noshowpos</code>	Отменяет вывод знака для положительного числа. Используется по умолчанию	Используется для вывода
<code>skipws</code>	Устанавливает автоматическое игнорирование начальных пробелов при чтении данных оператором ">>". Используется по умолчанию	Используется только для ввода
<code>noskipws</code>	Отменяет автоматическое игнорирование начальных пробелов при чтении данных оператором ">>". Используется по умолчанию	Используется только для ввода
<code>unitbuf</code>	Устанавливает вывод содержимого буфера после каждой операции записи	Используется для вывода
<code>nounitbuf</code>	Отменяет вывод содержимого буфера после каждой операции записи. Используется по умолчанию	Используется для вывода
<code>uppercase</code>	Устанавливает вывод символов в числах в верхнем регистре	Используется для вывода

Таблица 5.1 (окончание)

Имя	Действие	Примечание
<code>nouppercase</code>	Отменяет вывод символов в числах в верхнем регистре. Используется по умолчанию	Используется для вывода
<code>left</code>	Устанавливает выравнивание по левому краю	Используется для вывода
<code>rightt</code>	Устанавливает выравнивание по правому краю. Используется по умолчанию	Используется для вывода
<code>internal</code>	Устанавливает выравнивание знака по левому краю, а значения — по правому	Используется для вывода
<code>dec</code>	Устанавливает десятичную систему счисления (используется по умолчанию)	Используется для ввода-вывода
<code>hex</code>	Устанавливает шестнадцатеричную систему счисления	Используется для ввода-вывода
<code>oct</code>	Устанавливает восьмеричную систему счисления	Используется для ввода-вывода
<code>fixed</code>	Печатать вещественные числа в виде 15.00	Используется только для вывода
<code>scientific</code>	Печатать вещественные числа в виде 1.5E+000	Используется только для вывода
<code>endl</code>	Вставляет символ новой строки и выгружает поток	Используется только для вывода
<code>flush</code>	Выгружает поток <code>ostream</code>	Используется только для вывода
<code>ends</code>	Вставляет завершающий нулевой символ строки	Используется только для вывода
<code>setbase(int)</code>	Задаёт основание системы счисления (8, 10, 16 или 0). 0 является основанием по умолчанию (десятичная система, кроме случаев, когда вводятся 8- или 16-ричные числа)	Используется для ввода-вывода
<code>resetiosflags(long)</code>	Сбрасывает флаги состояния потока, биты которых в аргументе равны единице	Используется для ввода-вывода
<code>setiosflags(long)</code>	Устанавливает флаги состояния потока, биты которых в аргументе равны единице	Используется для ввода-вывода
<code>setfill(int)</code>	При избыточной ширине поля вывода устанавливает символ заполнитель с кодом, равным значению аргумента (по умолчанию таким символом является пробел)	Используется для вывода
<code>setprecision(int)</code>	Устанавливает точность для типа с плавающей точкой (по умолчанию точность 6)	Используется для вывода
<code>setw(int)</code>	Устанавливает ширину поля	Используется для ввода-вывода

Листинг 5.2. Флаги форматирования

```
// При установленном флаге действует текстовое представление булевских
// значений, иначе – числовое (1 и 0). Умолчание – числовое
// представление. fmtflags – тип битового поля флагов форматирования
static const fmtflags          boolalpha;

// Автоматически игнорировать начальные пробелы при вводе данных
// оператором >> (используется по умолчанию)
static const fmtflags          skipws;

// Выравнивание при выводе к левой границе поля
static const fmtflags          left;

// Выравнивание при выводе к правой границе поля (используется по
// умолчанию)
static const fmtflags          right;

// Вывод знака числа по левому краю, а числа – по правому. Промежуток
// заполняется символом заполнения (по умолчанию – пробелами)
static const fmtflags          internal;

// Чтение/запись в десятичной системе счисления (используется по
// умолчанию)
static const fmtflags          dec;

// Чтение/запись в восьмеричной системе счисления
static const fmtflags          oct;

// Чтение/запись в шестнадцатеричной системе счисления
static const fmtflags          hex;

// Для шестнадцатеричных чисел выводить префикс 0x, а восьмеричных –
// префикс 0
static const fmtflags          showbase;

// При выводе вещественных чисел отображать десятичную точку и дробную
// часть
static const fmtflags          showpoint;

// Печатать шестнадцатеричными прописными цифрами
static const fmtflags          uppercase;

// Печатать "+" перед положительным числом
static const fmtflags          showpos;
```

```
// Печатать вещественные числа в виде 1.50E+001
static const fmtflags      scientific;

// Печатать вещественные числа в виде 15.00
static const fmtflags      fixed;

// Выгружать буфер после каждого вывода
static const fmtflags      unitbuf;
```

Проиллюстрируем использование манипуляторов тремя простыми примерами (листинги 5.3—5.7).

Листинг 5.3. Файл MANIP1.CPP

```
/*
    Форматирование с использованием манипуляторов без параметров.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET
*/

#include <iostream>          // Ввод-вывод C++

using namespace            // Используем стандартное
    std;                   // пространство имен

// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    int        i;

    // Запросить ввод
    cout << " Please, enter an integer: ";

    // Извлечь число и проверить корректность ввода числа
    cin >> i;
    if( !cin )
    {
        // Информировать об ошибке
        cout << " Erratic input ..." << endl;
    }
    else
    {
        // Применить манипуляторы для вывода
        cout << "Hex: " << hex << i << endl << "Oct: " << oct << i
            << endl << "Dec: " << dec << i << endl;
```

```

    }

    return 0;
}

```

Листинг 5.4. Файл MANIP2.CPP

```

/*
    Форматированный вывод с использованием манипуляторов с параметром. Переназначение
    экранного вывода в файл на диске.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET
*/

#include <iostream>           // Ввод-вывод C++
#include <iomanip>            // Для параметризованных манипуляторов
#include <fstream>            // Файловый ввод-вывод C++

using namespace             // Используем стандартное
    std;                   // пространство имен

// Тестирование
int main( )                 // Возвращает 0 при успехе
{
    double    dbls[ ] = { 1.245, -12.99133, 134.007804,
                          -2.345, 0.000003, 0.0 };

    bool      b = true;

    // Перенаправление потока cout в файл scr - такой прием удобен при
    // выводе текста на русском языке. В отличие от экранного вывода
    // текст в файле scr будет читаемым
    ofstream  FileScr( "scr" );
    cout.rdbuf( FileScr.rdbuf( ) );

    cout << "Использование манипуляторов и флагов форматирования\n\n"
         << "Манипуляторы boolalpha и noboolalpha" << endl;
    cout << "Умолчение: b = " << b << boolalpha << ", boolalpha: b = "
         << b << noboolalpha << ", noboolalpha: b = " << b << endl
         << endl;

    cout << "Манипуляторы showpoint-showpos-setprecision(1)"
         << "\n-setiosflags( ios::internal )-setw( 20 )" << endl;
    cout << showpoint << showpos << setprecision( 1 )
         << setiosflags( ios::internal );
    for( int i = 0; i < sizeof( dbls )/sizeof( dbls[ 0 ] ); i++ )
    {
        cout << setw( 20 ) << dbls[ i ] << endl;
    }
}

```

```

}

cout << endl << "Манипуляторы noshowpoint-noshowpos-"
    "setprecision( 1 )\n-resetiosflags( ios::internal )-"
    "setiosflags( ios::right )\n-setw( 20 )" << endl;
cout << noshowpoint << noshowpos << setprecision( 1 ) <<
    resetiosflags( ios::internal ) << setiosflags( ios::right );
for( i = 0; i < sizeof( dbls )/sizeof( dbls[ 0 ] ); i++ )
{
    cout << setw( 20 ) << dbls[ i ] << endl;
}

cout << "\nМанипуляторы setfill( '.' )-setprecision( 4 )"
    "\n-setiosflags( ios::fixed )-setw( 20 )" << endl;
cout << setfill( '.' ) << setprecision( 4 )
    << setiosflags( ios::fixed );
for( i = 0; i < sizeof( dbls )/sizeof( dbls[ 0 ] ); i++ )
{
    cout << setw( 20 ) << dbls[ i ] << endl;
}
cout << endl;

return 0;
}

```

Листинг 5.5. Результаты выполнения программы

Использование манипуляторов и флагов форматирования

Манипуляторы boolalpha и noboolalpha

Умолчание: b = 1, boolalpha: b = true, noboolalpha: b = 1

Манипуляторы showpoint-showpos-setprecision(1)

```

-resetiosflags( ios::internal )-setw( 20 )
+           1.
-           1.e+001
+           1.e+002
-           2.
+           3.e-006
+           0.0

```

Манипуляторы noshowpoint-noshowpos-setprecision(1)

```

-resetiosflags( ios::internal )-setiosflags( ios::right )
-setw( 20 )
           1
        -1e+001

```

```

1e+002
-2
3e-006
0

```

```

Манипуляторы setfill( '.' )-setprecision( 4 )
-setiosflags( ios::fixed )-setw( 20 )
.....1.2450
.....-12.9913
.....134.0078
.....-2.3450
.....0.0000
.....0.0000

```

Листинг 5.6. Файл out.cpp

```

/*
    Параметризованные манипуляторы и флаги форматирования при выводе (это наиболее
    актуально и востребовано практикой).
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 7.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <iomanip>            // Манипуляторы с параметрами

using namespace            // Используем стандартное
    std;                   // пространство имен

// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    double    d = 1.4785234;
    double    d1 = 1.4785234e+10;

    // Вывод по умолчанию - ширина поля вывода минимальна, точность для
    // данных с плавающей точкой 6 десятичных цифр
    cout << "d=" << d << endl;
    cout << "d1=" << d1 << endl;

    // Ширина поля вывода, размещение в поле вывода, показ положительного
    // знака, задание символа-заполнителя, вывод в форме с порядком или
    // с фиксированной точкой. !!! Параметризованный манипулятор,
    // задающий ширину поля вывода размещается непосредственно перед
    // выводимым данным и действует только на него

    // При выводе показывать знак положительного числа; ширина поля
    // вывода 15 позиций; знак печатаемого данного прижать к левому

```

```

//   краю поля вывода, а значение - к правому краю; точность вывода
//   составляет две десятичные цифры; формат вывода по умолчанию -
//   с фиксированной точкой; избыточные позиции в поле вывода по
//   умолчанию заполняются пробелами
cout << setiosflags( ios::showpos | ios::internal )
    << setprecision( 2 ) << "d=" << setw( 15 ) << d << endl;

// Параметры вывода прежние, но формат вывода - с порядком
cout << setiosflags( ios::scientific ) << "d=" << setw( 15 ) << d
    << endl;

// Параметры вывода как в предыдущем выводе, но в поле вывода данное
//   прижимается к левой границе, а в качестве символа-заполнителя
//   используется точка
cout << resetiosflags( ios::internal ) << setiosflags( ios::left )
    << setfill( '.' ) << "d1=" << setw( 15 ) << d1 << endl << endl;

return 0;
}

```

Листинг 5.7. Результаты выполнения программы

```

d=1.47852
d1=1.47852e+010
d+=      1.5
d+=      1.48e+000
d1+=1.48e+010.....

```

Форматирующие методы. Для управления флагами форматирования в классе `ios` (точнее, в классе `ios_base`) имеются методы `flags()`, `setf()` и `unsetf()`:

```

// Возвращает текущие флаги форматирования: fmtflags - тип битового
//   поля флагов форматирования
fmtflags flags( ) const;

// Возвращает текущие флаги форматирования и устанавливает новые
//   флаги в соответствии с битами параметра
fmtflags flags( fmtflags );

// Возвращает текущие флаги форматирования. Присваивает флагам
//   форматирования, биты которых установлены в первом параметре,
//   значения соответствующих битов второго параметра
fmtflags setf( fmtflags, fmtflags );

// Возвращает текущие флаги форматирования. Устанавливает флаги,
//   биты которых установлены в параметре
fmtflags setf( fmtflags );

// Сбрасывает флаги, переданные в параметре
void unsetf( fmtflags );

```

Для управления минимальной шириной поля вывода, точностью вывода вещественных значений и символом заполнения избыточных позиций поля вывода используются следующие методы класса `ios`:

```
// Возвращает текущее значение ширины поля вывода
streamsize width( ) const;

// Возвращает текущее значение ширины поля вывода и устанавливает
// новую ширину поля вывода в соответствии со значением параметра
streamsize width( streamsize );

// Возвращает текущий символ заполнения поля вывода
char_type fill( ) const;

// Возвращает текущий символ заполнения поля вывода и устанавливает
// новый символ заполнения в соответствии со значением параметра
char_type fill( char_type );

// Устанавливает значение точности вывода вещественных чисел и
// возвращает старое значение точности
streamsize precision( streamsize );

// Возвращает значение точности вывода вещественных чисел
streamsize precision( ) const;
```

ЗАМЕЧАНИЕ

Перед установкой некоторых флагов форматирования требуется сбросить флаги, которые не могут быть установлены вместе с ними.

5.5. Методы обмена с потоками

В потоковых классах, наряду с операциями извлечения из потока ">>" и включения в поток "<<", определены методы для неформатированного чтения и записи в поток *без преобразования передаваемых данных*.

Рассмотрим наиболее употребительные *методы чтения* из потока, определенные в классе `istream` (включаемый файл `<istream>`):

```
// Возвращает код символа, извлеченного из потока. Метод обращается
// с символами-разделителями (символы пробельной группы - ' ',
// '\t', '\n') точно так же, как с другими символами
int get( );

// Возвращает ссылку на поток, из которого выполнялось чтение, и
// записывает извлеченный символ в аргумент
istream & get( char & );

// Считывает из потока num-1 символов (или пока не встретится символ
// lim) и копирует их в buf. Вместо символа lim в buf записывается
// символ конца строки '\0'. Символ lim остается в потоке.
// Возвращает ссылку на текущий поток
```

```

istream & get( char *buf, streamsize num, char lim );
// Считывает из потока num-1 символов и копирует их в buf. Если
// встречается символ новой строки или конец файла, то чтение
// завершается досрочно. Затем в buf записывается символ конца
// строки '\0'. Возвращает ссылку на текущий поток
istream & get( char *buf, streamsize num );

// Аналогичны соответствующим функциям get( ), но имеют следующие
// отличия. Чтение завершается не перед символом новой строки или
// символом lim, а включая этот символ, который копируют в buf
istream & getline( char *buf, streamsize num, char lim );
istream & getline( char *buf, streamsize num );

// Считывает num символов (или все символы до конца файла, если их
// меньше num) в buf и возвращает ссылку на текущий поток. Строка
// buf не завершается автоматически символом завершения строки
istream & read( char *buf, streamsize num );
// Работает аналогично методу read( ), но возвращает количество
// прочитанных символов
streamsize readsome( char *buf, streamsize num );

// Считывает и пропускает из потока num-1 символов (или пока не
// встретится символ lim). Возвращает ссылку на текущий поток
istream & ignore( streamsize num =1, int lim = EOF );

// Возвращает следующий символ, не удаляя его из потока, или EOF,
// если достигнут конец файла
int peek( );
// Помещает в поток символ ch, который становится текущим при
// извлечении из потока. Возвращает ссылку на текущий поток
istream & putback( char ch );
// Возвращает последний прочитанный символ в поток и возвращает
// ссылку на текущий поток
istream & unget( );

// Возвращает количество символов, прочитанных из потока с помощью
// последней функции неформатированного ввода (см. ранее)
streamsize gcount( ) const;

// Устанавливает текущую позицию чтения в позицию pos относительно
// начала потока
istream & seekg( pos_type pos );
// Перемещает текущую позицию чтения на off байтов, относительно
// позиции dir. Второй аргумент в вызове этой функции может
// принимать следующие значения: ios::beg (от начала файла),

```

```
// ios::cur (от текущей позиции), ios::end (от конца файла)
istream & seekg( off_type off, ios::seekdir dir );
// Возвращает текущую позицию чтения потока
pos_type tellg( );

В классе ostream (включаемый файл <ostream>) определены аналогичные методы
для неформатированного вывода:

// Записывает содержимое потока вывода из буфера на физическое
// устройство. Возвращает ссылку на текущий поток вывода
ostream & flush( );

// Выводит в поток вывода символ, заданный аргументом, и возвращает
// ссылку на поток
ostream & put( char );

// Записывает в выходной поток num символов из buf и возвращает
// ссылку на поток
ostream & write( const char *buf, streamsize num );

// Устанавливает текущую позицию записи в позицию pos относительно
// начала потока
ostream & seekp( pos_type pos );
// Перемещает текущую позицию записи на off байтов, относительно
// позиции dir. Второй аргумент в вызове этой функции может
// принимать следующие значения: ios::beg (от начала файла),
// ios::cur (от текущей позиции), ios::end (от конца файла)
ostream & seekp( off_type off, ios::seekdir dir );

// Возвращает текущую позицию записи потока
pos_type tellp( );
```

5.6. Файловый ввод-вывод

Когда программа начинает работу, четыре потока инициализируются автоматически и связываются с соответствующими им стандартными файлами (устройствами) — это `cin`, `cout`, `cerr` и `clog`. Если нужно инициализировать еще и другие потоки и связать их с некоторыми файлами, то сделать это следует явно.

Любой инициализируемый файловый поток может быть либо потоком ввода, либо потоком вывода, либо потоком ввода-вывода. *Поток ввода* — это всегда объект класса `ifstream` (см. включаемый файл `fstream`), порожденного из класса `istream` (см. включаемый файл `istream`). *Поток вывода* — это всегда объект класса `ofstream` (см. включаемый файл `fstream`), порожденного из класса `ostream` (см. включаемый файл `ostream`). *Поток ввода-вывода* — это всегда объект класса `fstream` (см. включаемый файл `fstream`), порожденного из класса `iostream` (см. файл `iostream`). В каждом из этих классов имеются конструкторы, выполняющие работу по связыванию объекта класса с соответствующим файлом.


```

// умолчанию): файл должен существовать

ios::out    = 0x02    // Открыть файл для записи (для объектов
// класса ofstream действует по
// умолчанию). При отсутствии файла
// создать его

ios::ate    = 0x04    // После открытия файла указатель потока
// для записи однократно позиционируется
// в конец файла

ios::app    = 0x08    // Открыть файл для записи только в конец
// файла

ios::trunc  = 0x10    // Удалить старое содержимое файла

ios::binary = 0x20    // Открыть файл в двоичном режиме

```

Сделаем ряд *очень важных замечаний*:

- ❑ режим `ios::in` эквивалентен режиму `"r"` функции открытия файла языка C `fopen()`;
- ❑ режим `ios::out` или комбинация режимов `ios::out | ios::trunc` эквивалентны режиму `"w"` функции языка C `fopen()`;
- ❑ комбинация режимов `ios::out | ios::app` эквивалентна режиму `"a"` функции языка C `fopen()`;
- ❑ комбинация режимов `ios::in | ios::out` эквивалентна режиму `"r+"` функции языка C `fopen()`;
- ❑ комбинация режимов `ios::in | ios::out | ios::trunc` эквивалентна режиму `"w+"` функции языка C `fopen()`;
- ❑ комбинация режимов `ios::in | ios::out | ios::app` эквивалентна режиму `"a+"` функции языка C `fopen()`;
- ❑ комбинации с режимами `ios::ate` и `ios::binary` не комментируются, а остальные комбинации недопустимы.

Открыть файл в программе можно с использованием либо конструкторов, либо метода `open()`, имеющего такие же параметры, как и в соответствующем конструкторе, например:

```

#include <iostream>    // Ввод-вывод C++
#include <fstream>    // Для файловых потоков
using namespace      // Используем стандартное
    std;             // пространство имен

// ...
// Использование конструктора
ifstream    inpf( "input.txt" );
if( !inpf )
{

```

```

    cout << "Ошибка 1 открытия файла input.txt для чтения"
        << endl;
    exit( 1 );
}

// Использование метода open( )
ofstream      outf;
outf.open( "output.txt", ios::out );
if( !outf )
{
    cout << "Ошибка 2 открытия файла output.txt для "
        "записи" << endl;
    exit( 2 );
}

```

Чтение и запись в файловый поток выполняются либо с помощью операций "<<" и ">>", аналогичных рассмотренным ранее, либо с помощью рассмотренных выше методов классов.

Для закрытия файлового потока определен метод `close()`:

```
void close( );
```

Но поскольку этот метод неявно выполняется деструктором, то явный вызов метода `close()` необходим только тогда, когда требуется закрыть файловый поток раньше конца его области видимости (действия).

Рассмотрим *два иллюстрирующих примера* (листинги 5.8—5.12). В первом из них файловый объект и открытие-закрытие файла для работы выполняются с использованием возможностей классов `ifstream` и `ofstream`. Во втором примере для этой цели используется простой производный от `fstream` класс с методами открытия-закрытия файлов, что позволяет весьма лаконично открывать и закрывать файлы, сохраняя исчерпывающую обработку ошибок.

СОВЕТ

Самым тщательным образом изучите эти примеры — в них в виде комментариев изложена очень важная информация.

Листинг 5.8. Файл FileInpOut.cpp

```

/*
    Файловый ввод-вывод на основе стандартной библиотеки языка C++ (классы ifstream,
    ofstream).
    В. Давыдов, консольное приложение, Microsoft Visual C++ .NET
*/

#include <iostream>          // Ввод-вывод C++
#include <fstream>           // Файловый ввод-вывод C++
#include <string>            // Строки C++

using namespace            // Используем стандартное

```

```
std;           // пространство имен

// Один из рекомендуемых стандартом языка вариантов оформления заголовка
//  главной функции
int main( )           // Возвращает 0 при успехе
{
    // Создаем файловый объект для ввода из файла FileInpOut.inp
    ifstream  FInp( "FileInpOut.inp" );
    // Конструктор класса ifstream имеет умалчиваемое значение ios::in
    //  для второго параметра, которое и использовано в данном случае
    if( !FInp )
    {
        cout << "Ошибка 10 открытия файла FileInpOut.inp для чтения"
                << endl;
        exit( 10 );
    }

    // Файловый ввод
    int      i;           // 21
    double   d;           // 1.5
    // Строка-буфер для пропуска пояснений в файле ввода (используем "не
    //  слепой" ввод)
    string    s;
    FInp >> s >> i >> s >> d;
    if( FInp.eof( ) )
    {
        cout << "Достигнут конец файла FileInpOut.inp" << endl;
    }
    if( !FInp )
    {
        cout << "Ошибка 20 ввода из файла FileInpOut.inp" << endl;
        exit( 20 );
    }

    // Закрытие файла (освобождение файлового объекта)
    FInp.close( );

    // Создаем файловый объект для записи в файл FileInpOut.out
    ofstream  FOut( "FileInpOut.out" );
    // Конструктор класса ofstream имеет умалчиваемое значение ios::out
    //  для второго параметра, которое и использовано в данном случае
    if( !FOut )
    {
        cout << "Ошибка 30 открытия файла FileInpOut.out для записи"
                << endl;
        exit( 30 );
    }
}
```

```

    }

    // Запись в файл
    FOut << "Прочитано: i = " << i << endl;

    // Закрывание файла (освобождение файлового объекта)
    FOut.close( );

    // С помощью файлового объекта открываем файл FileInOut.out для
    // дозаписи
    FOut.open( "FileInOut.out", ios::out | ios::app );
    if( !FOut )
    {
        cout << "Ошибка 40 открытия файла FileInOut.out для дозаписи"
              << endl;
        exit( 40 );
    }

    // Дозапись в файл
    FOut << "Прочитано: d = " << d << endl;

    // Закрывание файла (освобождение файлового объекта)
    FOut.close( );

    // Стандарт языка разрешает опускать оператор возврата
    // return 0;
}

```

Листинг 5.9. Файл ввода FileInOut.inp

```
i= 21 d= 1.5
```

Листинг 5.10. Файл вывода FileInOut.out

```
Прочитано: i = 21
Прочитано: d = 1.5
```

Теперь рассмотрим второй пример (листинг 5.11).

Листинг 5.11. Файл IOFile.h

```
/*
    Открытие-закрывание файлов на базе класса IOFILE, производного от класса fstream.
    Класс IOFILE обеспечивает также использование перегруженных операций вывода (<<) и
    ввода (>>) для стандартных типов.

```

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET.
*/

// Предотвращение многократного включения данного файла
#ifndef __IOFILE_H
#define __IOFILE_H

#include <iostream> // Для потокового ввода-вывода
#include <fstream>   // Для работы с файлами

using namespace     // Используем стандартное
    std;            // пространство имен

// *****
// Класс для открытия-закрытия файлов на базе класса fstream
class IOFILE : public fstream
{
    // Методы

public:

    void open_f( char *pFileName, int mode,
                 int error_num );
    void close_f( char *pFileName, int error_num );
};

// Реализация методов
// *****
// Открытие файла
void IOFILE :: open_f(
    // Указатель на строку - имя файла
    char *pFileName,
    int mode, // Режим доступа: 1-63
    // Код ошибки
    int error_num )
{
    // Проверка допустимости режима
    if( ( mode < 1 ) || ( mode > 63 ) )
    {
        cout << " Error " << error_num << ". Error of"
              << " the mode of access to the file: mode = "
              << mode << " (the values are admitted 1..63)" << endl;
        exit( error_num );
    }
}

```

```

    }

    // Открытие файла
    open( pFileName, mode );
    if( fail( ) )
    {
        cout << " Error " << error_num <<
            ". The open error of the file " << pFileName
            << " with access " << mode << endl;
        exit( error_num );
    }

    return;
}

// *****
// Закрытие файла
void IOFILE :: close_f(
    // Указатель на строку - имя файла
    char *pFileName,
    // Код ошибки
    int error_num )
{
    // Закрытие файла
    close( );
    if( fail( ) )
    {
        cout << " Error " << error_num << ". Error of file closing "
            << pFileName << endl;
        exit( error_num );
    }

    return;
}

#endif

```

Листинг 5.12. Файл IOFILE.CPP

```

/*
    Тестирование функций открытия-закрытия файлов, определяемых в классе IOFILE.
    Этот класс является производным от класса fstream, и его определение приведено во
    включаемом файле IOFILE.H. Тестирование работы с файлами в различных режимах
    с использованием методов произвольного доступа к файлам.

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET)

```

*/

```

```

// Класс для открытия-закрытия файлов на базе класса fstream и поддержки
// ввода-вывода языка C++
#include "IOFILE.H"

// Тестирование
int main( )           // Возвращает 0 при успехе
{
    IOFILE      f_in_out; // Файловый объект для чтения и записи

    // Открытие файла для чтения и записи ios::in | ios::out - файл
    // должен существовать( аналог режима "r+" для функции C fopen)
    f_in_out.open_f( "iofile.io", ios::in | ios::out, 10 );

    cout << "\nTesting of operation with the file open for"
           " reading-\nrecording - the file should exist - "
           "differently there is an \n error " << endl;
    // Перед запуском программы в файле содержатся следующие числа:
    // 10 20 30 40 50 60 70 80 90

    // Извлекаем три числа (по умолчанию извлечение выполнится с начала
    // файла) и выводим их на экран
    int      tmp;
    for( int i = 1; i <= 3; i++ )
    {
        // Для сокращения размера исходного текста здесь и далее проверку
        // достижения конца файла ввода и ошибки ввода не производим.
        // Вместе с тем в реальных программах это следует делать
        // обязательно
        f_in_out >> tmp; cout << tmp << endl;
    }
    // После ввода указатель ввода указывает в потоке на символ пробела,
    // предшествующий символу 4
    /* Экран выглядит следующим образом:
Testing of operation with the file open for reading -
recording - the file should exist - differently there is an
error
10
20
30
*/

    // Запоминаем текущую позицию указателя для чтения
    long      CurGetStream = f_in_out.tellg( );

    // Позиционируем указатель для записи в позицию, которую запомнили ранее

```

```

// CurGetStream: CurGetStream - смещение, ios::beg - относительно
// начала файла
f_in_out.seekp( CurGetStream, ios::beg );

// Записываем в файл три числа: 1, 2, 3. При записи после вывода
// очередного значения числа использование манипулятора endl
// приводит к выводу двух управляющих символов - '\r' и '\n'.
// Запись выполняется поверх
for( i = 1; i <= 3; i++ )
{
    f_in_out << i << endl;
}
/* Файл теперь выглядит следующим образом:
10 20 301
2
3
70 80 90
Видим, что эти числа записались, начиная с той позиции, которую мы установили
"поверх" прежнего содержимого файла
*/

// Запоминаем текущую позицию указателя для записи
long CurPutStream = f_in_out.tellp( );

// Позиционируем указатель для чтения в позицию, которую запомнили ранее
// CurPutStream: CurPutStream - смещение, ios::beg - относительно
// начала файла
f_in_out.seekg( CurPutStream, ios::beg );

// Извлекаем три числа и выводим их на экран
for( i = 1; i <= 3; i++ )
{
    f_in_out >> tmp; cout << tmp << endl;
}
/* Экранный вывод теперь выглядит так:
Testing of operation with the file open for reading -
recording - the file should exist - differently there is an
error
10
20
30
70
80
90
Видим, что чтение выполнено с того места, которое мы указали
*/

```

```
// Позиционируем файл для записи в конец файла: 0 - смещение,
//   ios::end - относительно конца файла
f_in_out.seekp( 0, ios::end );

// Записываем в файл три числа: 1, 2, 3
for( i = 1; i <= 3; i++ )
{
    f_in_out << i << endl;
}
/* Файл теперь выглядит следующим образом:
10 20 301
2
3
70 80 90
1
2
3

*/

// Сброс флагов состояния и закрытие файла
f_in_out.clear( );    // !!! Делайте это обязательно
f_in_out.close_f( "iofile.io", 20 );

// Открытие файла для режима "ios::in | ios::out | ios::app" (аналог
//   режима "a+" для функции C fopen). Файл открывается для чтения и
//   дозаписи в конец; если файл не существовал, то он создается
f_in_out.open_f( "iofile1.io", ios::in | ios::out | ios::app, 30 );

cout << "\n Testing of operation with the file open in the mode \n"
      << "\"ios::in+ios::out+ios::app\"" <<
      << "\n If the file did not exist, he forms " << endl;
/* Перед запуском программы в файле iofile1.io содержатся следующие числа:
1
2
3

*/

// Выводим в файл три числа
f_in_out << 100 << endl;
f_in_out << 200 << endl;
f_in_out << 300 << endl;
/* Файл теперь выглядит следующим образом:
1
2
3
```

100
200
300

Видим, что эти числа записались в конец файла
*/

```
// Позиционируем указатель для чтения в начало файла: 0 - смещение,
// ios::beg - относительно начала файла
f_in_out.seekg( 0, ios::beg );
```

```
// Извлекаем три числа и выводим их на экран
for( i = 1; i <= 3; i++ )
{
    f_in_out >> tmp; cout << tmp << endl;
}
```

/* Экран выглядит следующим образом:

```
Testing of operation with the file open for reading -
recording - the file should exist - differently there is an
error
```

10
20
30
70
80
90

Testing of operation with the file open in the mode

"ios::in+ios::out+ios::app"

If the file did not exist, he forms

1
2
3

*/

```
// Сброс флагов состояния и закрытие файла
f_in_out.clear( ); // !!!
f_in_out.close_f( "iofile1.io", 40 );
```

```
// Открытие файла для режима "ios::in | ios::out" (аналог режима "r+"
// для функции C fopen). Файл открывается для чтения и записи и
// должен существовать
f_in_out.open_f( "iofile2.io", ios::in | ios::out, 50 );
```

/* Перед запуском программы в файле iofile2.io содержатся
следующие числа:

1

```
2
3
    */

    // Извлекаем число
    f_in_out >> tmp;      // Будет извлечено число 1

    // Запоминаем текущую позицию указателя для чтения
    CurGetStream = f_in_out.tellg( );

    // Позиционируем указатель для записи в конец файла: 0 - смещение,
    //   ios::end - относительно конца файла
    f_in_out.seekp( 0, ios::end );
    // Выводим в файл три числа
    f_in_out << 100 << endl;
    f_in_out << 200 << endl;
    f_in_out << 300 << endl;
    /* Файл теперь выглядит следующим образом:
1
2
3
100
200
300

    Видим, что эти числа записались в конец файла
    */

    // Позиционируем указатель для чтения в начало файла: 0 - смещение,
    //   ios::beg - относительно начала файла
    f_in_out.seekg( 0, ios::beg );

    // Извлекаем три числа: будут извлечены числа 1, 2 и 3
    for( i = 1; i <= 3; i++ )
    {
        f_in_out >> tmp;
    }

    // Позиционируем указатель для чтения в позицию, которую запомнили ранее
    //   CurGetStream: CurGetStream - смещение, ios::beg - относительно
    //   начала файла
    f_in_out.seekg( CurGetStream, ios::beg );

    // Извлекаем число
    f_in_out >> tmp;      // Будет извлечено число 2

    /* Файл теперь выглядит следующим образом:
```

```

1
2
3
100
200
300
    */

    // Сброс флагов состояния и закрытие файла
    f_in_out.clear( );    // !!!
    f_in_out.close_f( "iofile2.io", 60 );

    return 0;            // Этот оператор стандарт языка
                        // разрешает опускать
}

```

В заключение рассмотрим еще один практически значимый пример (листинги 5.13—5.15). Он иллюстрирует следующие важные аспекты:

- ☐ потоковый ввод-вывод языка C++, закрепляя изложенный ранее материал;
- ☐ передачу параметров из конструктора производного класса в конструктор базового класса при использовании шаблонов классов;
- ☐ особенности работы с двумерным массивом, размещаемым в динамической памяти.

Листинг 5.13. Файл MATRIXB.H

```

/*
    Содержит определение шаблона базовых классов для работы с матрицами (двумерными массивами).
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET)
*/

// Предотвращение многократного включения данного файла
#ifndef __MATRIXB_H
#define __MATRIXB_H

    // Подключение определения класса IOFILE для открытия-закрытия файлов
    // на базе класса fstream, поддерживающего также ввод-вывод C++.
    // Исходный текст этого файла приведен в предыдущем примере
    #include "iofile.h"

    // Шаблон базовых классов для работы с матрицами (двумерными
    // массивами). Обратите внимание на заголовок объявления шаблона
    // классов
    template< class T > // T - обобщенный тип элемента

```

```

class Matrix_B
{
    // Данные

protected:

    T      **array; // Указатель на массив указателей на строки
    unsigned int
        rows,      // Количество строк матрицы
        columns;    // Количество столбцов матрицы

    // Методы

public:

    // Конструктор
    Matrix_B( unsigned int nrow, unsigned int ncol );

    // Конструктор копирования
    Matrix_B( const Matrix_B< T > &copy );

    // Деструктор
    ~Matrix_B( void );

    // Чтение элементов матрицы из заданного файла
    void ReadMatrix( char *file_inp );

};

//*****
// Конструктор - обратите внимание на заголовок определения
// конструктора (определение размещается в данном случае вне блока
// шаблона классов)
template< class T >
Matrix_B< T > :: Matrix_B(
    unsigned int
        nrow,      // Число строк матрицы
    unsigned int
        ncol )     // Число столбцов матрицы
{
    // Печать сообщения о вызове конструктора (используется в
    // методике испытания программы). Обратите внимание, что для
    // вывода сообщения на экран используется перегруженная в
    // классе iostream операция <<
    cout << "The constructor of the class Matrix_B is called"

```

```

    << endl;

// Проверка попытки создания матрицы с единичными или нулевыми
// размерами строки или столбца
if( nrow < 2 || ncol < 2 )
{
    cerr << "\n Error 10. One or both sizes of a "
          "matrix have values smaller two" <<
          "\n rows: " << nrow << " columns: " << ncol << endl;
    exit( 10 );
}

// Динамическое выделение памяти под матрицу
//*****
// Выделение памяти под массив указателей на строки
array = new T * [ nrow ];
if( !array )      // Проверка выделения памяти
{
    cerr << "\n Error 20 of the class Matrix_B of "
          "selection of memory under a pointer array"
          " on strings" << endl;
    exit( 20 );
}
// Выделение памяти под строки
for( unsigned int i = 0; i < nrow; i++ )
{
    // Выделение памяти под строку
    array[ i ] = new T [ ncol ];
    // Проверка выделения памяти под строку
    if( !array[ i ] )
    {
        cerr << "\n Error 30 of the class Matrix_B"
              " of selection of memory under string" << endl;
        exit( 30 );
    }
}

// Инициализация матрицы
for( i = 0; i < nrow; i++ )
    for( unsigned int j = 0; j < ncol; j++ )
        array[ i ][ j ] = ( T )0;

rows = nrow;      // Запоминание числа строк,
columns = ncol;    // столбцов матрицы

return;

```

```

}

//*****
// Конструктор копирования
template< class T >
Matrix_B< T > :: Matrix_B(
    // Ссылка на объект класса
    const Matrix_B< T > &copy )
{
    // Печать сообщения о вызове конструктора копирования
    // (используется в методике испытания программы)
    cout << "The konstruktur of copying of the class "
         "Matrix_B is called" << endl;

    // Динамическое выделение памяти под матрицу
    // *****
    // Выделение памяти под массив указателей на строки
    array = new T * [ copy.rows ];
    if( !array )        // Проверка выделения памяти
    {
        cerr << "\n Error 20 of the class Matrix_B of "
              "selection of memory under a pointer array"
              " on strings" << endl;
        exit( 20 );
    }
    // Выделение памяти под строки
    for( unsigned int i = 0; i < copy.rows; i++ )
    {
        // Выделение памяти под строку
        array[ i ] = new T [ copy.columns ];
        // Проверка выделения памяти под строку
        if( !array[ i ] )
        {
            cerr << "\n Error 30 of the class Matrix_B"
                  " of selection of memory under string" << endl;
            exit( 30 );
        }
    }
}

// Копирование значений матрицы
for( i = 0; i < copy.rows; i++ )
    for( unsigned int j = 0; j < copy.columns; j++ )
        array[ i ][ j ] = copy.array[ i ][ j ];

rows = copy.rows;
columns = copy.columns;

```

```

}

//*****
// Деструктор
template< class T >
Matrix_B< T > :: ~Matrix_B( void )
{
    // Печать сообщения о вызове деструктора (используется в методике
    //  испытания программы)
    cout << "Is called destructor of the class Matrix_B" << endl;

    // Высвобождение памяти из-под строк
    for( unsigned int i = 0; i < rows; i++ )
        delete [ ] array[ i ];

    delete [ ] array; // Высвобождение памяти из-под массива
                      //  указателей на строки
}

//*****
// Чтение элементов матрицы из заданного файла
template< class T >
void Matrix_B< T > :: ReadMatrix(
    // Указатель на строку с именем файла для чтения
    char *file_inp )
{
    // Объект класса IOFILE, производного от класса fstream
    IOFILE f_in;

    // Открытие файла для чтения
    f_in.open_f( file_inp, ios::in, 40 );

    // Чтение матрицы
    for( unsigned int i = 0; i < rows; i++ )
    {
        for( unsigned int j = 0; j < columns; j++ )
        {
            // Чтение элемента матрицы - используется перегруженная в
            //  классе iostream операция >>
            f_in >> array[ i ][ j ];
            // Проверка на отсутствие в файле элемента матрицы -
            //  обратите внимание, как выполняется проверка
            if( f_in.eof( ) )
            {
                cerr << "\n Error 50. In the file of"

```

```

        " input there is no unit of a matrix "
        << ( i*columns + j +1 ) <<
        "\n The end-of-file is detected" << endl;
    exit( 50 );
}
// Проверка чтения элемента матрицы - обратите внимание,
// как выполняется проверка
if( !f_in )
{
    cerr << "\n Read error 60 of a unit "
           "of a matrix " << i*columns + j << endl;
    exit( 60 );
}
}

// Закрытие файла ввода
f_in.close_f( file_inp, 70 );

return;
}

```

```
#endif
```

Листинг 5.14. Файл MATRIXD.H

```

/*
    Содержит определение шаблона производных классов для работы с матрицами (двумер-
    ными массивами). Определение шаблона базовых классов содержится в файле MATRIXB.H.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET
*/

// Предотвращение многократного включения данного файла
#ifndef __MATRIXD_H
#define __MATRIXD_H

    // Подключение определения шаблона базовых классов для работы с
    // матрицами (двумерными массивами)
    #include "matrixb.h"

    /**
    // Шаблон производных классов для работы с матрицами (двумерными
    // массивами). Обратите внимание на заголовок объявления шаблона
    // классов
    template< class T > // T - обобщенный тип элемента
    class Matrix_D : public Matrix_B< T >
    */

```

```

{
    // Методы

public:

    // Конструктор
    Matrix_D( unsigned int nrow, unsigned int ncol );

    // Печать матрицы
    void PrintMatrix( IOFILE & );

    // Перегрузка операции сложения
    Matrix_D< T > operator+( const Matrix_D< T > & );

    // Перегрузка операции присваивания
    Matrix_D< T > & operator=( const Matrix_D< T > & );

    // Перегрузка операции индексации
    T * & operator[ ]( unsigned int IxRow );

};

/*****
// Конструктор - в данном случае он нужен для передачи размеров
// создаваемой матрицы в конструктор базового класса. Обратите
// внимание на оформление заголовка конструктора - это очень важно!
template< class T >
Matrix_D< T > :: Matrix_D( unsigned int nrow,
                           unsigned int ncol )
: Matrix_B< T >( nrow, ncol )
{
    // Печать сообщения о вызове конструктора (используется в
    // методике испытания программы)
    cout << "The constructor of the class Matrix_D is called"
    << endl;
}

/*****
// Перегрузка операции сложения - обратите внимание на заголовок
// функции
template< class T >
Matrix_D< T > Matrix_D< T > :: operator+(
    // Объект класса (операнд 1)
    const Matrix_D< T > &OpR )
    // Ссылка на объект класса

```

```

        //      (операнд 2)
    {
        // Проверка совместимости размеров матриц для операции сложения
        if( rows != OpR.rows || columns != OpR.columns )
        {
            cerr << "\n Error 80 of the class matrix. "
                "\n Above matrixes the operation '+' "
                "can not be produced. \n For addition of"
                " matrixes their string and column sizes"
                " should coincide. \n Sizes of the left "
                "operand: " << rows << ' ' << columns <<
                "\n Sizes of the righth operand: " <<
                OpR.rows << ' ' << OpR.columns << endl;
            exit( 80 );
        }

        // Создание пустого объекта (результатирующая матрица)
        Matrix_D< T > result( rows, columns );

        // Сложение матриц
        for( unsigned int i = 0; i < rows; i++ )
            for( unsigned int j = 0; j < columns; j++ )
                result.array[ i ][ j ] = array[ i ][ j ] +
                    OpR.array[ i ][ j ];

        return result;
    }

    //*****
    // Перегрузка операции присваивания
    template< class T >
    Matrix_D< T > & Matrix_D< T > :: operator=(
        const Matrix_D< T > &OpR )
    {
        // Если размеры объекта не подходят
        if( rows != OpR.rows || columns != OpR.columns )
        {
            cout << "\n Error 90. Unequal sizes at assignment" << endl;
            exit( 90 );
        }

        // Выполнение операции присваивания
        for( unsigned int i = 0; i < rows; i++ )
            for( unsigned int j = 0; j < columns; j++ )
                array[ i ][ j ] = OpR.array[ i ][ j ];
    }

```

```

        rows = OpR.rows;
        columns = OpR.columns;

        return *this;
    }

    //*****
    // Печать матрицы
    template< class T >
    void Matrix_D< T > :: PrintMatrix( IOFILE &f_out )
    {
        for( unsigned int i = 0; i < rows; i++ )
        {
            for( unsigned int j = 0; j < columns; j++ )
            {
                f_out << array[i][j] << ' ';
            }
            f_out << endl;
        }

        return;
    }

    //*****
    // Перегрузка операции индексации
    template< class T >
    T * & Matrix_D< T > :: operator[ ](
        unsigned int
            IxRow )    // Индекс строки
    {
        return array[ IxRow ];
    }

#endif

```

Листинг 5.15. Файл MATRIX.CPP

```

/*
    Тестирование шаблонов базового и производного классов для работы с матрицами
    (двумерными массивами).

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET
*/

// Включение определения шаблона производных классов для работы
// с матрицами

```

```
#include "matrixd.h"

// Тестирование
int main( )           // Возвращает 0 при успехе
{
    // Объект класса IOFILE, производного от класса fstream, открываем
    // для записи в файл matrix.out
    IOFILE      f_out;
    f_out.open_f( "matrix.out", ios::out, 100 );

    // Создание массивов пустых объектов для проведения тестирования,
    // заполнение их значениями из файла и их печать
    Matrix_D< double >
        double_0( 2, 2 ), double_1( 2, 2 ),
        double_2( 2, 2 );
    double_0.ReadMatrix( "matrix.in" );
    double_1.ReadMatrix( "matrix.in" );
    double_2.ReadMatrix( "matrix.in" );
    f_out << "\n double_0:" << endl;
    double_0.PrintMatrix( f_out );
    f_out << "\n double_1:" << endl;
    double_1.PrintMatrix( f_out );
    f_out << "\n double_2:" << endl;
    double_2.PrintMatrix( f_out );

    // Тестирование перегруженной операции [ ]
    f_out << "\n Тестирование перегруженной операции [ ]:"
        "\ndouble d = double_0[ 1 ][ 1 ]; " << endl;
    double      d = double_0[ 1 ][ 1 ];
    f_out << "\n d: " << d << endl;
    f_out << "\n Тестирование перегруженной операции [ ]:"
        "\ndouble_0[ 1 ][ 1 ] = 12; d = double_0[ 1 ][ 1 ]; " << endl;
    double_0[ 1 ][ 1 ] = 12;
    d = double_0[ 1 ][ 1 ];
    f_out << "\n d: " << d << endl;

    // Тестирование элементарных выражений
    f_out << "\n Тестирование элементарных выражений \n"
        "double_2 = double_0 + double_1; " << endl;
    double_2 = double_0 + double_1;

    f_out << "\n double_2: " << endl;
    double_2.PrintMatrix( f_out );

    // Закрытие файла вывода
    f_out.close_f( "matrix.out", 110 );
```

```
    return 0;  
}
```

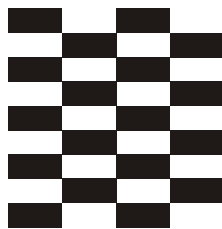
Вы, наверное, уже обратили внимание на то, что в консольных приложениях Visual C++ 6.0 экранный вывод выполняется на английском языке. Объясняется это тем, что кодировка букв русского алфавита в среде Windows (кодировка ANSI) и в среде DOS (кодировка OEM) не одинакова. Для букв же английского алфавита используется одинаковая кодировка. Впрочем, ранее в программном проекте с файлом `mapir2.cpp` был продемонстрирован простой и эффективный способ перенаправления экранного вывода в файл на магнитном диске, разрешающий эту коллизию.

5.7. Вопросы для самопроверки [4]

1. Как определить, когда использовать перегруженные операции ввода ">>" и вывода "<<", а когда другие методы классов потока?
2. Какие отличия между `cerr` и `clog`?
3. Зачем создавать потоки ввода-вывода, если отлично работают семейства функций `scanf()`-`fscanf()`, `printf()`-`fprintf()`-`sprintf()`?
4. Когда следует применять метод `ignore()`?
5. Что такое перегруженный оператор ввода и как он работает?
6. Что такое перегруженный оператор вывода и как он работает?
7. Какая ширина поля вывода принимается по умолчанию при выводе целых чисел?
8. Какое значение возвращает перегруженный оператор вывода?
9. Какой обязательный параметр принимает обычный конструктор объекта `ofstream`?
10. Что обеспечивает элементарный режим открытия `ios::ate`?

Ответы на вопросы с целью проверки можно посмотреть в *разд. П1.5 приложения 1*.

Глава 6



Обработка исключительных ситуаций [2, 4, 7, 9]

Исключительная ситуация или *исключение* — это возникновение непредвиденного или аварийного события, которое может порождаться самыми различными причинами, например из-за недостаточности распределяемых ресурсов. Язык C++ содержит средства обработки исключительных ситуаций. Эти средства чаще всего используются для обработки *ошибок*, возникающих при работе программы. Ошибки в программе возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. Таким образом, ошибка в работе программы является *частным случаем* исключительной ситуации.

Если не использовать средства обработки исключительных ситуаций, то возникновение исключения приводит обычно к завершению работы программы с выдачей системного сообщения об ошибке. Обработка же исключений позволяет диагностировать возникшее исключение (в частном случае ошибку), восстановить работоспособность программы и продолжить ее выполнение.

Обработка исключительных ситуаций позволяет логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Еще более важным является то, что в месте возникновения ошибки может быть неизвестно, как следует обработать ошибку, а обработчик ошибки не может определить место возникновения ошибки. Другим достоинством использования средств обработки исключений языка C++ является то, что для передачи информации об ошибке в место ее обработки не требуется применять возвращаемое значение, параметры или глобальные переменные. Поэтому интерфейс функций не раздувается. Это особенно важно, например, для конструкторов, которые по синтаксису не могут возвращать значение.

ЗАМЕЧАНИЕ

В принципе, ничто не мешает рассматривать в качестве исключений не только ошибки, но и нормальные ситуации, возникающие при обработке данных. При этом их обработка как исключительных ситуаций не имеет преимуществ перед другими решениями и не улучшает структуру и читаемость программы.

Подведем краткий итог сказанному.

- *Обработка исключений* языка C++ обеспечивает передачу управления и информации о возникшей ситуации в неопределенную точку вызова, где было выражено желание обрабатывать исключения этого типа. Исключения любого типа могут возникать и обрабатываться.

- Язык C++ обеспечивает обработку *явных исключений*, о генерации которых должен позаботиться сам разработчик. Это означает, что программист должен проверить необходимость обработки той или иной ситуации и возбудить исключение. Обработка исключений языка C++ является обработкой с завершением, т. е. у обработчика *нет* возможности продолжить выполнение программы с точки возбуждения исключения. Вместе с тем *сохраняется* возможность выполнения программного кода, следующего за обработчиками исключений.
- При возникновении исключения обработчику передается произвольное количество информации с контролем типов.

6.1. Общий механизм обработки исключений

Фрагмент программного кода, в котором может произойти ошибка, должен входить в *контролируемый блок*. Контролируемый блок представляет собой составной оператор, перед которым записано служебное слово `try`, переводимое как *проба, испытание*.

Обработка исключительной ситуации реализуется следующим образом.

1. Обработка исключения начинается с проверки появления ошибки (напомним, что указанную проверку должен предусмотреть разработчик). Во фрагменте программного кода, где обнаружена ошибка, генерируется исключение (это также должен предусмотреть разработчик). Для этого используется служебное слово `throw` (переводится как *бросать, возбуждать*) с параметром, определяющим вид исключения. Параметр может быть константой, переменной или объектом класса и используется для передачи информации об исключении его обработчику.
2. Отыскивается соответствующий обработчик исключения, ему передается управление и информация об исключении (параметр из `throw`).
3. Если обработчик исключения не найден, вызывается стандартная функция `terminate()`, которая, в свою очередь, вызывает функцию `abort()`, аварийно завершающую текущий процесс. Можно использовать собственную функцию `terminate()`, переопределив стандартную функцию.

В [3] (см. разд. 3.5) говорилось о том, что при вызове каждой функции в системном стеке создается область памяти для хранения локальных переменных функции, адресов аргументов функции, передаваемых по ссылке, копий аргументов, передаваемых по значению, и адреса возврата в вызывающую функцию. Термин *стек вызовов* обозначает последовательность вызванных, но еще не завершившихся функций. *Раскручиванием стека* называется процесс освобождения памяти из под локальных переменных функции, адресов аргументов функции, передаваемых по ссылке, копий аргументов, передаваемых по значению, и возврата управления в место вызова функции. Когда функция завершает работу, происходит естественное раскручивание стека. Тот же самый механизм используется и при обработке исключений. Поэтому, после того, как исключение было обработано, исполнение программы *не может* быть продолжено с точки генерации исключения. Подробнее этот механизм будет рассмотрен в разд. 6.3.

6.2. Синтаксис исключений

Служебное слово **try** предназначено для обозначения *контролируемого блока* — кода, в котором может генерироваться исключение. Блок заключается в фигурные скобки и представляет собой составной оператор:

```
try
{
    ...
}
```

Блок содержит последовательность операторов, включая вызовы функций, которые проверяются на возникновение исключения. Такой блок называют *защищенным блоком*. Все функции, прямо или косвенно вызываемые из *try*-блока, также считаются ему принадлежащими. По сути, именно в рамках защищенного блока может встретиться *выражение возбуждения исключения*, где необязательное *выражение* после служебного слова **throw** как раз и определяет тип и информацию, описывающую само исключение.

Генерация (возбуждение) исключения происходит по служебному слову **throw**, которое употребляется либо с выражением, либо без него:

```
throw [ выражение ] ;
```

Отметим, что служебное слово **throw** (бросать) было использовано вместо слов *signal* или *raise*, так как стандартная C-библиотека уже имеет функции с такими именами. Тип выражения, стоящего после **throw**, определяет тип порождаемого исключения. Как уже упоминалось ранее, исключение может быть как стандартного типа, так и типа, определенного пользователем (в том числе *может иметь тип класса*). При генерации исключения выполнение текущего защищенного блока прекращается и происходит поиск соответствующего обработчика исключения и передача ему управления. Чаще всего исключение генерируется *не непосредственно* в **try**-блоке, а в функциях, прямо или косвенно вложенных в него.

При необходимости, **try**-блоки можно вкладывать друг в друга. При этом не всегда исключение, возникшее во внутреннем блоке, может быть правильно обработано. В этом случае исключение передается на более высокий уровень с помощью служебного слова **throw** без *выражения*.

Обработчики исключений начинаются со служебного слова **catch** (переводится как *перехватывать*), за которым в круглых скобках следует тип обрабатываемого исключения. Они должны располагаться *непосредственно* за **try**-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существуют три формата записи обработчиков исключений.

```
// формат 1
catch( тип имя )
{
    // ... Тело обработчика
}
```

```
// формат 2
```

```

catch( тип )
{
    // ... Тело обработчика
}

// Формат 3: обработчик умолчания
catch( ... )
{
    // ... Тело обработчика
}

```

Первый формат применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий. Чаще всего — это вывод информации об исключении. Еще раз напоминаем, что в обработчике, как и в генераторе исключений, можно использовать любой допустимый в языке тип, включая класс языка C++. Второй формат обработчика не предполагает использования информации об исключении — играет роль только тип, который используется для выбора подходящего обработчика. Многоточие в круглых скобках в заголовке обработчика (формат 3) обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в каком они записаны, то *обработчик третьего типа следует помещать после всех остальных обработчиков*.

Обратите внимание, что *после обработки исключения* управление передается первому оператору, находящемуся непосредственно за обработчиками исключений (конечно же, если в теле обработчика выполнение программы аварийно не завершено). Туда же, *минуя код всех обработчиков*, передается управление, если исключение в `try`-блоке не было сгенерировано.

6.3. Перехват исключений

Когда с помощью `throw` генерируется исключение, функции исполнительной системы C++ выполняют следующие действия:

1. Создают копию *выражения-параметра* из `throw` в виде статического объекта, который существует до тех пор, пока исключение не будет обработано.
2. В поисках подходящего обработчика исключения раскручивают стек. При раскрутке стека, в числе указанных в разд. 6.1 действий, вызывают деструкторы локальных объектов-классов, выходящих из области действия, и деструкторы копий аргументов-классов, передаваемых по значению.
3. Передают статический объект (копию *выражения-параметра* из `throw`) и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

При раскручивании стека все обработчики на каждом уровне просматриваются последовательно, от внутреннего блока к внешнему, пока не будет найден подходящий обработчик. Обработчик считается найденным, если тип *выражения*, указанного после `throw`:

- тот же, что и указанный в параметре `catch` (параметр может быть записан в форме `T`, `const T`, `T &` или `const T &`, где `T` — тип исключения);

- является производным от указанного в параметре `catch` (если наследование производилось с ключом доступа `public`);
- является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре `catch`.

Из сказанного следует, что обработчики производных классов следует размещать до обработчиков базовых классов, поскольку в ином случае им никогда не будет передано управление. Обработчик указателя типа `void` автоматически скрывает указатель любого другого типа. Поэтому его также следует размещать после обработчиков указателей конкретного типа.

Рассмотрим иллюстрирующий пример (листинги 6.1, 6.2).

Листинг 6.1. Файл EXEPTION.CPP

```
/*
    Демонстрация обработки исключений - в примере используется класс.
    Бруно Баёв [2] и Павловская Т.А. [9], консольное приложение (Microsoft Visual
    Studio C++ 6.0)
*/

#include <iostream>          // Ввод-вывод C++
#include <fstream>           // Для работы с файлами

using namespace            // Используем стандартное
    std;                   // пространство имен
// *****
// Класс, информирующий о своем создании и уничтожении
class Hello
{
    // Методы

public:

    // Конструктор: подставляемый метод
    Hello( void )
    {
        cout << "Hello!" << endl;
    }

    // Деструктор: подставляемый метод
    ~Hello( void )
    {
        cout << "Bye!" << endl;
    }
};

//*****
```

```

// Внешняя функция открывает файл и при ошибке генерирует исключение.
// Функция косвенно входит в контролируемый блок
void f1( void )
{
    // Открываем файл
    ifstream ifs( "\\INVALID\\FILE\\NAME" );
    if( !ifs )
    {
        cout << "Генерируем исключение" << endl;
        throw "Ошибка открытия файла\n\n";
    }

    return;
}

//*****
// Внешняя функция создает локальный объект-класс и вызывает предыдущую
// функцию. Функция явно входит в контролируемый блок
void f2( void )
{
    // Создаем локальный объект-класс
    Hello H;

    // Вызываем функцию, генерирующую исключение
    f1( );

    return;
}

//*****
// Тестирование обработки исключений
int main( void ) // Возвращает 0 при успехе
{
    // Контролируемый блок
    try
    {
        cout << "Входим в try-блок" << endl;
        f2( );
        cout << "Выходим из try-блока" << endl;
        // Отсюда выполняется переход на оператор return 0;
        // (все обошлось благополучно)
    }

    // Обрабатываем исключения типа char *
    catch( const char *s )
    {
        cout << endl << "Вызван обработчик const char *. " << endl << s;
    }
}

```

```

    return 10;           // Выполнение программы продолжается -
                        //   управление передается на оператор return 0;
                        //   (следует за обработчиками исключений)
}

// Обрабатываем все остальные исключения
catch( ... )
{
    cout << "Вызван обработчик остальных исключений"
        << endl << endl;

    return 20;          // Выполнение программы продолжается -
                        //   управление передается на оператор return 0;
                        //   (следует за обработчиками исключений)
}

return 0;               // Все обошлось благополучно
}

```

Листинг 6.2. Результаты выполнения программы

```

Входим в try-блок
Hello!
Генерируем исключение
Вые!

Вызван обработчик const char *.
Ошибка открытия файла

Press any key to continue

```

ПРИМЕЧАНИЕ

Обратите внимание, что после порождения исключения в результате раскрутки стека был вызван деструктор локального объекта, хотя управление из функции `fl()` было передано обработчику, находящемуся в функции `main`. Однако сообщение `Выходим из try-блока` не было выведено, так как после генерации исключения управление было передано на первый из обработчиков исключений.

Нетрудно заметить, что механизм исключений позволяет корректно уничтожить объекты при возникновении ошибочных ситуаций. Поэтому выделение и освобождение ресурсов полезно оформлять с использованием классов, конструкторы которых выделяют ресурсы, а деструкторы — освобождают. В качестве примера можно привести класс для работы с файлом. Конструктор класса открывает файл, а деструктор — закрывает. В этом случае есть гарантия, что при возникновении ошибки файл будет корректно закрыт и информация не будет утеряна.

Чтобы внести большую ясность в рассматриваемую тему, рассмотрим и проанализируем еще несколько примеров.

6.4. Примеры обработки исключений

Приводимые далее первые три примера (листинги 6.3—6.5) рассматривают решение одной и той же элементарной задачи, причем в первом примере обработка ошибок выполняется с использованием традиционных средств языка C, а в двух последующих примерах обработка ошибок выполняется путем обработки исключений. Такой набор примеров использован для того, чтобы показать связь между двумя названными способами обработки ошибок и облегчить применение исключений.

Листинг 6.3. Файл TASK1.CPP

```
/*
    Пример простой программы с обработкой ошибок средствами языка C.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <stdio.h>           // Для ввода-вывода

int main( void )           // Возвращает 0 при успехе
{
    int      a, b,          // Аргументы функции
             c,              // Значение функции
    // Возвращаемое значение для fscanf()
    retcode;

    FILE      *f_in,        // Указатель на структуру со сведениями о файле
               // для чтения
               *f_out;      // Указатель на структуру со сведениями о файле
               // для записи

    // Открываем файл для чтения
    if( ( f_in = fopen( "task1.dat", "r" ) ) == NULL )
    {
        printf( "\n Ошибка 10. Файл task1.dat для чтения не "
                "открыт \n\n" );
        return 10;
    }

    // Читаем значения аргументов функции
    retcode = fscanf( f_in, " %i %i", &a, &b );
    if( retcode != 2 )
    {
        printf( "\n Ошибка 20. Произошла ошибка чтения из "
                "файла task1.dat \n\n" );
        return 20;
    }
}
```

```

// Закрываем файл для чтения
if( fclose( f_in ) == EOF )
{
    printf( "\n Ошибка 30. Файл task1.dat не закрыт \n\n" );
    return 30;
}

// Открываем файл для записи
if( ( f_out = fopen( "task1.out", "w" ) ) == NULL )
{
    printf( "\n Ошибка 40. Файл task1.out для записи не "
           "открыт \n\n" );
    return 40;
}

// Печатаем заголовок и аргументы функции
fprintf( f_out,
"\n                c := a + b                                \n"
"\n                Аргументы функции: a=%i    b=%i \n", a, b );

// Моделируем ошибку закрытия файла для записи
fclose( f_out );
// Закрываем файл для записи
if( fclose( f_out ) == EOF )
{
    printf( "\n Ошибка 50. Файл task1.out не "
           "закрыт \n\n" );
    return 50;
}

// Вычисляем значение функции - в этом месте обычно делается довольно
// много работы: проверяется область допустимых значений
// прочитанных данных (аргументов функции) и выполняется решение
// задачи
c = a + b;

// Открываем файл для дозаписи
if( ( f_out = fopen( "task1.out", "a" ) ) == NULL )
{
    printf( "\n Ошибка 60. Файл task1.out для дозаписи"
           " не открыт \n\n" );
    return 60;
}

// Печатаем значение функции
fprintf( f_out,

```

```

"\n                Значение функции: c=%i", c );

    // Закрываем файл
    if( fclose( f_out ) == EOF )
    {
        printf( "\n Ошибка 70. Файл task1.out не закрыт \n\n" );
        return 70;
    }

    return 0;
}

```

Для данного примера файл ввода может иметь, например, следующий вид:

```
3 4
```

После запуска программы на выполнение был получен следующий экранный вывод (в приведенном ранее тексте тестируется ошибка закрытия файла записи) — далее приводится содержимое файла, в который был переназначен экранный вывод:

```
Ошибка 50. Файл task1.out не закрыт
```

```
Press any key to continue
```

Поэкспериментируйте с этим проектом — протестируйте ошибки открытия файлов в режимах чтения, записи, дозаписи, ошибку чтения и протестируйте работу программы в нормальном режиме. Этот пример тривиален и преследует единственную цель — продемонстрировать *традиционную* обработку ошибок средствами языка C с целью ее сопоставления с обработкой ошибок путем обработки исключений. Этот пример, как, впрочем, и все остальные примеры, демонстрирует один из возможных вариантов хорошего оформления исходного текста.

Два следующих примера демонстрируют обработку ошибок с использованием обработки исключений. Первый из них для ввода-вывода использует средства языка C (листинг 6.4), а второй — средства языка C++ (листинг 6.5).

Листинг 6.4. Файл EXEPTION1.CPP

```

/*
    Вычисление C := A / B. Демонстрация использования исключений для обработки оши-
    бок в стиле языка C++, ввод-вывод выполняется в стиле языка C).
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <sstream>           // Для строковых потоков
#include <string>            // Для строк
#include <iostream>         // Для потоков ввода-вывода

using namespace            // Пространство имен стандартной
    std;                  // библиотеки

```

```

ostreamstream os;           // Объект выходного строкового потока (подробнее
                             // о строковых потоках см. в третьей части
                             // данного учебного пособия)

int main(                   // Возвращает 0 при успехе
    int    ArgC,           // Число аргументов командной строки
    char   *ArgV[ ] ) // Массив указателей на аргументы командной
                       // строки
{
    // Контролируемый блок
    try
    {
        int    a, b,       // Аргументы функции
               c,          // Значение функции
               retcode;     // Возвращаемое значение для fscanf
        FILE    *f_in,     // Указатель на структуру со сведениями о файле
               // для чтения
               *f_out;      // Указатель на структуру со сведениями о файле
               // для записи

        // Проверяем число аргументов командной строки
        if( ArgC != 3 )
        {
            os << "Неверное количество аргументов командной строки. \n"
                << "Формат командной строки д.б.: \n"
                << "exception1.exe файл_ввода файл_вывода \n" << endl;
            // Генерация (порождение) исключения в месте выявления ошибки
            throw os.str( );
        }

        // Открываем файл для чтения
        if( ( f_in = fopen( ArgV[ 1 ], "r" ) ) == NULL )
        {
            os << "Файл " << ArgV[ 1 ] << " для чтения не открыт \n"
                << endl;
            throw os.str( );
        }

        // Читаем значения данных
        retcode = fscanf( f_in, " %i %i", &a, &b );
        if( retcode != 2 )
        {
            os << "Ошибка чтения данных из файла " << ArgV[ 1 ] << endl
                << endl;
            throw os.str( );
        }
    }
}

```

```

    }

    // Закрываем файл для чтения
    //fclose( f_in ); // Имитация ошибки
    if( fclose( f_in ) == EOF )
    {
        os << "Ошибка закрытия файла чтения " << ArgV[ 1 ] << endl
            << endl;
        throw os.str( );
    }

    // Открываем файл для записи
    if( ( f_out = fopen( ArgV[ 2 ], "w" ) ) == NULL )
    {
        os << "Ошибка открытия файла записи " << ArgV[ 2 ] << endl
            << endl;
        throw os.str( );
    }

    // Печатаем прочитанные данные
    fprintf( f_out,
        "\n          Аргументы функции:"
        "\n          -----"
        "\n          a=%i   b=%i", a, b );

    // Закрываем файл для записи
    fclose( f_out ); // Моделируем ошибку закрытия файла записи
    if( fclose( f_out ) == EOF )
    {
        os << "Ошибка закрытия файла записи " << ArgV[ 2 ] << endl
            << endl;
        throw os.str( );
    }

    // Вычисляем значение функции
    if( b == 0 )
    {
        os << "Попытка деления на 0, делитель b = " << b << endl
            << endl;
        throw os.str( );
    }
    c = a / b;

    // Открываем файл для дозаписи
    if( ( f_out = fopen( ArgV[ 2 ], "a" ) ) == NULL )
    {

```

```

        os << "Ошибка открытия файла " << Argv[ 2 ]
        << " для дозаписи \n" << endl;
        throw os.str( );
    }

    // Печатаем результаты работы программы
    fprintf( f_out,
"\n                Значение функции c=a/b=%i", c );

    // Закрываем файл для дозаписи
    //fclose( f_out ); // Моделирование ошибки
    if( fclose( f_out ) == EOF )
    {
        os << "Ошибка закрытия файла записи " << Argv[ 2 ] << endl
        << endl;
        throw os.str( );
    }
    } // Конец try-блока

// *****
// Обрабатываем исключения типа string
catch( string s )
{
    cout << "Вызван обработчик string." << endl << endl;
    cout << s.c_str( );

    return 10;
}

// Обрабатываем все остальные исключения
catch( ... )
{
    cout << "Вызван обработчик всех остальных исключений" << endl
    << endl;

    return 20;
}

return 0;
}

```

Для данного примера файл ввода (exception1.dat) может иметь, например, такой вид:

3 4

После запуска программы на выполнение с использованием командной строки вида
exception.exe exception1.dat exception1.out

был получен следующий экранный вывод (в приведенном ранее тексте также тестируется ошибка закрытия файла записи):

Вызван обработчик string.

Ошибка закрытия файла записи exception1.out

Press any key to continue

СОВЕТ

Не пожалейте времени — внимательно изучите эту программу и поэкспериментируйте с проектом: протестируйте ошибки открытия файлов в режимах чтения, записи, дозаписи, ошибку чтения, ошибку деления на 0 и протестируйте работу программы в нормальном режиме.

Анализ данного примера позволяет сделать следующие *выводы и замечания*.

- ❑ Контролируемый блок `try{ ... }` определяет фрагмент программного кода, в котором может генерироваться исключение (в данном примере контролируемый блок содержит практически всю программу, кроме оператора `return 0;`).
- ❑ Следом за контролируемым блоком размещаются обработчики исключений, сгенерированных в этом блоке.
- ❑ Если при работе программы ошибки не возникли, то при завершении работы контролируемого блока обработчики исключений пропускаются и управление передается на оператор, непосредственно следующий за последним обработчиком исключений (в данном примере таким оператором является `return 0;`).
- ❑ Если при работе программы возникла какая-либо ошибка (в данном случае это ошибка закрытия файла записи), то управление из точки генерации исключения `throw os.str();` передается обработчику исключения:

```
// *****
// Обрабатываем исключения типа string
catch( string s )
{
    cout << "Вызван обработчик string." << endl << endl;
    cout << s.c_str( );

    return 10;
}
```

Далее, после обработки исключения, выполнение программы в данном примере завершается после выполнения оператора `return 10;`. Если этот оператор удалить, то после обработки исключения управление будет передано оператору, непосредственно следующему за последним обработчиком исключений — в данном примере таким оператором был бы оператор `return 0;`.

- ❑ Данный пример попутно демонстрирует использование выходного строкового потока для передачи сообщения об ошибке обработчику исключения для печати этого сообщения. Обратите внимание на следующие фрагменты программного кода, относящиеся к этому:

```
#include <sstream>          // Для строковых потоков
#include <string>            // Для строк
```

```

#include <iostream>          // Для потоков ввода-вывода

using namespace             // Пространство имен стандартной
    std;                   // библиотеки

ostringstream os;          // Объект выходного строкового потока
                            // (подробнее о строковых потоках см. в
                            // третьей части данного учебного пособия)

...

// Проверяем число аргументов командной строки
if( ArgC != 3 )
{
    os << "Неверное количество аргументов командной "
        "строки. \n" << "Формат командной строки д.б.: \n"
        << "exeption1.exe файл_ввода файл_вывода \n" << endl;
    // Генерация (порождение) исключения в месте выявления
    // ошибки
    throw os.str( );
}

...

// *****
// Обрабатываем исключения типа string
catch( string s )
{
    cout << "Вызван обработчик string." << endl << endl;
    cout << s.c_str( );

    return 10;
}

...

```

Третий пример (листинг 6.5) отличается тем, что при файловом вводе-выводе используются средства языка C++.

Листинг 6.5. Файл EXEPTION2.CPP

```

/*
    Вычисление C := A / B. Демонстрация использования исключений для обработки оши-
    бок и ввода-вывода в стиле языка C++.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <sstream>          // Для строковых потоков

```

```
#include <string>           // Для строк
#include <iostream>         // Для потоков ввода-вывода
#include <fstream>          // Для работы с файлами

using namespace            // Используем стандартное
    std;                   // пространство имен

ostringstream os;         // Объект выходного строкового потока

int main(                  // Возвращает 0 при успехе
    int    ArgC,           // Число аргументов командной строки
    char   *ArgV[ ] )     // Массив указателей на аргументы командной
                        // строки
{
    // Контролируемый блок
    try
    {
        int    a, b,       // Аргументы функции
            c;             // Значение функции

        // Проверяем число аргументов командной строки
        if( ArgC != 3 )
        {
            os << "Неверное количество аргументов командной строки. \n"
                << "Формат командной строки д.б.: \n"
                << "exception2.exe файл_ввода файл_вывода \n\n\0";
            // Генерация (порождение) исключения
            throw os.str( );
        }

        // Открываем файл для чтения
        fstream
            f( ArgV[ 1 ], ios::in );
        if( !f )
        {
            os << "Файл " << ArgV[ 1 ] << " для чтения не открыт \n\n\0";
            throw os.str( );
        }

        // Читаем значения данных
        f >> a >> b;
        if( !f )
        {
            os << "Ошибка чтения данных из файла " << ArgV[ 1 ]
                << "\n\n\0";
        }
    }
}
```

```

        throw os.str( );
    }

    // Закрываем файл для чтения
    //f.close( );          // Имитация ошибки
    f.close( );
    if( !f )
    {
        os << "Ошибка закрытия файла чтения " << ArgV[ 1 ]
            << "\n\n0";
        throw os.str( );
    }

    // Открываем файл для записи
    f.open( ArgV[ 2 ], ios::out );
    if( !f )
    {
        os << "Файл " << ArgV[ 2 ] << " для записи не открыт \n\n0";
        throw os.str( );
    }

    // Печатаем прочитанные данные
    f <<
        "\n                Аргументы функции:"
        "\n                -----"
        "\n                a=" << a << "    b=" << b << "\n\n0";

    // Закрываем файл для записи
    f.close( );          // Моделируем ошибку закрытия файла
    f.close( );
    if( !f )
    {
        os << "Ошибка закрытия файла записи " << ArgV[ 2 ]
            << "\n\n0";
        throw os.str( );
    }

    // Вычисляем значение функции
    if( b == 0 )
    {
        os << "Попытка деления на 0, делитель b = " << b << "\n\n0";
        throw os.str( );
    }
    c = a / b;

    // Открываем файл для дозаписи
    f.open( ArgV[ 2 ], ios::out | ios::app );

```

```

    if( !f )
    {
        os << "Файл " << ArgV[ 2 ] << " для дозаписи не "
            "открыт \n\n\0";
        throw os.str( );
    }

    // Печатаем результаты работы программы
    f <<
"\n                Значение функции c=a/b=" << c << "\n\n\0";

    // Закрываем файл для дозаписи
    //f.close( );      // Моделируем ошибку
    f.close( );
    if( !f )
    {
        os << "Ошибка закрытия файла дозаписи " << ArgV[ 2 ]
            << "\n\n\0";
        throw os.str( );
    }
}

// Обрабатываем исключения типа string
catch( string s )
{
    cout << "Вызван обработчик string." << endl << endl;
    cout << s.c_str( );

    //return 10;      // Так можно продолжить работу программы после
                    // обработки исключения - по return 10; или
                    // exit( 10 ); работа программы остановится

    //exit( 10 );
}

// Обрабатываем все остальные исключения
catch( ... )
{
    cout << "Вызван обработчик всех остальных исключений \n" << endl
        << endl;

    return 20;
}

// В данном примере в эту точку мы попадаем при нормальном выполнении
// программы или при отсутствии в обработчиках исключений
// операторов return или вызовов функции exit( )

```

```
cout << "Выполнение программы продолжается ..." << endl << endl;  
  
return 0;  
}
```

Этот пример интересен в двух отношениях. Во-первых, в нем использован файловый ввод-вывод с использованием средств языка C++. Во-вторых, пример демонстрирует возможность продолжения выполнения программы после обработки исключения. На эти аспекты и рекомендуем обратить внимание в первую очередь.

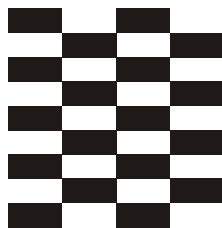
Подробную информацию об обработке исключений см. в [2, 4, 7, 9].

6.5. Вопросы для самопроверки [4]

1. Зачем тратить время на программирование исключений? Не лучше ли устранять ошибки по мере их возникновения?
2. Что такое исключение?
3. Для чего нужен блок `try`?
4. Для чего используется оператор `catch`?
5. Какую информацию может содержать исключение?
6. Когда создается объект исключения?
7. Что означает оператор `catch (...)`?

Для проверки правильности ответов можно воспользоваться *разд. П1.6 приложения 1*.

Глава 7



Динамическое определение типа и преобразование типов

При выполнении программы, написанной на языке C++, могут выполняться как явные, так и неявные преобразования типов. Правила выполнения неявных преобразований типов рассмотрены в [3, разд. 5.3, стр. 117—119]. Явные преобразования типов всегда выполняются по инициативе программиста, и для этого в языке C++ существуют следующие операции:

- операция приведения типа, унаследованная из языка C, и ее функциональная форма записи, введенная в язык C++;
- операции `const_cast`, `dynamic_cast`, `reinterpret_cast` и `static_cast`.

Следует заметить, что последняя группа операций включена в язык с целью обеспечения надежного преобразования типов.

Операция приведения типа в стиле языка C имеет следующий синтаксис [3]:

```
( тип ) выражение
```

В языке C++, наряду с этой операцией, унаследованной для обеспечения совместимости от языка C к языку C++, имеется еще один ее эквивалент, называемый *функциональной* формой записи преобразования типа:

```
тип ( выражение )
```

Результатом такой операции преобразования является значение заданного типа, что иллюстрируют листинги 7.1, 7.2.

Листинг 7.1. Файл TYPE_CAST_C.CPP

```
/*
    Преобразование типа в стиле языка C.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Для потокового ввода-вывода

using namespace
    std;                    // Стандартное пространство имен

int main( void )            // Возвращает 0 при успехе
```

```

{
    long int    li = 21;
    float      f = 12.21f;

    cout << "li = " << li << ", double( li ) = " << double( li ) << endl;
    cout << "f = " << f << ", ( int )f = " << ( int )f << endl << endl;

    return 0;
}

```

Листинг 7.2. Результаты выполнения программы, выводимые на экран

```

li = 21, double( li ) = 21
f = 12.21, ( int )f = 12

```

Press any key to continue

В этом примере значение объекта `li` приводится к типу `double`, а значение объекта `f` — к типу `int` с усечением дробной части. В обоих случаях кодовый формат представления результата операции приведения типа иной, чем кодовый формат исходных объектов.

Обратите внимание, что приведение типа в стиле языка C является источником возможных ошибок. Объясняется это тем, что приведение типа никаким образом не контролируется и вся ответственность за его результат возлагается на программиста (известно, что человеку свойственно ошибаться).

Для устранения этого недостатка в язык C++ введены более надежные операции преобразования типов `const_cast`, `dynamic_cast`, `reinterpret_cast` и `static_cast`, которые и следует использовать во всех необходимых случаях.

7.1. Операция *const_cast*

Операция преобразования типа `const_cast` применяется для того, чтобы аннулировать действие модификатора `const`:

```

const int    i;
int          *j = const_cast< int * > ( &i );

```

Необходимость преобразования в приведенном примере связана с тем, что при применении операции "&" к константному объекту результатом является константный указатель. Если требуется обычный указатель, то применяется операция `const_cast`.

В общем случае эта операция имеет следующий формат:

```
const_cast< T > ( v )
```

Обозначенный тип `T` должен быть таким же, как и тип `v` (обычно это указатель — см. пример ранее). Операция возвращает результат типа `T`.

Необходимость включения в язык операции `const_cast` обусловлена тем, что программист, проектирующий функцию, не обязан описывать не изменяемые в ней

параметры как `const`, хотя это и рекомендуется. Правила языка C++ запрещают передачу константного указателя на место обычного. Поэтому операция `const_cast` введена для того, чтобы обойти это ограничение. Естественно, что в подобном случае функция не должна изменять значение, на которое ссылается передаваемый указатель.

ЗАМЕЧАНИЕ

Если есть возможность добавить к описанию параметра функции модификатор `const`, то это предпочтительнее использования операции `const_cast` при вызове функции.

7.2. Преобразование типов во время компиляции (операция *`static_cast`*)

Операция преобразования типа `static_cast` выполняется компилятором и должна применяться только в том случае, когда без нее нельзя обойтись. Если же нужное преобразование типа достигается использованием стандартного (неявного) преобразования языка C++, то не следует использовать операцию `static_cast` только для того, чтобы избавиться от предупреждений транслятора. Предупреждения позволяют избежать возможных ошибок, а явное преобразование скрывает возможные ошибки.

В качестве примеров целесообразного применения операции `static_cast` можно назвать преобразование указателя `void*` к указателю на необходимый тип, приведение целого типа к типу перечисления или для других преобразований родственных типов.

Операция имеет следующий синтаксис:

```
static_cast< T > ( v )
```

Здесь `T` является типом ссылки, указателя, арифметическим или перечисляемым типом, а `v` — представителем ссылки, указателя, арифметического или перечисляемого типа. Результат операции имеет тип `T`.

Операцию `static_cast` можно применять для выполнения преобразований между целыми типами; целыми и вещественными типами; целыми и перечисляемыми типами; указателями и ссылками на объекты одной иерархии классов, при условии, что преобразование однозначно и не связано с нисходящим приведением типа виртуального базового класса. При выполнении этой операции кодовое представление может быть модифицировано.

Поясним сказанное иллюстрирующими примерами (листинги 7.3, 7.4).

Листинг 7.3. Файл `STATIC_CAST.CPP`

```
/*
    Статическое приведение типа с помощью static_cast.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Для потокового ввода-вывода
```

```
using namespace
    std;           // Стандартное пространство имен

int main( void )   // Возвращает 0 при успехе
{
    int           i = 21;

    // Приведение типа "целое <--> целое"
    cout << "int i = " << i << ", static_cast< long int > ( i ) = "
        << static_cast< long int > ( i ) << endl;

    // Приведение типа "вещественное <--> целое"
    float         f = 1.5f;
    cout << "float f = " << f << ", static_cast< int > ( f ) = "
        << static_cast< int > ( f ) << endl;

    // Приведение типа "перечисление <--> СИМВОЛЬНЫЙ"
    enum          Managers{ Ivanov, Petrov=48, Sidorov };
    Managers      Person = Petrov;
    cout << "Managers Person = " << Person <<
        ", static_cast< char > ( Person ) = "
        << static_cast< char > ( Person ) << endl << endl;

    return 0;
}
```

Листинг 7.4. Результаты работы программы, выводимые на экран

```
int i = 21, static_cast< long int > ( i ) = 21
float f = 1.5, static_cast< int > ( f ) = 1
Managers Person = 48, static_cast< char > ( Person ) = 0
```

Press any key to continue

Операция `static_cast` может также применяться для преобразования между указателями (ссылками) на базовый и производный классы одной иерархии, что иллюстрируют листинги 7.5, 7.6.

Листинг 7.5. Файл `STATIC_CAST_CLASS.CPP`

```
/*
    Статическое приведение типа с помощью static_cast для иерархии классов.
    Консольное приложение, Microsoft Visual Studio C++ 6.0
*/
```

```
#include <iostream>          // Для потокового ввода-вывода

using namespace
    std;                    // Стандартное пространство имен

// Определение базового класса
class Base
{
    // Методы

public:

    // Информация о классе
    void InfoBase( void )
    {
        cout << "Class Base" << endl;

        return;
    }
};

// Определение производного класса
class Derived : public Base
{
    // Методы

public:

    // Информация о классе
    void InfoDerived( void )
    {
        cout << "Class Derived" << endl;

        return;
    }
};

int main( void )            // Возвращает 0 при успехе
{
    Derived    d;           // Объект производного класса
    Base       b;           // Объект базового класса

    // Приведение типа для указателя "производный класс --> базовый"
    Base      *pb = static_cast< Base * > ( &d );
    pb->InfoBase( );
```

```
// Приведение типа для ссылки "базовый класс --> производный"
Derived    &rd = static_cast< Derived & > ( b );
rd.InfoDerived( );
cout << endl;

return 0;
}
```

Листинг 7.6. Результаты работы программы, выводимые на экран

```
Class Base
Class Derived

Press any key to continue
```

Еще раз напоминаем, что с помощью операции `static_cast` можно выполнять преобразования между указателями и ссылками на объекты иерархии классов при условии, что преобразование однозначно и не связано с нисходящим приведением типа для виртуального базового класса.

7.3. Преобразование типов во время выполнения программы (операция *dynamic_cast*)

Преобразование `dynamic_cast` позволяет производить безопасные, надежные преобразования типа, причем если преобразуются типы *полиморфных* объектов (полиморфный объект содержит хотя бы один виртуальный метод), то допустимость данного преобразования проверяется при выполнении программы. Преобразование `dynamic_cast` необходимо применять, когда правильность преобразования невозможно установить на этапе компиляции.

Операция имеет следующий синтаксис:

```
dynamic_cast< T > ( v )
```

Здесь `t` является типом ссылки, типом указателя на класс или `void*`, а `v` является ссылкой на базовый или производный класс, если `t` является типом ссылки, в противном случае `v` является указателем на базовый или производный класс. Результат операции имеет тип `t`, если преобразование выполнено успешно. При неудаче операция `dynamic_cast` генерирует исключение `bad_cast`, если `t` — ссылка, или возвращает нулевой указатель, если `t` — указатель. Если `t` и `v` не относятся к одной иерархии классов, то преобразование типа не допускается.

Преобразование из базового класса в производный называют *понижающим* (downcast), так как графически в иерархии наследования принято изображать производные классы ниже базовых. Преобразование из производного класса в базовый называют *повышающим* (upcast), а приведение между производными классами одного базового или между базовыми классами одного производного класса — *перекрестным* (crosscast).

ПРИМЕЧАНИЕ

Операцию `dynamic_cast` обычно применяют для полиморфных объектов и для понижающего преобразования виртуальных базовых классов. При этом операция использует механизм идентификации типа во время выполнения программы (Run-Time Type Information, RTTI). По умолчанию этот механизм отключен и его нужно включить. Например, в интегрированной среде разработки Microsoft Visual Studio C++ 6.0 для включения RTTI необходимо в настройках проекта (команда **Settings** меню **Project**) выбрать вкладку **C/C++**, далее категорию **C++ Language** и отметить флажком пункт **Enable Run-Time Type Information**. Кроме того, к программе необходимо подключить заголовочный файл `<typeinfo>`.

Повышающее преобразование. Следующий пример иллюстрирует использование операции `dynamic_cast` для повышающего преобразования (листинг 7.7).

Листинг 7.7. Файл UpCast.cpp

```
/*
    Повышающее динамическое приведение типа с помощью операции dynamic_cast для
    иерархии полиморфных классов. В этом программном проекте следует подключить RTTI.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Для потокового ввода-вывода
#include <typeinfo>          // Для RTTI

using namespace
    std;                  // Стандартное пространство имен

// Определение полиморфного базового класса
class Base
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base" << endl;

        return;
    }
};

// Определение полиморфного класса
class Base1
{
    // Методы
```

```

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base1" << endl;

        return;
    }
};

// Определение полиморфного производного класса
class Derived : public Base
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Derived" << endl;

        return;
    }
};

int main( void )           // Возвращает 0 при успехе
{
    // Объект полиморфного производного класса
    Derived    d;
    // Объект полиморфного базового класса
    Base       b;
    // Объект полиморфного класса
    Base1      b1;

    // Повышающее приведение типа для указателя "полиморфный производный
    // класс --> полиморфный базовый"
    Base       *pb = dynamic_cast< Base * > ( &d );
    if( pb )
        pb->Info( );
    // Некорректное приведение типа
    Base1      *pb1 = dynamic_cast< Base1 * > ( &d );
    if( !pb1 )

```

```

    cout << "Ошибка приведения типа - классы не входят в одну"
           " иерархию" << endl << endl;

// Повышающее приведение типа для ссылки "полиморфный производный
// класс --> полиморфный базовый"
try
{
    Base &br = dynamic_cast< Base & > ( d );
    br.Info( );
    // Некорректное приведение типа (классы не входят в одну
    // иерархию) - будет сгенерировано исключение
    Base1 &b1r = dynamic_cast< Base1 & > ( d );
    b1r.Info( );
}
catch( bad_cast )
{
    cout << "Обработка исключения bad_cast" << endl << endl;
}
catch( ... )
{
    cout << "Обработчик остальных исключений" << endl << endl;
}

return 0;
}

```

Листинг 7.8. Результаты работы программы, выводимые на экран

```

Class Derived
Ошибка приведения типа - классы не входят в одну иерархию

Class Derived
Обработчик остальных исключений

Press any key to continue

```

Понижающее преобразование. Чаще всего операция `dynamic_cast` применяется при понижающем преобразовании указателя/ссылки на базовый класс в указатель/ссылку на производный класс. В этом случае компилятор не имеет возможности проверить правильность приведения типа. Вместе с тем такую проверку все же можно выполнить с использованием RTTI, но при этом аргумент операции `dynamic_cast` должен быть полиморфного типа, то есть содержать хотя бы один виртуальный метод (при необходимости, такой метод можно ввести принудительно). Для полиморфного объекта операция `dynamic_cast` реализуется эффективно, так как информация о типе объекта заносится в таблицу виртуальных методов и доступ к ней осуществляется легко.

Результат операции `dynamic_cast` к указателю всегда требуется проверять явным образом, что иллюстрируют приводимые далее примеры (листинги 7.9, 7.10).

Листинг 7.9. Файл DownDynamicCast.cpp

```
/*
    Понижающее динамическое приведение типа указателя или ссылки с помощью операции
    dynamic_cast для иерархии полиморфных классов. В этом программном проекте следует
    подключить RTTI.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Для потокового ввода-вывода
#include <typeinfo>          // Для RTTI

using namespace
    std;                    // Стандартное пространство имен

// Определение полиморфного базового класса: RTTI поддерживается только
// для полиморфных типов
class Base
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base" << endl;

        return;
    }
};

// Определение полиморфного класса
class Base1
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base1" << endl;
```

```

        return;
    }
};

// Определение полиморфного производного класса
class Derived : public Base
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Derived" << endl;

        return;
    }
};

int main( void )           // Возвращает 0 при успехе
{
    // Объект полиморфного производного класса
    Derived    d;
    // Объект полиморфного базового класса размещается в динамической
    // памяти
    Base       *pb = new Base;
    // Указатель на объект полиморфного базового класса инициализирован
    // адресом объекта полиморфного производного класса
    Base       *pb1 = &d;
    // Объект полиморфного класса
    Base1      b1;

    // Понижающее приведение типа для указателя "полиморфный базовый
    // класс --> полиморфный производный" (для успеха указатель на
    // базовый класс должен иметь значение адреса объекта производного
    // класса)
    Derived    *pd = dynamic_cast< Derived * > ( pb1 );
    if( pd )
        pd->Info( );
    // Некорректные приведения типа
    pd = dynamic_cast< Derived * > ( &b1 );
    if( !pd )
        cout << "Ошибка приведения типа - классы не входят в одну"
            " иерархию" << endl;
}

```

```

if( typeid( *pb ) == typeid( Derived ) )
    pd = dynamic_cast< Derived * > ( pb );
else
    cout << "Ошибка приведения типа - аргумент операции dynamic_cast"
         << "\nне является адресом объекта Derived:\n"
         << "typeid( *pb ).name( ) = " << typeid( *pb ).name( )
         << ", \ntypeid( Derived ).name( ) = " <<
         << typeid( Derived ).name( ) << endl << endl;

// Понижающее приведение типа для ссылки "полиморфный базовый класс
// --> полиморфный производный"
try
{
    Derived    &dr = dynamic_cast< Derived & > ( d );
    dr.Info( );
    dr = dynamic_cast< Derived & > ( *pb1 );
    dr.Info( );
    // Некорректное приведение типа (классы не входят в одну
    // иерархию) - будет сгенерировано исключение
    dr = dynamic_cast< Derived & > ( b1 );
    dr.Info( );
    // Некорректное приведение типа (аргумент операции dynamic_cast
    // не является ссылкой на Derived) - будет сгенерировано
    // исключение
    //dr = dynamic_cast< Derived & > ( *pb );
    //dr.Info( );
}
catch( bad_cast )
{
    cout << "Обработка исключения bad_cast" << endl << endl;
}
catch( ... )
{
    cout << "Обработчик остальных исключений" << endl << endl;
}

return 0;
}

```

Листинг 7.10. Результаты работы программы, выводимые на экран

Class Derived

Ошибка приведения типа - классы не входят в одну иерархию

Ошибка приведения типа - аргумент операции dynamic_cast

не является адресом объекта Derived:

```
typeid( *pb ).name( ) = class Base,
typeid( Derived ).name( ) = class Derived
```

```
Class Derived
```

```
Class Derived
```

```
Обработчик остальных исключений
```

```
Press any key to continue
```

Отметим, что в рассмотренном примере использование вместо

```
Derived *pd = dynamic_cast< Derived * > ( pb1 );
```

понижающего приведения типа в стиле языка C

```
Derived *pd = ( Derived * )pb1;
```

сделало бы невозможным контроль допустимости операции. Если в этой ситуации указатель pb1 не будет указывать на объект класса Derived, то это приведет к ошибке.

Обратите внимание также на то, что данный пример демонстрирует применение операции typeid() для сравнения типов объектов и применение метода name() класса type_info для получения указателя на строку, содержащую имя типа объекта.

Другим недостатком приведения типа в стиле языка C является невозможность понижающего преобразования типа для виртуального базового класса — это в языке C++ запрещено синтаксически. С помощью операции **dynamic_cast** это становится возможным, если виртуальный базовый класс является полиморфным, а преобразование — однозначным. Сказанное иллюстрирует следующий пример (листинги 7.11, 7.12).

Листинг 7.11. Файл DownDynamicCastVirtBaseClass.cpp

```
/*
    Понижающее динамическое приведение типа указателя с помощью операции
    dynamic_cast для виртуального полиморфного базового класса. В проекте следует
    подключить RTTI.
*/
```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```
*/

#include <iostream>           // Для потокового ввода-вывода
#include <typeinfo>           // Для RTTI

using namespace
    std;                    // Стандартное пространство имен

// Определение полиморфного виртуального базового класса: RTTI
// поддерживается только для полиморфных типов
class Base
{
    // Методы
```

public:

```
// Информация о классе: виртуальный метод
virtual void Info( void )
{
    cout << "Class Base" << endl;

    return;
}

};

// Определение производного полиморфного класса для Base
class D1 : public virtual Base
{
    // Методы
```

public:

```
// Информация о классе: виртуальный метод
virtual void Info( void )
{
    cout << "Class D1" << endl;

    return;
}

};

// Определение полиморфного производного класса для Base
class D2 : public virtual Base
{
    // Методы
```

public:

```
// Информация о классе: виртуальный метод
virtual void Info( void )
{
    cout << "Class D2" << endl;

    return;
}

};

// Определение полиморфного производного класса для Base
class DD : public D1, public D2
{
```

```

// Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class DD" << endl;

        return;
    }
};

int main( void )          // Возвращает 0 при успехе
{
    // Объект самого нижнего полиморфного производного класса
    DD          dd;
    // Объект полиморфного виртуального базового класса размещается в
    // динамической памяти
    Base        *pb = new Base;
    // Указатель на полиморфный базовый класс инициализируется адресом
    // объекта полиморфного производного класса
    D1          *pd1 = &dd;

    // Понижающее приведение типа для указателя "полиморфный базовый
    // класс --> полиморфный производный" (для успеха указатель на
    // базовый класс должен иметь значение адреса объекта производного
    // класса)
    DD          *pd = dynamic_cast< DD * > ( pd1 );
    if( pd )
        pd->Info( );
    // Некорректное приведение типа
    pd = dynamic_cast< DD * > ( pb );
    if( pd )
        pd->Info( );
    else
        cout << "Ошибка приведения типа - аргумент операции dynamic_cast"
              << "\nне является адресом объекта DD:\n"
              << typeid( *pb ).name( ) = " << typeid( *pb ).name( )
              << ",\ntypeid( DD ).name( ) = " << typeid( DD ).name( )
              << endl << endl;

    return 0;
}

```

Листинг 7.12. Результаты работы программы, выводимые на экран

```

Class DD
Ошибка приведения типа - аргумент операции dynamic_cast
не является адресом объекта DD:
typeid( *pb ).name( ) = class Base,
typeid( DD ).name( ) = class DD

Press any key to continue

```

Перекрестное преобразование. Операция `dynamic_cast` позволяет, как показано далее (листинги 7.13, 7.14), выполнять безопасные преобразования между производными классами одного базового класса или между базовыми классами одного производного класса.

Листинг 7.13. Файл Cross_Cast.cpp

```

/*
    Перекрестное динамическое приведение типа указателя с помощью операции
    dynamic_cast. В проекте следует подключить RTTI.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Для потокового ввода-вывода
#include <typeinfo>          // Для RTTI

using namespace
    std;                    // Стандартное пространство имен

// Определение полиморфного базового класса: RTTI поддерживается только
// для полиморфных типов
class Base
{
    // Методы

public:
    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base" << endl;

        return;
    }
};

// Определение полиморфного базового класса: RTTI поддерживается только

```

```
// для полиморфных типов
class Base1
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base1" << endl;

        return;
    }
};

// Определение полиморфного производного класса для Base и Base1
class D : public Base, public Base1
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class D" << endl;

        return;
    }
};

int main( void )           // Возвращает 0 при успехе
{
    // Безопасное перекрестное преобразование типа между базовыми
    // полиморфными классами одного производного класса
    Base      *pBase = new D;
    Base1     *pBase1 = dynamic_cast< Base1 * > ( pBase );
    if( pBase1 )
        pBase->Info( );
    else
        cout << "Преобразование между базовыми полиморфными классами"
              << " одного\ппроизводного класса не выполнено" << endl << endl;

    return 0;
}
```

Листинг 7.14. Результаты работы программы, выводимые на экран

```
Class D
Press any key to continue
```

Операция `dynamic_cast` позволяет также, при необходимости, выполнять безопасное приведение типа между производными классами одного полиморфного базового класса (листинги 7.15, 7.16).

Листинг 7.15. Файл Cross_Cast1.cpp

```
/*
    Перекрестное динамическое приведение типа указателя с помощью операции
    dynamic_cast. В проекте следует подключить RTTI.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Для потокового ввода-вывода
#include <typeinfo>          // Для RTTI

using namespace
    std;                  // Стандартное пространство имен

// Определение полиморфного базового класса: RTTI поддерживается только
// для полиморфных типов
class Base
{
    // Методы

public:

    // Информация о классе: виртуальный метод
    virtual void Info( void )
    {
        cout << "Class Base" << endl;

        return;
    }
};

// Определение полиморфного производного класса для Base
class D1 : public Base
{
    // Методы
```

public:

// Информация о классе: виртуальный метод

void FunD1(**void**)

{

 cout << "Class D1" << endl;

return;

}

};

// Определение еще одного полиморфного производного класса для Base

class D : **public** Base

{

 // Методы

public:

// Информация о классе: виртуальный метод

void FunD(**void**)

{

 cout << "Class D" << endl;

return;

}

};

// Демонстрация перекрестного приведения типа между производными классами

// одного полиморфного базового класса

void demo(D *pD)

{

 D1 *pD1 = **dynamic_cast**< D1 * > (pD);

if(pD1)

 pD1->FunD1();

else

 cout << "Не выполнено" << endl;

return;

}

int main(**void**) // Возвращает 0 при успехе

{

 // Безопасное перекрестное преобразование типа между производными

```
// классами одного базового полиморфного класса
Base      *pBase = new D1;
// В следующем вызове функции аргумент приводится к типу D* для
// демонстрации возможности перекрестного преобразования между
// производными классами одного полиморфного базового класса
demo( ( D * )pBase );

return 0;
}
```

Листинг 7.16. Результаты работы программы, выводимые на экран

```
Class D1
Press any key to continue
```

В этой программе классы `D` и `D1` являются производными от класса `Base`. Функции `demo()` передается указатель на класс `D`, являющийся на самом деле указателем на "братский" класс `D1`. Поэтому динамическое перекрестное преобразование типа из `D` в `D1` завершается успешно.

Теперь на основании сравнительного анализа операций `static_cast` и `dynamic_cast` подведем некоторые итоги.

- ☐ Операция `dynamic_cast` использует RTTI. Поэтому, применяя эту операцию, убедитесь, что ее аргумент является полиморфным объектом и в компиляторе установлена опция, разрешающая использование RTTI.
- ☐ При понижающем преобразовании типа используйте операцию `dynamic_cast`.
- ☐ При преобразовании между указателями и ссылками на объекты в пределах иерархии классов в ряде случаев может применяться как операция `dynamic_cast`, так и операция `static_cast`. Применение операции `dynamic_cast` предпочтительнее по следующим причинам:
 - операция `dynamic_cast` надежнее, особенно при понижающих приведениях типа;
 - операция `dynamic_cast` не приводит к дополнительным расходам, если преобразование может быть безопасно произведено во время компиляции (в этом случае будет генерироваться тот же код, что и при использовании операции `static_cast`);
 - операция `dynamic_cast` допускает понижающее приведение виртуального базового класса;
 - операция `dynamic_cast` проверяет и производит перекрестное приведение типа.
- ☐ Операцию `static_cast` следует применять, если преобразование относится к указателям или ссылкам на не полиморфные объекты.

7.4. Преобразование "на свой страх и риск" (операция *reinterpret_cast*)

Преобразование с помощью операции `reinterpret_cast` применяется достаточно редко, когда нужно осуществить "экзотические" преобразования не связанных между собой типов. В результате такого преобразования получается значение нового типа, состоящее из той же цепочки двоичных разрядов, что и аргумент преобразования. Иными словами, операция `reinterpret_cast` используется для того, чтобы изменить точку зрения компилятора на тип объекта, при этом операция не модифицирует сам объект.

Операция имеет следующий синтаксис:

```
reinterpret_cast< T > ( v )
```

где `T` может быть типом ссылки, типом указателя, символьным, целым или вещественным типом, а `v` является представителем одного из только что перечисленных типов. Результат операции имеет тип `T`. Результат преобразования не контролируется и остается на совести программиста. Далее приводится пример использования операции `reinterpret_cast` для получения кодового представления числа с плавающей точкой (листинги 7.17, 7.18).

Листинг 7.17. Файл `Reinterpret_Cast.cpp`

```
/*
   Приведение типа с помощью операции reinterpret_cast (преобразование типа float
   в его кодовое представление).
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Для потокового ввода-вывода

using namespace
    std;                    // Стандартное пространство имен

// Преобразование типа float в его кодовое представление
//   (в шестнадцатеричный код)
void PrintDoubleToHex( float d )
{
    unsigned int
        *pui;
    pui = reinterpret_cast< unsigned int * > ( &d );
    cout << hex << *pui << endl;

    return;
}

int main( void )            // Возвращает 0 при успехе
```

```
{  
    // Преобразование float в шестнадцатеричный код  
    PrintDoubleToHex( 1.0f );  
    PrintDoubleToHex( 0.0f );  
  
    return 0;  
}
```

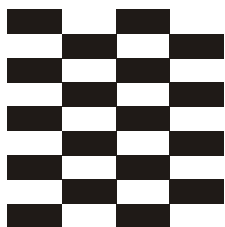
Листинг 7.18. Результат работы программы, выводимый на экран

```
3f800000  
0  
Press any key to continue
```

7.5. Вопросы для самопроверки

1. Для чего нужна операция преобразования типа `const_cast`?
2. Для чего нужна операция преобразования типа `static_cast`?
3. Для чего нужна операция преобразования типа `dynamic_cast`?
4. Для чего нужна операция преобразования типа `reinterpret_cast`?

Для проверки правильности ответов можно воспользоваться *разд. П1.6 приложения 1*.



Часть II

Прикладное программирование

Глава 8. Сортировка массивов

Глава 9. Транспортная задача (задача коммивояжера)

Глава 10. Поиск в таблице

Глава 11. Списки. Очереди и стеки

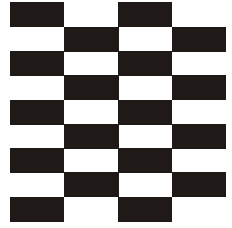
Глава 12. Сортировка файлов

Как указывалось в [3], в число классических задач *прикладного программирования* принято включать следующие задачи, составляющие золотой багаж любого программиста:

- ☐ сортировка (сортировка массивов, сортировка файлов);
- ☐ транспортная задача (задача коммивояжера);
- ☐ поиск в таблице;
- ☐ обработка списков;
- ☐ работа с очередями;
- ☐ численный анализ (вычислительная математика).

Численный анализ изучается в отдельном курсе, а остальные задачи прикладного программирования обсуждаются далее. Рассмотрение указанных задач представляет значительный интерес по следующим причинам. Во-первых, задачи прикладного программирования сами по себе имеют большое утилитарное значение и уже поэтому их изучение целесообразно. Во-вторых, решение этих задач с использованием технологии ООП является хорошим практикумом освоения объектно-ориентированного программирования. В третьих, для ряда задач прикладного программирования (сортировка массивов, транспортная задача, поиск в таблице) алгоритмическая реализация и программная реализация на основе технологии структурного и модульного программирования подробно рассмотрены в [3]. Поэтому при обсуждении этих задач можно целиком сосредоточить свое внимание на технологии ООП.

Перед рассмотрением задач прикладного программирования еще раз вернитесь к материалу *разд. 2.3 (рекомендации по составу класса; отличия структур и объединений от классов)*. Этот материал очень важен для проектирования объектно-ориентированных программ.



Глава 8

Сортировка массивов

Понятие сортировки, отличия сортировки массивов (внутренняя сортировка) от сортировки файлов (внешняя сортировка), алгоритмическая реализация всех методов сортировки массивов и их программная реализация на основе технологии структурного и модульного программирования рассмотрены в [3]. Воспользуемся указанными результатами для проектирования объектно-ориентированной программы сортировки массивов.

Первый вопрос, на который следует получить ответ, это определить, может ли и должна ли программа быть объектно-ориентированной или нет? В данном случае ответ "Да" очевиден. Почему? Объектом является массив, который целесообразно разместить в динамической памяти. Состояние объекта целиком и полностью определяется состоянием элементов массива. Для работы с объектом можно использовать интерфейс, содержащий следующий набор операций: размещение массива в динамической памяти, инициализация массива, просмотр (вывод) значений элементов массива, сортировка массива различными способами (простые и сложные сортировки выбором, обменом или вставками), освобождение занятой памяти. Следовательно, сортировку массива удобно реализовать с использованием класса или иерархии классов, к проектированию которых мы и перейдем сейчас.

8.1. Спецификация класса для сортировки массива

При проектировании спецификации класса необходимо решить следующие вопросы:

- ☐ выбрать целесообразную иерархию классов;
- ☐ определить, достаточным ли является использование обычных классов или следует применить шаблоны классов;
- ☐ спроектировать структуру каждого из классов иерархии (определить состав членов класса, их функциональное назначение и доступность);
- ☐ спроектировать файловую структуру класса (размещать ли целиком определения классов в заголовочных файлах или объявления классов помещать в заголовочные файлы, а реализацию методов классов — в файлы с расширением `cpp`).

Иерархия шаблонных или обычных классов. Поскольку сортировка массива является хотя и важной, но все же частной задачей, являющейся одним из этапов более сложной задачи, в которой требуется использовать отсортированный массив,

то представляется целесообразным реализовать сортировку массива в одном классе или в иерархии классов. Этот класс (классы) можно использовать в качестве базового (базовых) при решении задач, использующих отсортированные массивы.

Как указывалось ранее, при сортировке массивов используется следующий набор операций: размещение массива в динамической памяти, инициализация массива, просмотр (вывод) значений элементов массива, сортировка массива различными способами (простые и сложные сортировки выбором, обменом или вставками), освобождение занятой памяти. Анализируя этот набор операций, нетрудно заметить, что часть операций (размещение массива в динамической памяти, инициализация массива, просмотр или вывод значений элементов массива, освобождение занятой памяти) является типовой и используется не только при сортировке массива, но и при решении практически любых других задач обработки массивов.

Из сказанного следует, что в качестве вершины иерархии классов следует использовать базовый класс, в котором реализованы следующие типовые операции:

- ☐ размещение массива в динамической памяти (конструктор);
- ☐ инициализация массива значениями, читаемыми, например, из файла на магнитном диске;
- ☐ просмотр или вывод, например в файл, значений элементов массива;
- ☐ освобождение занятой массивом динамической памяти (деструктор).

Одним из следующих по иерархии нижележащих классов может быть класс, производный от базового класса, в котором реализованы известные методы сортировки массивов.

Таким образом, получаем следующую иерархию классов, которую можно использовать для сортировки массивов.

// Базовый класс: конструктор, деструктор, инициализация и вывод массива

```
class SArT_B
{
    ...
};
```

// Производный класс: различные методы сортировки массивов

```
class SArT_D : public SArT_B
{
    ...
};
```

Вполне очевидно, что элементы сортируемых массивов могут быть различного типа. Поэтому для сортировки массивов следует использовать иерархию шаблонных классов:

// Шаблон базовых классов: конструктор, деструктор, инициализация и вывод
// массива

```
template< class T >          // T - тип ключа сортировки элементов массива
class SArT_B
{
    ...
```

```
};

// Шаблон производных классов: различные методы сортировки массивов
template< class T >
class SArT_D : public SArT_B< T >
{
    ...
};
```

Структура классов иерархии. Структура классов иерархии определяется составом членов класса, их функциональным назначением и доступностью.

В целях общности в качестве сортируемого массива будем, в соответствии с [3], рассматривать *массив структур*, элементы которого имеют следующий тип:

```
struct ELEMENT    // Тип элемента массива
{
    T              key;        // Ключ сортировки
    // Описание других компонентов элемента
};
```

В шаблоне базовых классов следует иметь следующие данные, члены шаблона классов:

- указатель на начало массива структур в динамической памяти с типом `ELEMENT *`;
- размер массива структур.

Эти данные должны наследоваться в производные классы и их следует защитить спецификатором доступа **protected**:

В шаблоне базовых классов достаточно иметь только интерфейсные методы (**public**): конструктор, деструктор и методы для ввода и вывода значений элементов массива.

Таким образом, шаблон базовых классов может иметь следующий вид:

```
// Шаблон базовых классов для сортировки массивов
template< class T >
class SArT_B          // SortArrayTemplate_Base
{
    // Локальные типы

    struct ELEMENT    // Тип элемента массива
    {
        T            key;        // Ключ сортировки
        // Описание других компонент элемента
    };

    // Данные

protected:

    ELEMENT          *arr;        // Адрес первого элемента массива
```

```

int          size;      // Размер массива

// Методы

public:

    SArT_B( int s );      // Конструктор: s - размер массива

    ~SArT_B( void );      // Деструктор

    // Заполнение массива значениями, читаемыми из файла на магнитном
    // диске: f_name - файл ввода
    void inp_arr( char f_name[ ] );

    // Вывод содержимого массива в файл на магнитном диске: f_name - файл
    // вывода, mode - режим его открытия, text - текст для печати
    // заголовка
    void print_arr( char f_name[ ], int mode, char text[ ] );

};

```

В шаблоне производных классов нужно, прежде всего, иметь конструктор и интерфейсные методы (**public:**) для простой и сложной сортировки массива выбором, вставками и обменом. Единственным назначением конструктора шаблона производных классов является передача размера массива конструктору шаблона производных классов. Кроме того, в соответствии с [3], алгоритм сортировки массива сложным выбором с использованием двоичного дерева использует вспомогательную функцию `sift()` для "просеивания" дерева. Поэтому соответствующий метод шаблона производных классов должен быть оформлен как закрытый (**private:**). Аналогичным образом, сложная сортировка обменом (не рекурсивная сортировка Хоора) для хранения границ сегментов, подлежащих дальнейшему разбиению, использует стек отложенных сегментов со следующим типом элементов:

```

struct STACK      // Тип элемента стека
{
    int          l;      // Левая граница сегмента
    int          r;      // Правая граница сегмента
};

```

Для занесения границ сегментов в стек и извлечения их из стека сортировка Хоора использует две вспомогательных функции: `push()` и `pop()`. Поэтому соответствующие им методы шаблона производных классов должны быть закрытыми (**private:**).

Таким образом, шаблон производных классов может иметь следующую структуру:

```

// Размер стека отложенных сегментов: (log2(s)+1), где s - размер массива
const int        M = 16;

struct STACK      // Тип элемента стека
{

```

```

    int      l;          // Левая граница сегмента
    int      r;          // Правая граница сегмента
};

// *****
// Шаблон производных классов для сортировки массивов
template< class T >
class SArT_D : public SArT_B< T >
{
    // Методы

public:

    // Конструктор - обратите внимание на механизм передачи параметра из
    // конструктора производного класса конструктору базового класса
    SArT_D( int s ) : SArT_B< T >( s ){ }

    // Сортировка простыми включениями - по неубыванию
    void insertsrt( void );

    // Сортировка простым выбором - по неубыванию
    void selectsort( void );

    // Сортировка простым обменом - по неубыванию (метод "пузырька")
    void bubblesort( void );

    // Сортировка массива сложным выбором с использованием пирамиды -
    // двоичного дерева
    void treesort( void );

    // Сложная сортировка массива вставками (метод Шелла)
    void shellsort( void );

    // Быстрая сортировка массива - нерекурсивный вариант
    void quicksort( void );

private:

    // Занесение в стек сегментов (для сортировки Хоора)
    void push( int left, int right, STACK s[ M ], int &sp );

    // Извлечение сегмента из стека (для сортировки Хоора)
    void pop( int &l, int &r, STACK s[ M ], int &sp );

    // Просеивание (для сортировки с помощью двоичного дерева)

```

```
void sift( int root, int last );

};
```

Файловая структура иерархии шаблонных классов. Ранее было обосновано и показано (см. *разд. 1.6*), что определение шаблона классов должно целиком размещаться в заголовочном файле. Но поскольку базовый класс для работы с массивами может использоваться не только для сортировки массивов, но и при решении других задач обработки массивов, то шаблон базовых классов и шаблон производных классов должны размещаться в разных заголовочных файлах.

8.2. Сортировка массивов с использованием шаблонных классов

Теперь приведем полную программу сортировки массивов с использованием шаблонов классов, соответствующих спецификации, разработанной в предыдущем разделе (листинги 8.1—8.4). Эта программа использует класс `IOFILE`, рассмотренный в *разд. 5.6*. Данный класс позволяет удобно открывать и закрывать файлы и обеспечивает использование перегруженных операций "<<" и ">>" для вывода и ввода предопределенных типов. Определение класса `IOFILE` содержится в заголовочном файле `IOFile.h`, приведенном в *разд. 5.6*.

Листинг 8.1. Файл `SArT_B.H`

```
/*
Сортировка динамически размещенного массива.
Используется шаблон базовых классов со следующим набором операций:
- динамическое размещение массива с инициализацией (конструктор);
- освобождение занятой динамической памяти (деструктор);
- заполнение массива;
- печать содержимого массива.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предотвращение многократного включения данного файла
#ifndef __SArT_B_H
#define __SArT_B_H

// Определение класса для открытия-закрытия файлов на базе
// стандартного класса fstream
#include "iofile.h"

// *****
// Шаблон базовых классов для сортировки массивов
template< class T >
class SArT_B
{
```

```

// Локальные типы

struct ELEMENT    // Тип элемента массива
{
    T key;         // Ключ сортировки
    // Описание других компонентов элемента
};

// Данные

protected:

ELEMENT
    *arr;         // Адрес первого элемента массива
int    size;     // Размер массива

// Методы

public:

SArT_B( int s ); // Конструктор

// Деструктор - подставляемый метод
~SArT_B( void )
{
    if( arr )
    {
        delete [ ] arr; arr = NULL;
    }
}

// Заполнение массива значениями, читаемыми из файла на магнитном
// диске
void inp_arr( char f_name[ ] );

// Вывод содержимого массива в файл на магнитном диске
void print_arr( char f_name[ ], int mode, char text[ ] );

};

// *****
// Конструктор
template< class T >
SArT_B< T > :: SArT_B(
    int    s )    // Число элементов массива
{

```

```

    if( s < 2 )
    {
        cout << "\n In the array there should be 2 or more units "
              << endl;
        exit( 1 );
    }

    arr = new ELEMENT[ s ];
    if( !arr )
    {
        cout << "\n Error of allocation of the array in dynamic "
              "memory " << endl;
        exit( 2 );
    }

    for( int i=0; i<s; i++ )
    {
        arr[ i ].key = 0;
    }

    size = s;
}

// *****
// Заполнение массива значениями, читаемыми из файла на магнитном
// диске
template< class T >
void SArT_B< T > :: inp_arr(
    // Файл ввода
    char f_name[ ] )
{
    IOFILE f_in;      // Файловый объект для ввода

    // Открытие файла для чтения
    f_in.open_f( f_name, ios::in, 3 );

    // Заполнение массива
    for( int i=0; i<size; i++ )
    {
        f_in >> arr[ i ].key;
        // Обработка ошибки чтения
        if( !f_in )
        {
            cout << endl << " Read error";
            exit( 4 );
        }
    }
}

```

```

    }

    // Закрытие файла ввода
    f_in.close_f( f_name, 5 );

    return;
}

// *****
// Вывод содержимого массива в файл на магнитном диске
template< class T >
void SArT_B< T > :: print_arr(
    char    f_name[ ], // Файл вывода
    int     mode,      // Режим открытия файла
    char    text[ ] ) // Заголовок для печати
{
    IOFILE f_out;      // Файловый объект для вывода

    // Открытие файла для записи
    f_out.open_f( f_name, mode, 6 );

    f_out << endl << text;

    for( int i=0; i < size; i++ )
    {
        if( !( i%4 ) )
            f_out << endl;
        f_out.width( 14 ); f_out << arr[ i ].key;
    }

    // Закрытие файла вывода
    f_out.close_f( f_name, 7 );

    return;
}
#endif

```

Листинг 8.2. Файл SArT_D.H

```

/*
    Сортировка динамически размещенного массива.

    Используется шаблон производных классов от шаблона базовых классов,
    определенного в файле SArT_B.h, со следующим набором операций:
    - простая сортировка массива вставками;
    - сортировка простым выбором;
    - сортировка простым обменом - по неубыванию (метод "пузырька");

```

- сортировка массива сложным выбором с использованием пирамиды - двоичного дерева;
- сложная сортировка массива вставками (метод Шелла);
- нерекурсивная сортировка Хоора.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/

// Предотвращение многократного включения данного файла
#ifndef __SArT_D_H
#define __SArT_D_H

    // Шаблон базовых классов для сортировки массива
    #include "SArT_B.h"

    // Размер стека отложенных сегментов: (log2(s)+1), где s - размер
    // массива
    const int M = 16;

    struct STACK          // Тип элемента стека
    {
        int      l;       // Левая граница сегмента
        int      r;       // Правая граница сегмента
    };

    // *****
    // Шаблон производных классов для сортировки массивов
    template< class T >
    class SArT_D : public SArT_B< T >
    {
        // Методы

    public:

        // Конструктор - обратите внимание на механизм передачи параметра
        // из конструктора производного класса конструктору базового
        // класса
        SArT_D( int s ) : SArT_B< T >( s ){ }

        // Сортировка простыми включениями - по неубыванию
        void insertsort( void );

        // Сортировка простым выбором - по неубыванию
        void selectsort( void );

        // Сортировка простым обменом - по неубыванию (метод "пузырька")
        void bubblesort( void );

        // Сортировка массива сложным выбором с использованием пирамиды -

```

```

// двоичного дерева
void treesort( void );

// Сложная сортировка массива вставками (метод Шелла)
void shellsort( void );

// Быстрая сортировка массива - нерекурсивный вариант
void quicksort( void );

private:

// Занесение в стек сегментов (для сортировки Хоора)
void push( int left, int right, STACK s[ M ], int &sp );

// Извлечение сегмента из стека (для сортировки Хоора)
void pop( int &l, int &r, STACK s[ M ], int &sp );

// Просеивание (для сортировки с помощью двоичного дерева)
void sift( int root, int last );

}; // Конец определения класса

// *****
// Сортировка простыми включениями - по неубыванию: сортируются
// элементы массива с индексами от 1 до size-1, элемент с индексом
// 0 используется в качестве вспомогательного (левый "барьер")
template< class T >
void SAR_T_D< T > :: insertsort( void )
{
    int    i,          // Индекс вставляемого элемента
           j;          // Индекс элемента в упорядоченном сегменте
    ELEMENT
        copy;          // copy = arr[ i ]

    // Перебор вставляемых элементов
    for( i=2; i < size; i++ )
    {
        copy = arr[ i ];
        // Установка "барьера"
        arr[ 0 ] = copy;

        // Вставка copy на нужное место
        j = i -1;
        while( copy.key < arr[ j ].key )
        {
            // Сдвиг
            arr[ j+1 ] = arr[ j ]; j--;

```

```

    }
    arr[ j+1 ] = copy;
}

return;
}

// *****
// Сортировка массива простым выбором - по неубыванию
template< class T >
void SArT_D< T > :: selectsort( void )
{
    int    l,          // Индекс последнего элемента из пока
                //      неупорядоченных
    indmax, // Индекс наибольшего элемента
    k;      // Индекс анализируемого элемента
    ELEMENT
    max;     // Для наибольшего элемента среди элементов
                //      с индексами 0..l

    // Перебираем границы неотсортированной части массива
    for( l = size-1; l >= 1; l-- )
    {
        // Присвоить indmax индекс элемента массива с наибольшим
        // значением ключа из arr[ 0 ], ..., arr[ l ]
        indmax = 0; max = arr[ 0 ];
        for( k=1; k<=l; k++)
        {
            if( arr[ k ].key > max.key )
            {
                indmax = k; max = arr[ k ];
            }
        }

        // Поменять местами arr[ indmax ] и arr[ l ]
        arr[ indmax ] = arr[ l ]; arr[ l ] = max;
    }

    return;
}

// *****
// Сортировка простым обменом - по неубыванию (метод "пузырька")
template< class T >
void SArT_D< T > :: bubblesort( void )
{
    int    k;          // Индекс анализируемого элемента

```

```

ELEMENT
    temp;        // Для перестановки в "arr"
int    sorted,   // !=0 => отсортирован
        change,   // !=0 => были перестановки
        i = 0;    // Счетчик проходов

sorted = 0;
// Цикл проходов
while( !sorted )
{
    change = 0;
    // Цикл перебора пар
    for( k = 1; k < size-i; k++ )
    {
        if( arr[ k-1 ].key > arr[ k ].key )
        {
            temp = arr[ k ];
            arr[ k ] = arr[ k-1 ];
            arr[ k-1 ] = temp; change = 1;
        }
    }
    sorted = !change; i++;
}

return;
}

// *****
// Сортировка массива сложным выбором с использованием пирамиды -
// двоичного дерева
// -----
// Просеивание
template< class T >
void SArT_D< T > :: sift(
    int    root,      // Корень дерева или поддеревя
    int    last )     // Последняя вершина в дереве
{
    int    i,          // Позиция "дырки"
        j1,           // j1 = 2*i -- следующая вершина снизу и слева
                    // для i
        j2,           // j2 = 2*i+1 - следующая вершина снизу и справа
                    // для i
        j;            // Претендент из j1 и j2 на заполнение "дыры"
ELEMENT
    сору;            // Просеиваемый элемент
int    found;       // 1 => нашли место для вставки сору

```

```

// Подготовка
copy = arr[ root-1 ]; i = root; found = 0;
// Просеивание
while( !found )
{
    // Определение j1 и j2 для зафиксированного i
    j1 = 2 * i; j2 = j1 + 1;
    // Анализ вариантов заполнения "дыры"
    if( j1 > last )
    {
        // Следующего уровня вниз нет
        found = 1;
    }
    else
    {
        // Следующий вниз уровень есть
        if( j1 == last )
        {
            j = j1;
        }
        else
        {
            j = arr[ j1-1 ].key >= arr[ j2-1 ].key ? j1 : j2;
        }
        // Выяснить, кто заполняет "дыру"
        if( arr[ j-1 ].key <= copy.key )
        {
            found = 1;
        }
        else
        {
            arr[ i-1 ] = arr[ j-1 ]; i = j;
        }
    }
}

arr[ i-1 ] = copy;

return;
}
// -----
// Сортировка массива сложным выбором с использованием пирамиды -
// двоичного дерева
template< class T >
void SArT_D< T > :: treesort( void )
{
    int    temproot, // Индекс корня частичного поддерева
           templast; // Последний элемент в неупорядоченном поддереве

```

```

ELEMENT
    tempcopy; // Для перестановок в "arr"

// Подготовка дерева
for( temproot = size/2; temproot > 0; temproot-- )
{
    sift( temproot, size );
}

// Сортировка по дереву
for( templast = size; templast >= 2; templast-- )
{
    // Переставить максимум из корня дерева на окончательное
    // место
    tempcopy = arr[ 0 ];
    arr[ 0 ] = arr[ templast-1 ];
    arr[ templast-1 ] = tempcopy;
    // Просеять новый корень на место - восстановить дерево
    sift( 1, templast-1 );
}

return;
}

// *****
// Сложная сортировка массива вставками (метод Шелла)
template< class T >
void SarT_D< T > :: shellsort( void )
{
    int    d,          // Дистанция Шелла
           fillpos,    // Местоположение "дыры"
           i,          // Индекс анализируемого элемента в "arr"
           j;          // Индекс претендента слева на заполнение "дыры"
    ELEMENT
    copy;    // copy = arr[ i ]
    int    found;    // 1 => нашли место для вставки copy

    d = size;
    while( d > 1 )
    {
        d = d/2;
        // Отсортировать вставками при текущем d
        for( i = d; i < size; i++ )
        {
            copy = arr[ i ]; fillpos = i;
            // Найти место вставки copy

```

```

        found = 0;
    do
    {
        j = fillpos - d;
        if( j<0 )
        { // Претендента слева нет
            found = 1;
        }
        else
        { // Претендент слева больше - сдвиг
            if( arr[ j ].key <= copy.key )
            {
                found = 1;
            }
            else
            {
                arr[ fillpos ] = arr[ j ];
                fillpos = j;
            }
        }
    }
    while( !found );

    // Вставка copy
    arr[fillpos] = copy;
}

}

return;
}

// *****
// Быстрая сортировка Хоора - нерекурсивный вариант
// -----
// Занесение в стек сегментов
template< class T >
void SArT_D< T > :: push(
    int    left,    // Левая граница сегмента
    int    right,   // Правая граница сегмента
    STACK  s[ M ],  // Стек границ сегментов
    int    &sp )    // sp - указатель вершины стека
{
    if( ( right-left) >= 1 )
    {
        sp++; s[ sp ].l = left; s[ sp ].r = right;
    }
}

```

```

    return;
}
// -----
// Извлечение сегмента из стека
template< class T >
void SArT_D< T > :: pop(
    int    &l,        // Указатель на левую границу сегмента
    int    &r,        // Указатель на правую границу сегмента
    STACK  s[ M ],    // Стек границ сегментов
    int    &sp )      // sp - указатель вершины стека
{
    l = s[ sp ].l; r = s[ sp ].r; sp--;

    return;
}
// -----
// Быстрая сортировка массива - нерекурсивный вариант
template< class T >
void SArT_D< T > :: quicksort( void )
{
    int    left,      // Левая граница разделяемого сегмента
           right,     // Правая граница разделяемого сегмента
           i,         // Индекс кандидата на обмен слева - направо
           j;         // Индекс кандидата на обмен справа - налево
    ELEMENT
           median,    // Медиана разделяемого сегмента
           copy;      // Для перестановки кандидатов
    STACK  s[ M ];    // Стек границ сегментов
    int    sp;        // Указатель вершины стека

    sp = -1;          // Стек пуст
    // sp >= 0 => стек не пуст
    push( 0, size-1, s, sp );

    while( sp >= 0 )
    {
        // Подготовка верхнего сегмента из стека для деления
        pop( left, right, s, sp );
        median = arr[ ( left + right )/2 ];
        i = left; j = right;
        // Разделение текущего сегмента
        while( i <= j )
        {
            // Найти кандидата на обмен слева
            while( arr[ i ].key < median.key )
                i++;

```

```

        // Найти кандидата на обмен справа
        while( median.key < arr[ j ].key )
            j--;
        // Обмен, если кандидаты находятся в разных подсегментах
        if( i <= j )
        {
            copy = arr[ i ]; arr[ i ] = arr[ j ];
            arr[ j ] = copy; i++; j--;
        }
    }

    // Поместить в стек сначала более длинный подсегмент, а затем -
    // более короткий
    if( ( j-left ) < ( right-i ) )
    {
        // Левый подсегмент - короче
        push( i, right, s, sp );
        push( left, j, s, sp );
    }
    else
    {
        // Правый подсегмент - короче
        push( left, j, s, sp );
        push( i, right, s, sp );
    }
}

return;
}

#endif

```

Листинг 8.3. Файл SArT.CPP

```

/*
    Сортировка массивов с использованием шаблонов базовых SArT_V и производных
    SArT_D классов, определение которых приведено во включаемых файлах SArT_V.H и
    SArT_D.H. Для открытия-закрытия файлов в шаблоне базовых классов SArT_V использует-
    ся класс IOFILE, определение которого приведено во включаемом файле IOFILE.H.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6
*/

#include "SArT_D.h"          // Класс для сортировки массивов

// *****

// Тестирование
int main( void )            // Возвращает 0 при успехе
{
    SArT_D< int >
        ar( 8 ); // Создаем массив из 8 элементов

```

```
// Заполняем сортируемый массив
ar.inp_arr( "sort_arr.dat" );
// Печатаем сортируемый массив
ar.print_arr( "sort_arr.out", ios::out,
              "\nThe array before sorting:" );
ar.insertsort( ); // Сортируем массив простыми включениями
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after sorting by simple inclusions"
              "\n (first unit - auxiliary, is not sorted):");

// Заполняем сортируемый массив
ar.inp_arr( "sort_arr.dat" );
// Печатаем сортируемый массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              "\nThe array before sorting:" );
ar.selectsort( ); // Сортируем массив простым выбором
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after sorting by simple choice:");

// Заполняем сортируемый массив
ar.inp_arr( "sort_arr.dat" );
// Печатаем сортируемый массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              "\nThe array before sorting:" );
ar.bubblesort( ); // Сортируем массив простым обменом
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after sorting by simple exchange:");

// Заполняем сортируемый массив
ar.inp_arr( "sort_arr.dat" );
// Печатаем сортируемый массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              "\nThe array before sorting:" );
ar.treesort( ); // Сортируем массив с помощью двоичного дерева
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after sorting with the help"
              " of a binary tree:");

// Заполняем сортируемый массив
ar.inp_arr( "sort_arr.dat" );
// Печатаем сортируемый массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              "\nThe array before sorting:" );
```

```

// Сложная сортировка вставками (сортировка Шелла)
ar.shellsort( );
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after sorting Shell:" );

// Заполняем сортируемый массив
ar.inp_arr( "sort_arr.dat" );
// Печатаем сортируемый массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              "\nThe array before sorting:" );
// Нерекурсивная быстрая сортировка Хоора
ar.quicksort( );
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after fast not of recursive"
              " sorting Hoora:" );

return 0;
}

```

Листинг 8.4. Результаты работы программы, выводимые в файл на магнитном диске

The array before sorting:

1	8	2	7
3	6	4	5

The array after sorting by simple inclusions
(first unit - auxiliary, is not sorted):

5	2	3	4
5	6	7	8

The array before sorting:

1	8	2	7
3	6	4	5

The array after sorting by simple choice:

1	2	3	4
5	6	7	8

The array before sorting:

1	8	2	7
3	6	4	5

The array after sorting by simple exchange:

1	2	3	4
5	6	7	8

The array before sorting:

1	8	2	7
3	6	4	5

The array after sorting with the help of a binary tree:

1	2	3	4
5	6	7	8

The array before sorting:

1	8	2	7
3	6	4	5

The array after sorting Shell:

1	2	3	4
5	6	7	8

The array before sorting:

1	8	2	7
3	6	4	5

The array after fast not of recursive sorting Hoор:

1	2	3	4
5	6	7	8

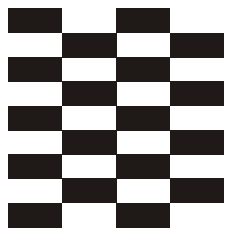
Внимательно изучите эту программу. Она является весьма полезной и познавательной.

8.3. Упражнения для самопроверки

1. Программу из *разд. 8.2* в учебных целях модифицируйте таким образом, чтобы в ней использовались не шаблоны классов, а обычные классы. Считайте, что ключ сортировки имеет тип `double`. Для сокращения объема программы в производном классе оставьте только лучшую из простых сортировок (вставками) и лучшую из сложных сортировок (Хоора).
2. Программу из упражнения 1 модифицируйте таким образом, чтобы объявления классов размещались в заголовочных файлах, а реализация методов классов — в файлах с расширением `cpp`. Сказанное не относится к классу `IOFILE`.
3. Программу из *разд. 8.2* в учебных целях модифицируйте таким образом, чтобы в ней использовались не шаблоны классов, а обычные классы. Считайте, что ключ сортировки имеет тип `long`. Для сокращения объема программы в производном классе оставьте только лучшую из простых сортировок (вставками) и лучшую из сложных сортировок (Хоора). Нерекursивный вариант сортировки Хоора замените рекурсивным, подробно описанным и реализованным в [3].

Ответы на эти упражнения приведены в *разд. П1.7 приложения 1*.

Глава 9



Транспортная задача (задача коммивояжера)

Большая часть наших рассуждений, относящихся к спецификации иерархии классов для сортировки массивов, справедлива и для решения транспортной задачи. Прежде чем перейти к проектированию объектно-ориентированной программы для решения транспортной задачи, внимательно рассмотрите учебный материал, приведенный в [3]: терминологию, относящуюся к графам, формы задания графа и рекурсивный алгоритм решения транспортной задачи, реализованный на основе технологии структурного и модульного программирования. Как и в предыдущем разделе, воспользуемся указанными результатами для проектирования объектно-ориентированной программы решения транспортной задачи.

Вновь возникает вопрос: "Может ли и должна ли программа быть объектно-ориентированной или нет?" И в данном случае ответ "Да" очевиден. Почему? Объектом является граф, информацию о котором целесообразно разместить в динамической памяти. Свойства графа целиком определяются этой информацией. Граф, как объект, может представлять собой конкретные различные объекты реального мира. Например, вершины графа могут представлять города, а дуги — соединяющие их дороги. Для этого примера можно говорить о поиске оптимального пути между заданными городами, то есть о решении транспортной задачи (задачи коммивояжера). Для работы с такого рода объектом-графом можно использовать интерфейс, содержащий следующий набор операций: размещение информации о графе в динамической памяти, инициализация данных о графе, просмотр (вывод) значений данных о графе, решение транспортной задачи с использованием рекурсивного подхода, вывод сведений о полученном решении и освобождение памяти, занятой информацией о графе. Следовательно, решение транспортной задачи удобно реализовать с использованием класса или иерархии классов, к проектированию которых мы и перейдем сейчас.

9.1. Спецификация класса для решения транспортной задачи

Иерархия шаблонных или обычных классов. Как только что указывалось, при решении транспортной задачи можно использовать следующий интерфейсный набор операций: размещение информации о графе в динамической памяти, инициализация данных о графе, просмотр (вывод) значений данных о графе, решение транспортной задачи с использованием рекурсивного подхода, вывод сведений о полученном решении и освобождение памяти, занятой информацией о графе. Анализируя этот

набор операций, нетрудно заметить, что часть перечисленных операций (размещение информации о графе в динамической памяти, инициализация данных о графе, вывод значений данных о графе и освобождение памяти, занятой информацией о графе) является типовой и используется не только при решении транспортной задачи, но и при решении практически любых других задач обработки графов.

По этой причине в качестве вершины иерархии классов следует использовать базовый класс, в котором реализованы следующие типовые операции:

- размещение информации о графе в динамической памяти (конструктор);
- инициализация данных о графе значениями, читаемыми, например, из файла на магнитном диске;
- просмотр или вывод, например в файл, значений данных о графе;
- освобождение занятой данными о графе динамической памяти (деструктор).

Одним из возможных, следующих по иерархии нижележащих классов может быть класс, производный от базового класса, в котором реализовано решение транспортной задачи.

Таким образом, получаем следующую иерархию классов, которую можно использовать для решения транспортной задачи:

```
// Базовый класс: конструктор, деструктор, инициализация и вывод данных
// о графе
class GR
{
    ...
};

// Производный класс: решение транспортной задачи
class GR_CT : public GR
{
    ...
};
```

Вполне очевидно, что при задании информации о весах дуг графа, соединяющих вершины, вполне достаточным является использование любого из вещественных типов. Поэтому для решения транспортной задачи и других задач обработки графов, в отличие от сортировки массивов, достаточно использовать иерархию обычных, не шаблонных классов.

Структура классов иерархии определяется составом членов класса, их функциональным назначением и доступностью.

В соответствии с [3], наиболее распространенным способом задания информации о графе является ее задание в виде списка ребер (рис. 9.1):

```
struct A                // Arc: дуга (ребро) графа
{
    int      first;      // 1-я вершина ребра
    int      last;       // 2-я вершина ребра
    float    weight;     // Вес ребра
};
```

```
};

// Адрес первого элемента массива структур с информацией о ребрах графа
А *pArc;
```

Этот способ обеспечивает экономию памяти и является более алгоритмичным. Последнее означает, что алгоритмы решения задач с использованием графов, заданных списком ребер, проще и эффективнее.

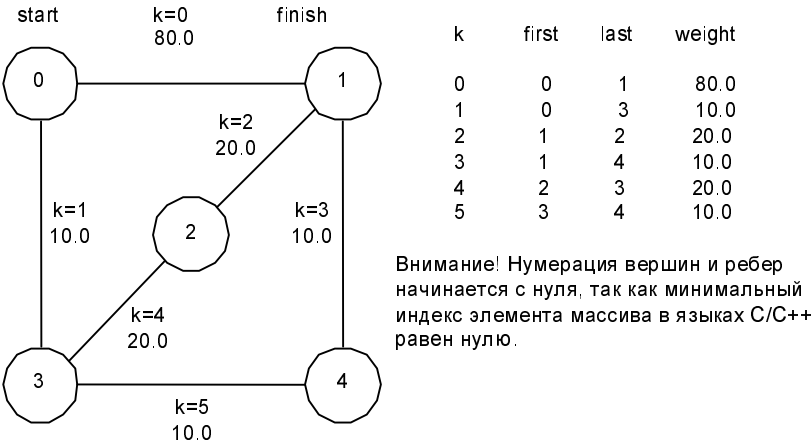


Рис. 9.1. Пример графа, заданного списком ребер

В качестве данных, членов класса обычно выбирают данные, которые используются в нескольких методах рассматриваемого или производного от него классов. В соответствии со сказанным, в *базовом классе* следует иметь следующие данные:

- указатель на начало массива структур в динамической памяти, содержащего информацию о ребрах графа с типом `А *`;
- число вершин (`NumTop`) и число ребер (`NumArc`) графа;
- вершина — старт пути (`start`);
- вершина — финиш пути (`finish`).

Все эти данные должны наследоваться в производные классы и их следует защитить спецификатором доступа `protected::`. Кроме того, данные `А *pArc`, `NumTop` и `NumArc` используются еще и в базовом классе.

В базовом классе достаточно иметь только интерфейсные методы (`public::`): конструктор, деструктор и методы для ввода и вывода значений данных о графе.

Таким образом, базовый класс может иметь следующий вид:

```
struct А // Arc: дуга (ребро) графа
{
    int first; // 1-я вершина ребра
    int last; // 2-я вершина ребра
    float weight; // Вес ребра
}
```

```

};

// *****
// Базовый класс для объекта графа
class GR
{
    // Данные

protected:

    int      NumTop,    // Число вершин
             NumArc,    // Число ребер
             start,     // Вершина - старт пути
             finish;    // Вершина - финиш пути

    // Адрес первого элемента массива структур с информацией о ребрах
    // графа
    A        *pArc;

    // Методы

public:

    // Конструктор
    GR( int nt, int na );

    ~GR( void )      // Деструктор: подставляемый метод
    {
        if( pArc )
        {
            delete [ ] pArc; pArc = NULL;
        }
    }

    // Ввод информации о графе из файла на магнитном диске
    void ReadGraph( char f_name[ ] );

    // Вывод информации о графе в файл на магнитном диске
    void WriteGraph( char f_name[ ], int mode, char text[ ] );

}; // Конец объявления класса

```

В производном классе нужно иметь данные, члены класса, и методы класса. Как показано в [3], информацию о наилучшем пути между заданными вершинами удобно хранить в виде линейного списка, реализованного на базе массива структур. При этом используется структурный тип следующего вида:

```

struct W          // Way: путь до одной вершины
{

```

```

int      exist;    // ( != 0 ) - путь имеется
int      ref;      // REference: предыдущая вершина, через которую
                  //   проходит путь
float    SumDist;  // Суммарная длина минимального пути
};

```

Следовательно, в производном классе в качестве данного следует использовать указатель на массив структур, размещенный в динамической памяти:

```

// Адрес первого элемента массива структур с информацией
//   о минимальном пути между заданными вершинами
W        *pMinWay;

```

Этот указатель используется только в производном классе и его можно защитить (**private:**).

В качестве методов производного класса можно использовать интерфейсные методы (**public:**): конструктор (размещение в динамической памяти массива структур с информацией о наилучшем пути и его инициализация), деструктор (освобождение динамической памяти), метод `solution()` для решения транспортной задачи и метод `OutRes()` для печати результатов решения транспортной задачи. В соответствии с декомпозицией решения транспортной задачи метод `solution()` использует два вспомогательных взаимно-рекурсивных метода — `PassWay()` и `ForStep()`, которые следует закрыть. Рекурсивный метод `PassWay()` использует две вспомогательных переменных — `one` и `two`, которые достаточно иметь в одном экземпляре. Поэтому они сделаны закрытыми членами класса (**private:**).

Таким образом, производный класс может иметь следующую структуру:

```

struct W                // Way: путь до одной вершины
{
    int      exist;      // ( != 0 ) - путь имеется
    int      ref;        // REference: предыдущая вершина, через которую
                        //   проходит путь
    float    SumDist;    // Суммарная длина минимального пути
};

// *****
// Производный класс для решения задачи коммивояжера: GRaph for the
//   Commercial Traveller
class GR_CT : public GR
{
    // Данные

private:

    // Адрес первого элемента массива структур с информацией
    //   о минимальном пути между заданными вершинами
    W        *pMinWay;

    // Нижеследующие данные определены здесь для экономии памяти стека,
    //   так как они используются в одном экземпляре в рекурсивной

```

```

// функции PassWay
int         one,          // 1-я вершина текущего ребра
            two;          // 2-я вершина текущего ребра

// Методы

public:

// Конструктор
GR_CT( int nt, int na );

~GR_CT( void )          // Подставляемый деструктор
{
    if( pMinWay )
    {
        delete [ ] pMinWay; pMinWay = NULL;
    }
}

// Поиск оптимального пути в неориентированном взвешенном графе
void solution( void );

// Печать информации о наилучшем пути от start до finish
void OutRes( char f_name[ ], int mode, char text[ ] );

private:

// Прохождение пути от достигнутой вершины InterMediate, если он
// есть, до вершины finish
void PassWay( int InterMediate );

// Шаг вперед из достигнутой вершины по заданному ребру
void ForStep( int top1, int IndArc, int top2 );

};                                     // Конец объявления класса

```

Файловая структура иерархии классов. Ранее было указано (см. разд. 1.6), что определения обычных классов можно целиком размещать в соответствующих заголовочных файлах либо объявления классов помещать в соответствующие заголовочные файлы, а реализацию их методов — в файлы с расширением `сpp`. Первый вариант представляется автору более целесообразным. Он и будет использован в следующем разделе. Альтернативный вариант реализации программы будет предложено выполнить читателям самостоятельно в разд. 9.3 в качестве упражнения для самопроверки (для всех упражнений такого рода в учебном пособии в *приложении 1* приведены ответы). Поскольку базовый класс для работы с графами может использоваться не только для решения транспортной задачи, но и при решении других задач обработки графов, то определения базового класса и производного класса должны размещаться в разных заголовочных файлах.

9.2. Решение транспортной задачи с использованием иерархии обычных классов

Теперь приведем полную программу решения транспортной задачи с использованием иерархии обычных классов, соответствующих спецификации, разработанной в предыдущем разделе (листинги 9.1—9.4). Эта программа использует класс `IOFile`, разработанный в *разд. 5.6*. Этот класс позволяет удобно открывать и закрывать файлы и обеспечивает использование перегруженных операций "<<" и ">>" для вывода и ввода предопределенных типов. Исходный текст заголовочного файла `IOFile.h` с описанием класса `IOFile` приведен ранее и здесь не повторяется.

Листинг 9.1. Файл `Graph_B.h`

```
/*
    Базовый класс для объекта графа со следующим набором операций:
    - динамическое размещение данных о графе (конструктор);
    - освобождение занятой динамической памяти (деструктор);
    - чтение информации о графе;
    - печать информации о графе.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6
*/

// Предотвращение многократного включения данного файла
#ifndef __GRAPH_B_H
#define __GRAPH_B_H

    // Класс для открытия-закрытия файлов на базе стандартного класса
    // fstream и подключение перегруженных операций << и >>
    #include "IOFile.h"

    struct A                // Arc: дуга (ребро) графа
    {
        int    first;       // 1-я вершина ребра
        int    last;        // 2-я вершина ребра
        float  weight;      // Вес ребра
    };

    // *****
    // Базовый класс для объекта графа
    class GR
    {
        // Данные

    protected:

        int    NumTop,      // Число вершин
```

```

        NumArc,    // Число ребер
        start,    // Вершина - старт пути
        finish;   // Вершина - финиш пути

// Адрес первого элемента массива структур с информацией о ребрах
// графа
A      *pArc;

// Методы

public:

// Конструктор
GR( int nt, int na );

~GR( void )      // Деструктор: подставляемый метод
{
    if( pArc )
    {
        delete [ ] pArc; pArc = NULL;
    }
}

// Ввод информации о графе из файла на магнитном диске
void ReadGraph( char f_name[ ] );

// Вывод информации о графе в файл на магнитном диске
void WriteGraph( char f_name[ ], int mode, char text[ ] );

};          // Конец объявления класса

// *****
// Конструктор
GR :: GR(
    int    nt,        // Число вершин графа
    int    na )       // Число ребер графа
{
    // Проверка области допустимых значений nt и na
    if( nt < 2 )
    {
        cout << "\n Error 10. In the graph there should"
              " be 2 or more top " << endl;
        exit( 10 );
    }
    if( na < 1 )
    {

```

```

        cout << "\n Error 20. In the graph there should"
              " be 1 or more edges " << endl;
        exit( 20 );
    }

    // Размещаем массив ребер в динамической памяти
    pArc = new A[ na ];
    if( !pArc )
    {
        cout << "\n Error 30. The information on edges of a graph "
              "was not placed in dynamic memory " << endl;
        exit( 30 );
    }

    // Инициализация данных
    for( int i = 0; i < na; i++ )
    {
        pArc[ i ].first = 0; pArc[ i ].last = 0;
        pArc[ i ].weight = 0.0f;
    }
    NumTop = nt; NumArc = na;
}

// *****
// Ввод информации о графе из файла на магнитном диске
void GR :: ReadGraph(
    // Файл ввода
    char f_name[ ] )
{
    IOFILE f_in;      // файловый объект для ввода

    // Открытие файла для чтения
    f_in.open_f( f_name, ios::in, 50 );

    // Чтение данных о графе
    f_in >> start >> finish;
    if( !f_in )
    {
        cout << "\n Error 60. A read error from the file" << f_name
              << endl;
        exit( 60 );
    }
    for( int i=0; i<NumArc; i++ )
    {
        f_in >> pArc[ i ].first >> pArc[ i ].last >>
              pArc[ i ].weight;
    }
}

```

```

        if( !f_in )
        {
            cout << "\n Error 70. A read error from the file"
                << f_name << endl;
            exit( 70 );
        }
    }

    // Закрытие файла ввода
    f_in.close_f( f_name, 80 );

    return;
}

// *****
// Вывод информации о графе в файл на магнитном диске
void GR :: WriteGraph(
    char   f_name[ ], // Файл вывода
    int    mode,      // РЕЖИМ ОТКРЫТИЯ файла
    char   text[ ] ) // Заголовок для печати
{
    IOFILE f_out;      // Файловый объект для вывода

    // Открытие файла для вывода
    f_out.open_f( f_name, mode, 90 );

    // Вывод информации о графе
    f_out << text << endl << "        Число вершин графа: " << NumTop
        << ", число ребер: " << NumArc << endl << endl <<
        " Индекс ребра   1 вершина   2 вершина       Вес ребра ";
    for( int i=0; i<NumArc; i++ )
    {
        f_out << endl; f_out.width( 8 );
        f_out << i; f_out.width( 13 );
        f_out << pArc[ i ].first; f_out.width( 13 );
        f_out << pArc[ i ].last;
        f_out.width( 13 ); f_out << pArc[ i ].weight;
    }

    // Закрытие файла вывода
    f_out.close_f( f_name, 100 );

    return;
}

#endif

```

Листинг 9.2. Файл Graph_D.h

```

/*
    Задача коммивояжера - нахождение наилучшего пути из заданной вершины графа
    (из заданного города) в другую заданную вершину (в другой заданный город).

    Используется класс, производный от базового класса (см. файл Graph_B.h) со сле-
    дующим набором операций:
        - поиск оптимального пути в неориентированном взвешенном графе;
        - выполнение шага вперед из достигнутой вершины по заданному ребру;
        - прохождение пути, если он есть, от достигнутой вершины до конечной вершины;
        - печать информации о наилучшем пути между заданными вершинами.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предотвращение многократного включения данного файла
#ifndef __GRAPH_D_H
#define __GRAPH_D_H

    // Подключение базового класса
    #include "Graph_B.h"

    struct W          // Way: путь до одной вершины
    {
        int    exist;    // ( != 0 ) - путь имеется
        int    ref;      // REference: предыдущая вершина, через которую
                        //   проходит путь
        float  SumDist;  // Суммарная длина минимального пути
    };

    // *****
    // Производный класс для решения задачи коммивояжера: GRaph for the
    //   Commercial Traveller
    class GR_CT : public GR
    {
        // Данные

    private:

        // Адрес первого элемента массива структур с информацией
        //   о минимальном пути между заданными вершинами
        W      *pMinWay;

        // Нижеследующие данные определены здесь для экономии памяти
        //   стека, так как они используются в одном экземпляре
        //   в рекурсивной функции PassWay
        int    one,      // 1-я вершина текущего ребра

```

```

        two;          // 2-я вершина текущего ребра

// Методы

public:

    // Конструктор
    GR_CT( int nt, int na );

    ~GR_CT( void )      // Подставляемый деструктор
    {
        if( pMinWay )
        {
            delete [ ] pMinWay; pMinWay = NULL;
        }
    }

    // Поиск оптимального пути в неориентированном взвешенном графе
    void solution( void );

    // Печать информации о наилучшем пути от start до finish
    void OutRes( char f_name[ ], int mode, char text[ ] );

private:

    // Прохождение пути от достигнутой вершины InterMediate, если он
    // есть, до вершины finish
    void PassWay( int InterMediate );

    // Шаг вперед из достигнутой вершины по заданному ребру
    void ForStep( int top1, int IndArc, int top2 );

};          // Конец объявления класса

// *****
// Конструктор
GR_CT :: GR_CT(
    int    nt,          // Число вершин графа
    int    na )         // Число ребер графа
: GR( nt, na )
{
    // Размещаем массив структур с информацией о наилучшем пути
    // в динамической памяти
    pMinWay = new W[ nt ];
    if( !pMinWay )
    {
        cout << "\n Error 40. The information on the "

```

```

        "best path was not placed in dynamic memory " << endl;
        exit( 40 );
    }

    // Инициализация данных
    for( int i = 0; i < nt; i++ )
    {
        pMinWay[ i ].exist = 0; pMinWay[ i ].ref = 0;
        pMinWay[ i ].SumDist = 0.0f;
    }
}

// *****
// Поиск оптимального пути в неориентированном взвешенном графе
void GR_CT :: solution( void )
{
    // Подготовка
    pMinWay[ start ].exist = 1;
    pMinWay[ start ].ref = -1;

    PassWay( start );

    return;
}

// -----
// Прохождение пути от достигнутой вершины InterMediate, если он
// есть, до вершины finish
void GR_CT :: PassWay(
    // Достигнутая вершина - отправная точка пути
    int    InterMediate )
{
    int    k;           // Индекс текущей дуги графа

    if( InterMediate == finish )
        return;         // !!! Выход из рекурсии

    for( k = 0; k < NumArc; k++ )
    {
        // Перебор ребер графа
        one = pArc[ k ].first; two = pArc[ k ].last;
        if( one == InterMediate )
            ForStep( one, k, two );
        else if( two == InterMediate )
            ForStep( two, k, one );
    }

    return;             // Альтернативный вариант выхода из рекурсии
}

```

```

}

// -----
// Шаг вперед из достигнутой вершины по заданному ребру
void GR_CT :: ForStep(
    int    top1,      // Достигнутая вершина, из которой шагаем вперед
    int    IndArc,    // Индекс ребра, по которому делается шаг вперед
    int    top2 )     // Вершина на конце ребра
{
    float  NewDist;   // Расстояние до top2 по пути через top1

    NewDist = pMinWay[ top1 ].SumDist + pArc[ IndArc ].weight;
    if( !pMinWay[ top2 ].exist )
    {
        // Пока пути до top2 нет
        pMinWay[ top2 ].exist = 1;
        pMinWay[ top2 ].SumDist = NewDist;
        pMinWay[ top2 ].ref = top1; PassWay( top2 );
    }
    else
    {
        // Путь до top2 существует
        if( pMinWay[ top2 ].SumDist > NewDist )
        {
            // Новый путь короче
            pMinWay[ top2 ].SumDist = NewDist;
            pMinWay[ top2 ].ref = top1;
            PassWay( top2 );
        }
    }
}

return;
}

// *****
// Печать информации о наилучшем пути от start до finish
void GR_CT :: OutRes(
    char    f_name[ ], // Файл вывода
    int     mode,      // Режим открытия файла
    char    text[ ] )  // Заголовок для печати
{
    IOFILE f_out;      // Файловый объект для вывода

    // Открытие файла для вывода
    f_out.open_f( f_name, mode, 110 );

    // Данные для инвертирования списка ссылок
    int     TempTop,   // Текущая вершина пути
            pred,      // Предыдущая вершина пути
            next;      // Следующая вершина пути

```

```

if( !pMinWay[finish]. exist )
{
    cout << "\n Error 120. The required path does not exist "
        << endl;
    exit( 120 );
}

// Инвертирование списка ссылок
TempTop = finish; next = -1;
while( TempTop != -1 )
{
    pred = pMinWay[ TempTop ].ref;
    pMinWay[ TempTop ].ref = next;
    next = TempTop; TempTop = pred;
}

// Вывод результатов поиска
f_out << text << "\n Оптимальный путь от старта до финиша "
    "проходит через \nследующие вершины (первая вершина списка"
    " - старт, \nпоследняя - финиш пути):" << endl;

int    i = 0;    // Счетчик отпечатанных вершин
while( next != -1 )
{
    if( !( i%4 ) )
        f_out << endl;
    i++; f_out.width( 12 ); f_out << next;
    next = pMinWay[ next ].ref;
}

// Закрытие файла вывода
f_out.close_f( f_name, 130 );

return;
}

#endif

```

Листинг 9.3. Файл TestGr.cpp

```

/*
    Тестирование решения транспортной задачи (задачи коммивояжера). При решении
    задачи используется базовый класс GR, определение которого приведено во включаемом
    файле Graph_V.h, и производный класс GR_CT, определение которого приведено
    во включаемом файле Graph_D.h. Для открытия-закрытия файлов используется класс
    IOFILE, определение которого приведено во включаемом файле IOFile.h.

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6

```
*/

#include "Graph_D.h"      // Класс для решения транспортной задачи

// *****
// Тестирование
int main( void )         // Возвращает 0 при успехе
{
    GR_CT      g( 5, 6 ); // Создаем граф из 5 вершин и 6 ребер

    // Ввод информации о графе из файла
    g.ReadGraph( "graph.dat" );

    // Вывод информации о графе в файл
    g.WriteGraph( "graph.out", ios::out,
"                ИНФОРМАЦИЯ О ГРАФЕ \n" );

    g.solution( );        // Ищем наилучший путь между вершинами

    g.OutRes( "graph.out", ios::out | ios::app,
"\n\n          РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ \n");

    return 0;
}
```

Листинг 9.4. Результаты работы программы, выводимые в файл на магнитном диске

ИНФОРМАЦИЯ О ГРАФЕ

Число вершин графа: 5, число ребер: 6

Индекс ребра	1 вершина	2 вершина	Вес ребра
0	0	1	80
1	0	3	10
2	1	2	20
3	1	4	10
4	2	3	20
5	3	4	10

РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ

Оптимальный путь от старта до финиша проходит через следующие вершины (первая вершина списка - старт, последняя - финиш пути):

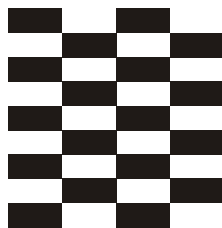
0 3 4 1

9.3. Упражнения для самопроверки

1. Программу из *разд. 9.2* модифицируйте таким образом, чтобы объявления классов размещались в заголовочных файлах, а реализация методов классов — в файлах с расширением `spp`.
2. Программу из упражнения 1 в учебных целях модифицируйте таким образом, чтобы вместо иерархии классов для решения транспортной задачи использовался один класс. Объявление этого класса поместите в заголовочный файл, а реализацию методов — в файл с расширением `spp`.

Ответы на эти упражнения приведены в *разд. П1.8 приложения 1*.

Глава 10



Поиск в таблице

Как и в *разд. 8, 9*, прежде чем перейти к проектированию объектно-ориентированной программы для решения задачи поиска в таблице, внимательно рассмотрите учебный материал, приведенный в [3]. Он будет использован в качестве основы для разработки программы.

При решении задачи поиска объектом является таблица, информацию о которой целесообразно разместить в динамической памяти. Свойства таблицы определяются этой информацией. Таблица, как объект, может представлять собой конкретные различные объекты реального мира. В виде таблицы можно представить себе словарь, например англо-русский. В строке такой таблицы в поле ключа может содержаться некоторое английское слово, а в поле данного — связанная с ним информация. Для этого примера можно говорить о поиске заданного слова в словаре. Для работы с такого рода объектом-таблицей можно использовать интерфейс, содержащий следующий набор операций: размещение таблицы в динамической памяти, заполнение таблицы данными, просмотр (вывод) содержимого таблицы, поиск информации в таблице, вывод сведений о результатах поиска и освобождение памяти, занятой таблицей. Из сказанного со всей очевидностью следует, что решение задачи поиска в таблице удобно реализовать с использованием класса или иерархии классов.

10.1. Спецификация класса для поиска в таблице

Иерархия шаблонных или обычных классов. Строка таблицы содержит два поля: поле ключа, используемое для поиска в таблице, и поле данного. В конкретных случаях эти параметры могут варьироваться и их можно задать в виде настроечных параметров шаблона классов. При поиске в таблице можно использовать следующий интерфейсный набор операций: размещение таблицы в динамической памяти, заполнение таблицы данными, просмотр (вывод) содержимого таблицы, поиск информации в таблице, вывод сведений о результатах поиска и освобождение памяти, занятой таблицей. При этом существенным является то, что алгоритмы заполнения таблицы существенно зависят от используемых методов поиска [3]. По этой причине отрывать их от методов поиска, помещая методы заполнения таблицы в базовый класс, не представляется целесообразным. Следовательно, в данном случае иерархия классов не нужна и для поиска в таблице достаточно использовать один шаблонный класс:

```
// Объявление шаблонного класса для поиска в таблице: LenKey-1 - длина
// поля ключа, LenData-1 - длина поля данного в строке таблицы
```

```
template< unsigned int LenKey, unsigned int LenData >
class SEARCH_TABLE
{
    // ...
}; // Конец объявления шаблонного класса
```

Структура класса. Структура классов иерархии определяется составом членов класса, их функциональным назначением и защищенностью. Ранее в *разд. 2.3* указывалось, что для обоснованного выбора структуры класса можно руководствоваться следующими общими соображениями.

Если некоторое данное используется более чем в одном методе класса, или даже в единственном методе, который при решении задачи часто вызывается, то такое данное следует сделать членом класса. При этом если такой член-данные используется только методами данного класса, то его следует закрыть (снабдить спецификатором доступа **private**:). Если член-данные должен быть доступен в этом классе и производных от него классах, то его следует защитить (снабдить спецификатором доступа **protected**:). Открывать члены-данные класса не следует практически никогда, так как при этом класс становится уязвимым, что делает программный код ненадежным. В соответствии с этим мы выбираем в шаблонном классе следующие данные, являющиеся членами класса. Для представления строки таблицы будем использовать локальный структурный тип вида:

```
struct STR_TAB // Строка таблицы
{
    // Ключ
    char      Key[ LenKey ];
    // Данное
    char      Data[ LenData ];
};
```

При поиске в таблице практически во всех методах используются следующие данные, которые целесообразно сделать членами класса и закрыть:

```
// Данные
```

private:

```
STR_TAB    *pTable; // Адрес первой строки таблицы
unsigned int
    size; // Размер таблицы
```

При проектировании методов класса обычно руководствуются следующими соображениями. В качестве метода класса следует использовать функцию, реализующую решение некоторой функционально-законченной задачи. Для обеспечения разумного размера метода можно использовать "правило 7 ± 2 ", смысл которого рассмотрен в [3]. В соответствии с этим правилом размер метода не должен превышать 25–80 строк исходного текста, а количество параметров не должно превышать 5–7. Выполнение этих требований можно обеспечить путем надлежащей иерархической декомпозиции "больших" методов. Все разновидности конструкторов, деструкторы и интерфейсные методы класса, с помощью которых пользователь взаимодействует с классом, нужно сделать доступными из любой программной среды (**public**:). Вспомогательные,

не интерфейсные методы следует либо закрыть (**private:**), если они нужны только в этом классе, либо защитить (**protected:**), если они нужны в этом и производных от него классах. Функциональную декомпозицию решения задачи, выполненную в [3], вполне можно взять за основу при выборе состава методов класса. Таким образом, получаем следующую структуру шаблона классов:

```
// Объявление шаблонного класса для поиска в таблице: LenKey-1 - длина
// поля ключа, LenData-1 - длина поля данного в строке таблицы
```

```
template< unsigned int LenKey, unsigned int LenData >
```

```
class SEARCH_TABLE
```

```
{
```

```
    // Локальные типы
```

```
    struct STR_TAB          // Строка таблицы
```

```
    {
```

```
        // Ключ
```

```
        char    Key[ LenKey ];
```

```
        // Данное
```

```
        char    Data[ LenData ];
```

```
    };
```

```
    // Данные
```

```
private:
```

```
    STR_TAB    *pTable; // Адрес первой строки таблицы
```

```
    unsigned int
```

```
        size;          // Размер таблицы
```

```
    // файловый объект определяем здесь, чтобы при рекурсивных вызовах
```

```
    // вспомогательного метода Round( ) не создавались его копии
```

```
    IOFILE      FIn;      // файловый объект для ввода
```

```
    // Методы
```

```
public:                // Интерфейсные методы
```

```
    // Конструктор: размещает таблицу в динамической памяти и
```

```
    // инициализирует ее
```

```
    SEARCH_TABLE( unsigned int s );
```

```
    // Деструктор: освобождает динамическую память
```

```
    ~SEARCH_TABLE( void )
```

```
    {
```

```
        if( pTable )
```

```
        {
```

```
            delete [ ] pTable; pTable = NULL;
```

```

    }
}

// Заполнение таблицы для последовательного поиска
void SeqInpTab( char FName[ ] = "SearchT.dat" );

// Заполнение таблицы для логарифмического поиска: используется
// вспомогательный рекурсивный метод Round( )
void LogInpTab( char FName[ ] = "SearchT.dat" );

// Заполнение таблицы для хэш-поиска: используются вспомогательные
// методы Kod( ) и Hash( )
void HashInpTab( char FName[ ] = "SearchT.dat",
                unsigned int TableLen = 2 );

// Вывод таблицы
void PrintTab( char FName[ ] = "SearchT.out", int mode = ios::app,
             char text[ ] = "        Состояние таблицы:" );

// Последовательный поиск в таблице
void SequentialSearch( char KeyWord[ ], bool &found,
                     unsigned int &IxLine );

// Вывод результатов поиска в таблице
void PrintSearch( char KeyWord[ ], bool found, unsigned int IxLine,
                char FName[ ] = "SearchT.out",
                int mode = ios::app );

// Бинарный (логарифмический) поиск в таблице, подготовленной
// в форме алфавитно-упорядоченного двоичного дерева
void LogarithmSearch( char KeyWord[ ], bool &found,
                    unsigned int &IxLine );

// Поиск в хэш-таблице: используются вспомогательные методы Kod( ) и
// Hash( )
void HashSearch( char KeyWord[ ], bool &found,
               unsigned int &IxLine );

private:                // Вспомогательные методы

// Обход вершин дерева с целью формирования словаря для бинарного
// (логарифмического) поиска. !!! Читаемые данные должны быть
// алфавитно-упорядоченными по ключам
void Round( unsigned int root );

// Преобразование символа ключа - строчная латинская буква, цифра или

```

```

// пробел - в его порядковый номер (целое число)
int Kod( char symbol );

// Хэш-функция ключа KeyWord[ ] из LenKey-1 символов (символ -
// строчная латинская буква, цифра или пробел) для таблицы из size
// строк
unsigned int Hash( char KeyWord[ ] );

}; // Конец объявления шаблонного класса

```

Новым, по сравнению с программными проектами, приведенными в *разд. 8* и *9*, является широкое использование в методах класса умалчиваемых значений параметров. Далее, в *разд. 10.2*, читатель увидит, какие дополнительные удобства обеспечивает это при работе с шаблонным классом.

Файловая структура шаблонного класса. В *разд. 1.6* было обосновано и показано, что определение шаблона классов должно целиком размещаться в заголовочном файле.

10.2. Поиск в таблице с использованием шаблонного класса

Теперь приведем полную программу поиска в таблице с использованием шаблонного класса, соответствующего разработанной спецификации (листинги 10.1—10.3). Эта программа использует класс `IOFILE` (см. *разд. 5.6*), позволяющий удобно открывать и закрывать файлы и обеспечивающий использование перегруженных операций "<<" и ">>" для вывода и ввода предопределенных типов. Исходный текст заголовочного файла `IOFile.h` с описанием класса `IOFILE` приведен ранее и здесь не повторяется.

Листинг 10.1. Файл `SearchT.h`

```

/*
Поиск в таблице. Используется шаблонный класс со следующим набором операций:
1. Динамическое размещение таблицы с инициализацией (конструктор).
2. Освобождение занятой динамической памяти (деструктор).
3. Заполнение таблицы для последовательного поиска, логарифмического поиска
   и хэш-поиска.
4. Печать содержимого таблицы.
5. Последовательный поиск в таблице.
6. Логарифмический поиск в таблице.
7. Хэш-поиск в таблице.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// *****
// Предотвращение многократного включения данного файла
#ifndef __SEARCHT_H
#define __SEARCHT_H

```

```

// Класс для открытия-закрытия файлов на базе стандартного класса
// fstream и подключение перегруженных операций << и >>
#include "IOFile.h"

#include <iomanip>    // Для манипуляторов ввода-вывода
#include <string.h>   // Для работы с С-строками

// *****
// Объявление шаблонного класса для поиска в таблице: LenKey-1 -
// длина поля ключа, LenData-1 - длина поля данного в строке
// таблицы
template< unsigned int LenKey, unsigned int LenData >
class SEARCH_TABLE
{
    // Локальные типы

    struct STR_TAB    // Строка таблицы
    {
        // Ключ
        char
            Key[ LenKey ];
        // Данное
        char
            Data[ LenData ];
    };

    // Данные

private:

    STR_TAB
        *pTable;    // Адрес первой строки таблицы
    unsigned int
        size;        // Размер таблицы

    // файловый объект определяем здесь, чтобы при рекурсивных
    // вызовах не создавались его копии
    IOFILE FIn;      // Файловый объект для ввода

    // Методы

public:

    // Конструктор

```

```
SEARCH_TABLE( unsigned int s );

// Деструктор
~SEARCH_TABLE( void )
{
    if( pTable )
    {
        delete [ ] pTable; pTable = NULL;
    }
}

// Заполнение таблицы для последовательного поиска
void SeqInpTab( char FName[ ] = "SearchT.dat" );

// Заполнение таблицы для логарифмического поиска
void LogInpTab( char FName[ ] = "SearchT.dat" );

// Заполнение таблицы для хэш-поиска
void HashInpTab( char FName[ ] = "SearchT.dat",
                unsigned int TableLen = 2 );

// Вывод таблицы
void PrintTab( char FName[ ] = "SearchT.out",
              int mode = ios::out | ios::app,
              char text[ ] = "        Состояние таблицы:" );

// Последовательный поиск в таблице
void SequentialSearch( char KeyWord[ ], bool &found,
                     unsigned int &IxLine );

// Вывод результатов поиска в таблице
void PrintSearch( char KeyWord[ ], bool found,
                 unsigned int IxLine,
                 char FName[ ] = "SearchT.out",
                 int mode = ios::out | ios::app );

// Бинарный (логарифмический) поиск в таблице, подготовленной
// в форме алфавитно-упорядоченного двоичного дерева
void LogarithmSearch( char KeyWord[ ], bool &found,
                    unsigned int &IxLine );

// Поиск в хэш-таблице
void HashSearch( char KeyWord[ ], bool &found,
                unsigned int &IxLine );
```

```

private:

    // Обход вершин дерева с целью формирования словаря для бинарного
    // (логарифмического) поиска.!!! Читаемые данные должны быть
    // алфавитно-упорядочены по ключам
    void Round( unsigned int root );

    // Преобразование символа ключа - строчная латинская буква, цифра
    // или пробел - в его порядковый номер (целое число)
    int Kod( char symbol );

    // Хэш-функция ключа KeyWord[ ] из LenKey-1 символов (символ -
    // строчная латинская буква, цифра или пробел) для таблицы из
    // size-строк
    unsigned int Hash( char KeyWord[ ] );

};                                     // Конец объявления шаблонного класса

// *****
// Конструктор: s - число строк таблицы
template< unsigned int LenKey, unsigned int LenData >
SEARCH_TABLE< LenKey, LenData >
:: SEARCH_TABLE( unsigned int s )
{
    // Проверяем, подходит ли размер таблицы
    if( s<2 )
    {
        cout << "\n Error 10. In the table should be "
              "not less than two string" << endl;
        exit( 10 );
    }

    // Размещаем таблицу в динамической памяти
    pTable = new STR_TAB[ s ];
    if( !pTable )
    {
        cout << "\n Error 20. The table in dynamic "
              "memory not was placed " << endl;
        exit( 20 );
    }

    // Инициализация таблицы
    for( unsigned int i=0; i<s; i++ )
    {
        pTable[ i ].Key[ 0 ] = '\0';
        pTable[ i ].Data[ 0 ] = '\0';
    }
}

```

```

    }
    size = s;
}

// *****
// Заполнение таблицы для последовательного поиска
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: SeqInpTab(
    char    FName[ ] )// Файл ввода
{
    // Открытие файла для чтения
    FIn.open_f( FName, ios::in, 30 );

    // Заполнение таблицы
    for( unsigned int i=0; i<size; i++ )
    {
        FIn >> pTable[ i ].Key >> pTable[ i ].Data;
        if( !FIn )
        {
            cout << endl << " Error 40. A read error "
                "of units of the table";
            exit( 40 );
        }
    }

    // Закрытие файла ввода
    FIn.close_f( FName, 50 );

    return;
}

// *****
// Заполнение таблицы для логарифмического поиска
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: LogInpTab(
    char    FName[ ] )// Файл ввода
{
    // Открытие файла для чтения
    FIn.open_f( FName, ios::in, 60 );

    Round( 1 );          // Рекурсивное заполнение таблицы

    // Закрытие файла ввода
    FIn.close_f( FName, 70 );

    return;
}

```

```

}

// *****
// Вывод таблицы
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: PrintTab(
    char    FName[ ], // Файл вывода
    int     mode,      // Режим открытия файла
    char    text[ ] ) // Заголовок для печати
{
    IOFILE FOut;       // Файловый объект для вывода

    // Открытие файла для записи
    FOut.open_f( FName, mode, 80 );

    // Печать таблицы с заголовком
    FOut << endl << text << endl;
    for( unsigned int i=0; i<size; i++ )
    {
        FOut.width( LenKey-1 );
        FOut << setiosflags( ios::left ) << pTable[ i ].Key;
        FOut.width( LenData-1 );
        FOut << setiosflags( ios::left ) << pTable[ i ].Data << endl;
    }

    // Закрытие файла вывода
    FOut.close_f( FName, 90 );

    return;
}

// *****
// Последовательный поиск в таблице
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: SequentialSearch(
    // Ключ для поиска строки в таблице
    char    KeyWord[ ],
    bool    &found,    // 1 - нашли
    unsigned int
        &IdxLine ) // Индекс найденной строки
{
    unsigned int
        i;          // Индекс анализируемой строки
    bool    EndTab; // true - конец заполненной части таблицы

```

```

// Подготовка к поиску
found = false; EndTab = false;

// Поиск
i = 0;
while( !found && !EndTab )
{
    if( !strcmp( KeyWord, pTable[ i ].Key ) )
    {
        // Нашли
        found = true; IxLine = i;
    }
    else
    {
        // Шаг вперед по таблице
        if( i == size-1 )
            EndTab = true;
        else
            i++;
    }
}

return;
}

// *****
// Вывод результатов поиска в таблице
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: PrintSearch(
    // Ключевое слово для поиска
    char    KeyWord[ ],
    bool    found,      // 1 - нашли в таблице
    unsigned int
        IxLine,      // Индекс строки в таблице
    char    FName[ ], // Файл вывода
    int     mode )    // Режим открытия файла
{
    IOFILE FOut;      // Файловый объект для вывода

    // Открытие файла для записи
    FOut.open_f( FName, mode, 100 );

    FOut << endl << "Результаты поиска для ключевого слова: "
        << KeyWord << endl;
    if( found )
    {
        FOut << "Индекс строки в таблице: " << IxLine <<

```

```

        ". Найденная строка:" << endl;
        FOut.width( LenKey-1 );
        FOut << setiosflags( ios::left ) << pTable[ IxLine ].Key;
        FOut.width( LenData-1 );
        FOut << setiosflags( ios::left ) << pTable[ IxLine ].Data
            << endl;
    }
    else
        FOut << "Строка с ключом \" << KeyWord <<
            "\" в таблице не найдена" << endl;

    // Закрытие файла вывода
    FOut.close_f( FName, 110 );

    return;
}

// *****
// Обход вершин дерева с целью формирования словаря для бинарного
// (логарифмического) поиска. !!! Читаемые данные должны быть
// алфавитно-упорядоченными по ключам
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: Round(
    unsigned int
        root )    // Корень дерева
{
    if( size < root )
        return;    // Выход из рекурсии

    Round( 2*root ); // Обойти вершины левого поддерева

    // Приписать корню очередную по алфавиту строку таблицы
    FIn >> setw( LenKey-1 ) >> pTable[ root-1 ].Key >>
        setw( LenData-1 ) >> pTable[ root-1 ].Data;
    if( !FIn )
    {
        cout << endl << " Error 120. A read error ";
        exit( 120 );
    }

    // Обойти вершины правого поддерева
    Round( 2*root+1 );

    return;
}

```

```

// *****
// Бинарный (логарифмический) поиск в таблице, подготовленной в форме
// алфавитно-упорядоченного двоичного дерева
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: LogarithmSearch(
    // Ключ для поиска строки в таблице
    char    KeyWord[ ],
    bool    &found,    // true - нашли
    unsigned int
        &IxLine ) // Индекс найденной строки
{
    unsigned int
        i;          // Индекс вершины дерева
    bool    EndTab;  // true - достигнут конец таблицы

    // Подготовка к поиску
    found = false; EndTab = false;

    // Поиск
    i = 1;
    while( !found && !EndTab )
    {
        if( !strcmp( KeyWord, pTable[ i-1 ].Key ) )
        {
            //Нашли
            found = true; IxLine = i-1;
        }
        else
        {
            // Шаг вперед по таблице
            if( strcmp( KeyWord, pTable[ i-1 ].Key )<0 )
                i = 2*i;
            else
                i = 2*i+1;
            EndTab = ( i>size );
        }
    }

    return;
}

// *****
// Преобразование символа ключа - строчная латинская буква, цифра или
// пробел - в его порядковый номер (целое число)
template< unsigned int LenKey, unsigned int LenData >
int SEARCH_TABLE< LenKey, LenData > :: Kod(

```

```

        // Порядковый номер символа
char    symbol ) // Преобразуемый символ
{
    switch( symbol )
    {
        case 'a': return 0;
        case 'b': return 1;
        case 'c': return 2;
        // И так далее
        case 'y': return 24;
        case 'z': return 25;
        case '0': return 26;
        case '1': return 27;
        // И так далее
        case '8': return 34;
        case '9': return 35;
        case ' ': return 36;

        default:
            cout << "\n Error 130. The invalid character in a key:"
                 " \n the key can consist only of line Latin "
                 "characters, digits and blanks " << endl;
            exit( 130 );
    }

    return -1; // Этот оператор не будет выполняться - он
              // помещен сюда для "глупого" компилятора
}

// *****
// Хэш-функция ключа KeyWord[ ] из LenKey-1 СИМВОЛОВ (СИМВОЛ -
//   строчная латинская буква, цифра или пробел) для таблицы из size
//   строк
template< unsigned int LenKey, unsigned int LenData >
unsigned int SEARCH_TABLE< LenKey, LenData > :: Hash(
    // Возвращает индекс строки таблицы
    // Ключ
    char    KeyWord[ ] )
{
    unsigned int
        IKey, // Индекс символа в ключе
        ih = 0; // Значение хэш-функции

    // Вычисление индекса строки таблицы
    for( IKey=0; IKey<strlen( KeyWord ); IKey++ )

```

```

    {
        ih = ih * 37 + Kod( KeyWord[ IKey ] );
        ih = ih % size;
    }

    return ih;
}

// *****
// Заполнение таблицы для хэш-поиска
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: HashInpTab(
    char    FName[ ], // Файл ввода
    unsigned int
        TableLen )// Число вводимых строк
{
    unsigned int
        i,          // Индекс строки таблицы
        line;       // Индекс текущей строки файла
    bool    found;   // true - найдена позиция вставки
    // Заносимое слово
    char    KeyWord[ LenKey ];

    // Инициализация таблицы нуль-символами
    for( i=0; i<size; i++ )
    {
        for( unsigned int il=0; il<LenKey; il++ )
            pTable[ i ].Key[ il ] = '\0';
        for( il=0; il<LenData; il++ )
            pTable[ i ].Data[ il ] = '\0';
    }

    // Отметка строк таблицы как свободных
    for( i=0; i<size; i++ )
        pTable[ i ].Key[ 0 ] = ' ';

    // Открытие файла для чтения
    FIn.open_f( FName, ios::in, 140 );

    // Занесение в таблицу исходных строк
    for( line=0; line<TableLen; line++ )
    {
        // Цикл чтения исходных строк
        FIn >> KeyWord;
        if( !FIn ) // Обработка ошибки чтения
        {
            cout << endl << "Error 150. A read error";

```

```

        exit( 150 );
    }

    // Поиск индекса "i" строки таблицы для ее заполнения
    // Пока индекс не найден
    found = false;
    i = Hash( KeyWord );
    while( !found )
    {
        if( pTable[ i ].Key[ 0 ] == ' ' )
            // Индекс найден
            found = true;
        else
        {
            // КОНФЛИКТ - шаг по таблице
            i++; i = ( i > ( size-1 ) ? 0 : i );
        }
    }

    // Чтение данного
    FIn >> pTable[ i ].Data;
    if( !FIn )
    {
        cout << endl << "Error 160. A read error ";
        exit( 160 );
    }

    // Занесение ключа в строку "i" таблицы
    strcpy( pTable[ i ].Key, KeyWord );
}

// Закрытие файла ввода
FIn.close_f( FName, 170 );

return;
}

// *****
// Поиск в хэш-таблице
template< unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE< LenKey, LenData > :: HashSearch(
    // Ключевое слово для поиска
    char    KeyWord[ ],
    bool    &found,      // true - нашли
    unsigned int
        &IdxLine ) // Индекс найденной строки в таблице
{

```

```

    unsigned int
        i;           // Индекс строки таблицы
    bool    EndTab;   // true - достигли свободной строки

    // Подготовка к поиску
    found = false; EndTab = false; i = Hash( KeyWord );

    // Поиск в таблице
    while( !found && !EndTab )
    {
        if( pTable[ i ].Key[ 0 ] == ' ' )
            // Достигли свободной строки
            EndTab = true;
        else
        {
            if( !strcmp( pTable[ i ].Key, KeyWord ) )
            {
                // Нашли
                found = true; IxLine = i;
            }
            else
            {
                // Шаг по таблице
                i++; i = ( i>( size-1 )?0:i );
            }
        }
    }

    return;
}

#endif

```

Листинг 10.2. Файл TestSearch.cpp

```

/*
    Поиск в таблице с использованием шаблонного класса SEARCH_TABLE, объявление
    которого приведено во включаемом файле SearchT.h. Для открытия-закрытия файлов
    используется класс IOFILE, определение которого приведено во включаемом файле
    IOFile.h.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6
*/

#include "SearchT.h"           // Шаблонный класс для поиска в таблице

// Тестирование
int main( void )              // Возвращает 0 при успехе
{

```

```

// Создаем таблицу из 4 строк: 8 - длина ключа, 62 - длина данного
//   в строке таблицы
SEARCH_TABLE< 8, 62 >
    t( 4 );

bool    found;    // true - нашли ключевое слово в таблице
unsigned int
    IxLine;    // Индекс найденной строки

// Заполняем таблицу для последовательного поиска и печатаем ее
t.SeqInpTab( );
t.PrintTab( "SearchT.out", ios::out );
IOFILE    FOut;    // Файловый объект для вывода
// Вывод заголовка
FOut.open_f( "SearchT.out", ios::out | ios::app, 180 );
FOut << "\n    Тестирование последовательного поиска" << endl;
FOut.close_f( "SearchT.out", 190 );
// Последовательный поиск и его результаты
t.SequentialSearch( "and", found, IxLine );
t.PrintSearch( "and", found, IxLine );
t.SequentialSearch( "word", found, IxLine );
t.PrintSearch( "word", found, IxLine );

// Заполняем таблицу для логарифмического поиска и печатаем ее
t.LogInpTab( ); t.PrintTab( );
// Вывод заголовка
FOut.open_f( "SearchT.out", ios::out | ios::app, 200 );
FOut << "\n    Тестирование логарифмического поиска" << endl;
FOut.close_f( "SearchT.out", 210 );
// Логарифмический поиск и его результаты
t.LogarithmSearch( "and", found, IxLine );
t.PrintSearch( "and", found, IxLine );
t.LogarithmSearch( "word", found, IxLine );
t.PrintSearch( "word", found, IxLine );

// Заполняем таблицу для хэш-поиска и печатаем ее
t.HashInpTab( ); t.PrintTab( );
// Печать заголовка
FOut.open_f( "SearchT.out", ios::out | ios::app, 220 );
FOut << "\n    Тестирование хэш-поиска" << endl;
FOut.close_f( "SearchT.out", 230 );
// Хэш-поиск и его результаты
t.HashSearch( "work", found, IxLine );
t.PrintSearch( "work", found, IxLine );
t.HashSearch( "type", found, IxLine );
t.PrintSearch( "type", found, IxLine );

return 0;
}

```

Листинг 10.3. Результаты работы программы, выводимые в файл на магнитном диске

Состояние таблицы:

```
call   вызов
type   тип
word   слово
work   работа
```

Тестирование последовательного поиска

Результаты поиска для ключевого слова: and
Строка с ключом "and" в таблице не найдена

Результаты поиска для ключевого слова: word
Индекс строки в таблице: 2. Найденная строка:
word слово

Состояние таблицы:

```
word   слово
type   тип
work   работа
call   вызов
```

Тестирование логарифмического поиска

Результаты поиска для ключевого слова: and
Строка с ключом "and" в таблице не найдена

Результаты поиска для ключевого слова: word
Индекс строки в таблице: 0. Найденная строка:
word слово

Состояние таблицы:

```
call   вызов

type   тип
```

Тестирование хэш-поиска

Результаты поиска для ключевого слова: work
Строка с ключом "work" в таблице не найдена

Результаты поиска для ключевого слова: type
Индекс строки в таблице: 2. Найденная строка:
type тип

Обратите внимание на файл `TestSearch.cpp`, в котором выполняется тестирование спроектированного класса. Методы этого класса вызываются с количеством аргументов, меньшим или равным числу параметров соответствующих методов, причем в некоторых из них аргументы не используются вообще. Это возможно потому, что большинство методов используют параметры с умалчиваемыми значениями. Так, в вызове метода `t.PrintTab()` аргументы отсутствуют — вместо них используются умалчиваемые значения параметров. Такой вызов метода `t.PrintTab()` эквивалентен следующему вызову метода с полным списком аргументов:

```
t.PrintTab( "SearchT.out", ios::app,  
           "             Состояние таблицы:");
```

ЗАМЕЧАНИЕ

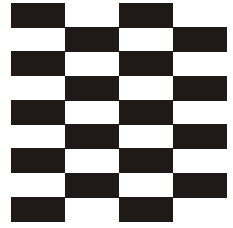
В методах класса, как и в обычных функциях, можно использовать параметры с умалчиваемыми значениями. Пользуйтесь такой возможностью — это очень удобно!

10.3. Упражнения для самопроверки

1. В программе из *разд. 10.2* модифицируйте тестирующий файл `TestSearch.cpp` таким образом, чтобы вызовы методов шаблонного класса использовали полные списки аргументов и были полностью эквивалентны вызовам из *разд. 10.2*.
2. Программу из *разд. 10.2* в учебных целях модифицируйте таким образом, чтобы использовалась иерархия шаблонных классов. В шаблонном базовом классе используйте конструктор, деструктор и метод для печати строк таблицы. В шаблонном производном классе для компактности программного кода оставьте только конструктор; метод заполнения таблицы для хэш-поиска; методы, реализующие хэш-поиск и печать результатов поиска.
3. Программу из упражнения 2 модифицируйте так, чтобы в поле ключа можно было использовать только русские строчные буквы.

Ответы на эти упражнения приведены в *разд. П1.9 приложения 1*.

Глава 11



Списки. Очереди и стеки

Обобщенный связанный список представляет собой структуру данных, в которой элементы содержат явные указания на связи между собой. Как указывалось ранее [3], списки являются наиболее *эффективными* структурами данных, если часто требуется выполнять операции добавления или исключения элементов.

В качестве простых примеров списков можно назвать рассмотренные в [3] две разновидности списков:

- ❑ классический однонаправленный не кольцевой линейный список (см. гл. 1, 2 и в [3] разд. 8);
- ❑ специализированный не кольцевой линейный список на базе массива структур, в котором хранится информация о наилучшем пути в графе (транспортная задача, см. гл. 9).

Основными средствами обработки списков в языках C/C++ являются *указатели*.

Существует множество способов реализации списков.

- ❑ *Линейные списки* (однонаправленные не кольцевые списки). На базе линейного списка можно эффективно реализовать динамический стек.
- ❑ *Циклические списки* (однонаправленные кольцевые списки). В таком списке поле `next` последнего элемента содержит указатель назад на первый элемент (рис. 11.1).

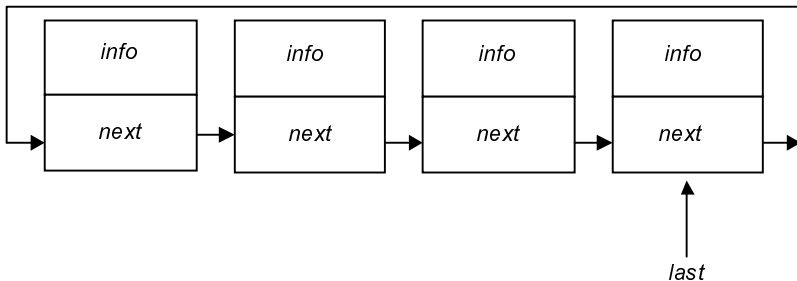


Рис. 11.1. Циклический список

Такие списки не имеют первого и последнего элементов. При работе с циклическим списком удобно использовать указатель на последний элемент. Это позволяет более

эффективно, по сравнению с кольцевым однонаправленным линейным списком, реализовать операции добавления или удаления элемента из конца списка (не нужно продвигаться по всему списку для получения указателя на последний элемент). Вместе с тем, аналогичные операции с началом списка остаются почти такими же эффективными, так как указатель на начало списка легко получить (*last* → *next*).

- **Двунаправленные списки.** Каждый элемент двунаправленного списка содержит два указателя: один указывает на предшествующий элемент, а другой — на последующий. Такие списки также могут быть линейными и циклическими (рис. 11.2). Отличительной особенностью подобных списков является более эффективная реализация операций вставки-удаления элементов. Двунаправленные линейные списки используют обычно два указателя — на левый и правый концы списка. На базе двунаправленного линейного списка можно реализовать, например, универсальную динамическую очередь.

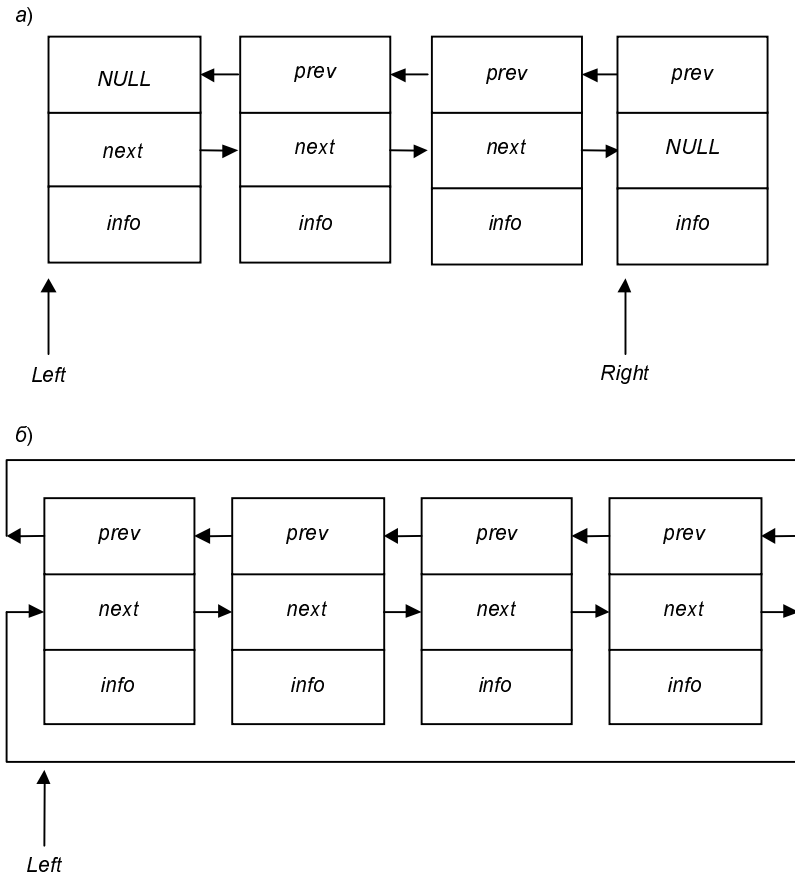


Рис. 11.2. Двунаправленные списки: а) линейный; б) циклический

□ *Мультисписки* — структуры данных, состоящие из элементов, которые могут принадлежать нескольким спискам, не повторяясь в нескольких экземплярах. Такие элементы имеют указатели для каждого списка, которому они принадлежат. Списки в мультисписке могут быть линейными и циклическими, однонаправленными и двунаправленными.

11.1. Бинарные деревья

Бинарные деревья кратко рассмотрены в [3], в *разд. 14.2*. С бинарными деревьями мы уже имели дело при сортировке массивов (сложная сортировка выбором) и при поиске в таблице (логарифмический поиск). Рассмотрим бинарные деревья подробнее и сделаем некоторые обобщения и дополнения.

Бинарное дерево — это конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый *корнем* дерева, а остальные элементы множества делятся на два непересекающихся подмножества, каждое из которых само является бинарным деревом. Эти подмножества называются *левым* и *правым поддеревьями* исходного дерева. Общепринятый способ изображения бинарного дерева представлен на рис. 11.3.

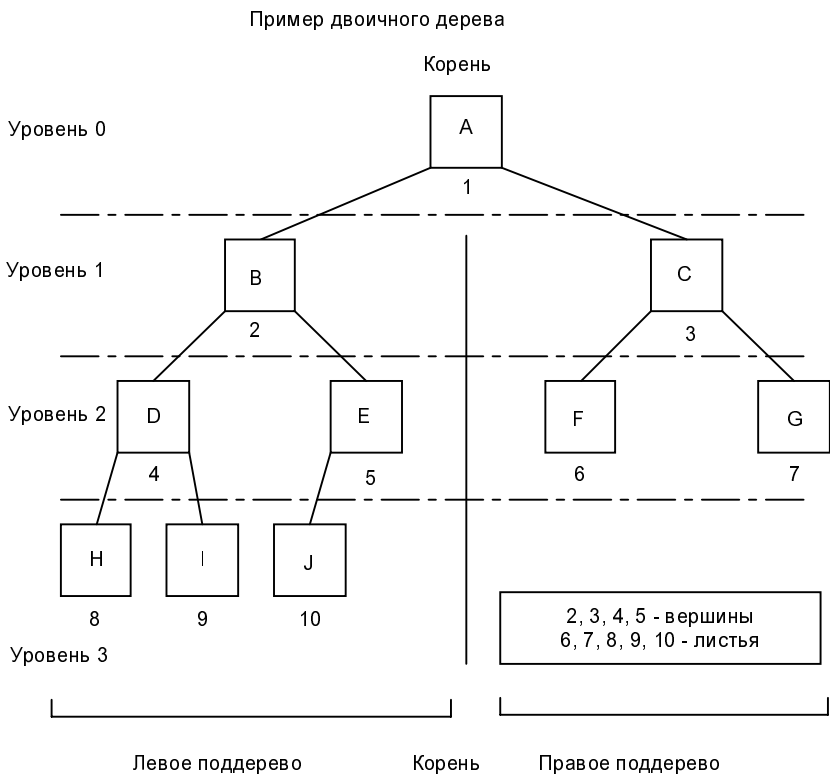


Рис. 11.3. Бинарное дерево

Если A — корень бинарного дерева, а B и C — соответственно корень его левого и правого поддеревья, то говорят, что A является *отцом* B и C , а B и C являются *левым и правым сыновьями*. Два узла являются *братьями*, если они сыновья одного и того же отца. Узлы, не имеющие сыновей (узлы H , I , J , F и G на рис. 11.3), называются *листьями* дерева. Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правое и левое поддеревья, то дерево называется *строго бинарным деревом*.

Уровень узла в бинарном дереве определен следующим образом: корень дерева имеет уровень 0, а уровень любого другого узла дерева на 1 больше уровня своего отца. *Глубина* бинарного дерева — это максимальный уровень листа дерева, что равно длине самого большого пути от корня к листу дерева.

Графическое представление (рис. 11.3) наглядно, и им удобно пользоваться при работе с бинарными деревьями. Однако следует помнить, что память компьютера линейна и с этой точки зрения можно считать, что бинарное дерево представляет собой *разновидность связанного списка*. Поэтому основные операции над бинарными деревьями во многом те же самые, что и для списков. Элементы дерева можно добавлять, удалять, а также осуществлять к ним доступ различными способами. Рассмотрение этих вопросов выходит за рамки данного курса. Вместе с тем, рекомендуем читателям в качестве упражнения спроектировать программы для сложной сортировки массива выбором и для логарифмического поиска в таблице, используя бинарное дерево на базе связанного списка.

Далее рассмотрим решение ряда практически значимых задач с использованием связанных списков. В соответствующих программных проектах будем использовать средства объектно-ориентированного программирования.

11.2. Очереди и стеки

Краткая характеристика очередей и их частных разновидностей приведена в [3] в *разд. 14.3*. Понятие очереди всем хорошо знакомо из повседневной жизни. Элементами очереди в общем случае являются заказы на то или иное обслуживание: выбить чек на нужную сумму в кассе магазина, получить нужную информацию в справочном бюро и т. д.

В программировании также имеются структуры данных, называемые *очередями*. Они используются, например, для моделирования реальных очередей с целью определения их характеристик (средняя длина очереди, время пребывания заказа в очереди, вероятность отказа в постановке в очередь при ограничении ее максимальной длины и т. п.) при данном законе поступления заказов и дисциплине их обслуживания. Ясно, что структуры данных, используемые для этой цели, должны отражать специфику моделируемых объектов, т. е. реальных очередей.

Заметим, что по своему существу очередь является сугубо динамическим объектом — с течением времени и длина очереди, и набор образующих ее элементов изменяются. Заметим также, что структура данных, называемая *очередью*, используется и как средство программирования при решении различных задач, в том числе и не связанных с реальными очередями. Примером такого рода является, например, *стек*.

Над очередью определены три основных операции:

- занесение элемента (заказа на обслуживание) в очередь;
- выбор элемента из очереди (для его обслуживания);
- просмотр элементов очереди.

При выборе элемента из очереди, естественно, он исключается из очереди. В общем случае, в очереди доступны два конца: левый (из него может выбираться или в него может заноситься очередной элемент) и правый конец очереди с аналогичными возможностями.

В частном случае, очередь может иметь только один конец. Очередь с одним концом и дисциплиной обслуживания, при которой на обслуживание первым выбирается тот элемент очереди, который поступил в нее последним, в программировании называют *стеком* (stack). Стек — это одна из наиболее употребительных структур данных, которая оказывается весьма удобной при решении различных задач. Дисциплина обслуживания, принятая в стеке, называется LIFO (Last Input — First Output, т. е. "последний внесен — первый извлечен").

Очередь может также использовать дисциплину обслуживания, называемую FIFO (First Input — First Output, т. е. "первый внесен — первый извлечен"). Такая очередь использует два конца — с одного из них элемент помещается в очередь, а с другого — извлекается из очереди (рис. 11.4).

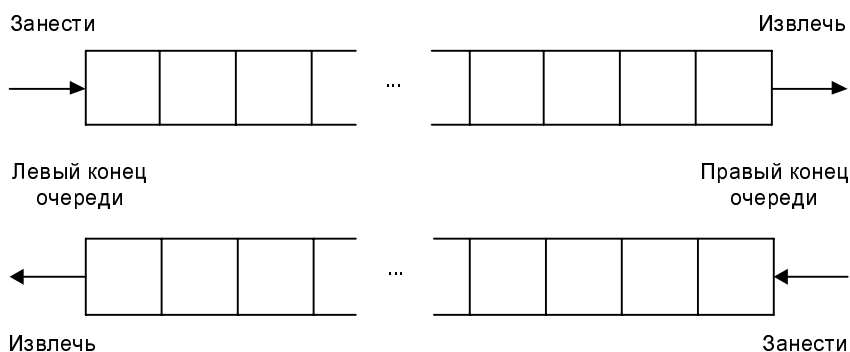


Рис. 11.4. Очередь FIFO

В общем случае, для очереди используются как ее левый, так и ее правый концы без ограничений на извлечение элемента или занесение элемента в очередь. Все остальные виды очереди могут быть получены из общего случая как частные ограниченные варианты (табл. 11.1).

Каждую из перечисленных в таблице очередей можно реализовать либо как очередь ограниченного размера на базе массива ([3], *разд. 14.4*), либо как очередь неограниченного размера на базе однонаправленного или двунаправленного линейного списка. Вначале рассмотрим оба варианта реализации универсальной очереди и сопоставим их друг с другом.

Таблица 11.1. Универсальная очередь и ее ограниченные варианты

Вид очереди	Операции с очередью				
	Занести слева	Занести справа	Извлечь слева	Извлечь справа	Состояние очереди
Неограниченная (универсальная) очередь	+	+	+	+	+
Ограниченная очередь с двумя концами с занесением с одного конца:					
а)	+	–	+	+	+
б)	–	+	+	+	+
Ограниченная очередь с двумя концами с извлечением с одного конца:					
а)	+	+	+	–	+
б)	+	+	–	+	+
Очередь FIFO — ограниченная очередь с двумя концами, с занесением с одного, а извлечением с другого конца очереди:					
а)	+	–	–	+	+
б)	–	+	+	–	+
Стек — очередь LIFO, ограниченная очередь с одним концом:					
а)	+	–	+	–	+
б)	–	+	–	+	+

11.2.1. Универсальная очередь неограниченного размера

Достоинством такой очереди является *отсутствие* ограничений на ее размер. Универсальная очередь неограниченного размера строится на базе двунаправленного линейного списка с размещением элементов в динамической памяти. Ее некоторым недостатком можно считать *повышенный расход памяти* по сравнению с универсальной очередью ограниченного размера на базе массива. Поскольку однонаправленный линейный список был подробно рассмотрен ранее (гл. 1 и 2) и в [3] (разд. 8), то далее приведем только прокомментированный текст программы, обеспечивающей

работу с универсальной очередью неограниченного размера (листинги 11.1, 11.2). Небольшими отличиями приводимого далее программного проекта являются использование вместо однонаправленного списка двунаправленного неколецевого линейного списка, оформление класса в виде шаблонного класса и сужение выполняемых над списком операций. Следствием этого является использование в шаблонном классе не одного указателя, а двух — указателя на правый и левый концы очереди. Отметим, что использование шаблона классов делает возможным хранение в очереди данных любых типов.

Листинг 11.1. Файл UniQueue.h

```
/*
    Содержит стандартные и нестандартные включаемые файлы, определение структуры для
    элемента очереди и определение шаблона классов для работы с универсальной очередью
    неограниченного размера (динамической очередью).

    В шаблоне классов реализованы следующие операции:
    * инициализация очереди (конструктор);
    * разрушение очереди с освобождением занятой динамической памяти (деструктор);
    * занесение элемента в очередь с левого конца;
    * занесение элемента в очередь с правого конца;
    * извлечение элемента с левого конца очереди;
    * извлечение элемента с правого конца очереди;
    * печать состояния очереди с использованием указателя на левый конец очереди;
    * печать состояния очереди с использованием указателя на правый конец очереди.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предотвращение многократного включения данного файла
#ifndef __UniQueue_H
#define __UniQueue_H

    // Класс для открытия-закрытия файлов на базе стандартного класса
    // fstream и подключение перегруженных операций << и >>. Файл
    // IOFile.h приведен в разд. 5.6
    #include "IOFile.h"

    // Объявление шаблонного класса для работы с универсальной очередью
    // неограниченного размера (T - тип значений, хранимых в очереди)
    template< class T >
    class UniversalQueue
    {
        // Локальные типы

        struct ELEM          // Структура для элемента очереди
        {
            T data;          // Данное, хранимое в элементе
```

```

        ELEM
        *pRight; // Указатель на соседа справа
        ELEM
        *pLeft;  // Указатель на соседа слева
    };

    // Данные

protected:

    ELEM *pRight; // Указатель на правый конец очереди
    ELEM *pLeft;  // Указатель на левый конец очереди

    // Методы

public:

    // Конструктор (подставляемый метод)
    UniversalQueue( void )
    {
        // Вначале очередь пуста
        pRight = pLeft = NULL;
    }

    // Деструктор
    ~UniversalQueue( void );

    // Добавление элемента в правый конец очереди
    void AddRight( const T &add );

    // Добавление элемента в левый конец очереди
    void AddLeft( const T &add );

    // Извлечение элемента из левого конца очереди
    bool OutLeft( T &out );

    // Извлечение элемента из правого конца очереди
    bool OutRight( T &out );

    // Печать содержимого очереди с использованием указателя на левый
    // конец
    void LeftPrint( char FileNameOut[ ] = "Queue.out",
        int mode = ios::app, char Header[ ] = "" );

    // Печать содержимого очереди с использованием указателя на
    // правый конец

```

```

    void RightPrint( char FileNameOut[ ] = "Queue.out",
        int mode = ios::app, char Header[ ] = "" );

};                                     // Конец объявления шаблонного класса

// *****
// Деструктор
template< class T >
UniversalQueue< T > :: ~UniversalQueue( void )
{
    ELEM    *del;        // Указатель на удаляемый элемент

    while( pLeft )
    {
        del = pLeft; pLeft = pLeft->pRight; delete del;
    }

    pRight = NULL;
}

// *****
// Добавление элемента в правый конец очереди
template< class T >
void UniversalQueue< T > :: AddRight(
    const T
        &add )        // Данное для добавляемого элемента
{
    // Указатель на добавляемый элемент очереди
    ELEM    *temp;

    temp = new ELEM; // Динамическое размещение элемента очереди
    if( !temp )
    {
        cout << "\n Error 10. The unit of the queue is not placed "
            << endl;
        exit( 10 );
    }

    // Занесение данного для хранения
    temp->data = add;

    // Новый элемент является последним
    temp->pRight = NULL;

    // Связываем новый элемент с остальной частью очереди
    if( !pRight )

```

```

        {
            // Очередь была пуста
            pRight = pLeft = temp; temp->pLeft = NULL;
        }
        else
        {
            // Очередь не была пуста
            temp->pLeft = pRight;
            pRight->pRight = temp; pRight = temp;
        }

        return;
    }

// *****
// Добавление элемента в левый конец очереди
template< class T >
void UniversalQueue< T > :: AddLeft(
    const T
        &add )    // Данное для добавляемого элемента
{
    // Указатель на добавляемый элемент очереди
    ELEM *temp;

    temp = new ELEM; // Динамическое размещение элемента очереди
    if( !temp )
    {
        cout << "\n Error 20. The unit of the queue is not placed "
            << endl;
        exit( 20 );
    }

    // Занесение данного для хранения
    temp->data = add;

    // Новый элемент является первым
    temp->pLeft = NULL;

    // Связываем новый элемент с остальной частью очереди
    if( !pLeft )
    {
        // Очередь была пуста
        pRight = pLeft = temp; temp->pRight = NULL;
    }
    else
    {
        // Очередь не была пуста
        temp->pRight = pLeft;
        pLeft->pLeft = temp; pLeft = temp;
    }
}

```

```

    return;
}

// *****
// Извлечение элемента из левого конца очереди
template< class T >
bool UniversalQueue< T > :: OutLeft(
    // false - извлечение не выполнено
    T      &out )    // Значение, которое извлекли
{
    ELEM   *del;     // Указатель на удаляемый элемент

    // Очередь пуста?
    if( !pLeft )
    {
        return false;
    }

    // В очереди один элемент?
    if( !pLeft->pRight )
    {
        out = pLeft->data; delete pLeft;
        pLeft = pRight = NULL;
        return true;
    }

    // В очереди больше, чем один элемент
    del = pLeft; out = pLeft->data;
    pLeft = pLeft->pRight; delete del;
    pLeft->pLeft = NULL;

    return true;
}

// *****
// Извлечение элемента из правого конца очереди
template< class T >
bool UniversalQueue< T > :: OutRight(
    // false - извлечение не выполнено
    T      &out )    // Значение, которое извлекли
{
    ELEM   *del;     // Указатель на удаляемый элемент

    // Очередь пуста?
    if( !pRight )

```

```

    {
        return false;
    }

    // В очереди один элемент?
    if( !pRight->pLeft )
    {
        out = pRight->data; delete pRight;
        pLeft = pRight = NULL;
        return true;
    }

    // В очереди больше, чем один элемент
    del = pRight; out = pRight->data;
    pRight = pRight->pLeft; delete del;
    pRight->pRight = NULL;

    return true;
}

// *****
// Печать содержимого очереди с использованием указателя на левый
// конец
template< class T >
void UniversalQueue< T > :: LeftPrint(
    // Файл вывода
    char    FileNameOut[ ],
    int     mode,      // Режим открытия файла
    // Заголовок для печати
    char    Header[ ] )
{
    IOFILE FOut;      // Файловый объект для вывода

    // Открытие файла
    FOut.open_f( FileNameOut, mode, 30 );

    // Вывод заголовка
    FOut << Header << endl;

    // Печать содержимого очереди
    ELEM    *pCur = pLeft;
    if( !pCur )
    {
        FOut << "Предупреждение 40. Очередь пуста " << endl;
    }
    while( pCur )
    {

```

```

        FOut << pCur->data << endl;
        pCur = pCur->pRight;
    }

    // Закрытие файла
    FOut.close_f( FileNameOut, 50 );

    return;
}

// *****
// Печать содержимого очереди с использованием указателя на правый
// конец
template< class T >
void UniversalQueue< T > :: RightPrint(
    // Файл вывода
    char    FileNameOut[ ],
    int     mode,      // Режим открытия файла
    // Заголовок для печати
    char    Header[ ] )
{
    IOFILE FOut;      // Файловый объект для вывода

    // Открытие файла
    FOut.open_f( FileNameOut, mode, 60 );

    // Вывод заголовка
    FOut << Header << endl;

    // Печать содержимого очереди
    ELEM    *pCur = pRight;
    if( !pCur )
    {
        FOut << "Предупреждение 70. Очередь пуста " << endl;
    }
    while( pCur )
    {
        FOut << pCur->data << endl;
        pCur = pCur->pLeft;
    }

    // Закрытие файла
    FOut.close_f( FileNameOut, 80 );

    return;
}

#endif

```

Листинг 11.2. Файл TestUniqueQueue.cpp

```

/*
    Работа с универсальной очередью неограниченного размера с использованием
    шаблонного класса UniversalQueue, определенного в файле UniqueQueue.h. Для открытия-
    закрытия файлов используется класс IOFILE, определение которого приведено во
    включаемом файле IOFile.h.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Стандартные и нестандартные заголовочные файлы, структура элемента
// очереди и определение шаблона классов UniversalQueue для работы с
// универсальной очередью неограниченного размера
#include "UniqueQueue.h"

// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    // Определение uq классом UniversalQueue< char >
    UniversalQueue< char >
        uq;

    char        ch;          // Для извлеченного элемента
    IOFILE      FOut;        // Файловый объект для вывода

    // Тестирование работы с пустой очередью

    FOut.open_f( "UniqueQueue.out", ios::out, 100 );
    FOut << "Тестирование работы с пустой очередью" << endl << endl;
    // Извлечение из левого конца очереди
    if( !uq.OutLeft( ch ) )
        FOut << "Извлечение слева не выполнено - очередь пуста" << endl;
    else
        FOut << "Извлекли слева:" << ch << endl;
    // Извлечение из правого конца очереди
    if( !uq.OutRight( ch ) )
        FOut << "Извлечение справа не выполнено - очередь пуста" << endl;
    else
        FOut << "Извлекли справа:" << ch << endl << endl;
    FOut.close_f( "UniqueQueue.out", 110 );
    // Печать содержимого очереди с помощью правого указателя
    uq.RightPrint( "UniqueQueue.out", ios::out | ios::app,
        "Состояние очереди (используется правый указатель):" );
    // Печать содержимого очереди с помощью левого указателя
    uq.LeftPrint( "UniqueQueue.out", ios::out | ios::app,
        "Состояние очереди (используется левый указатель):" );
}

```

```
// Тестирование работы с непустой очередью

// Добавление в правый конец очереди '3'
uq.AddRight( '3' );
// Добавление в правый конец очереди '4'
uq.AddRight( '4' );
// Добавление в правый конец очереди '5'
uq.AddRight( '5' );
// Добавление в левый конец очереди '2'
uq.AddLeft( '2' );
// Добавление в левый конец очереди '1'
uq.AddLeft( '1' );
// Добавление в левый конец очереди '0'
uq.AddLeft( '0' );
// Извлечение из левого конца очереди
uq.OutLeft( ch );
// Извлечение из правого конца очереди
uq.OutRight( ch );
// Печать содержимого очереди с помощью правого указателя
uq.RightPrint( "Unique.out", ios::out | ios::app,
    "\nТестирование работы с непустой очередью\n"
    "\nЗанесли справа '3', '4', '5', слева '2', '1', '0',"
    "\n извлекли по одному элементу слева и справа."
    "\nСостояние очереди (используется правый указатель):" );
// Печать содержимого очереди с помощью левого указателя
uq.LeftPrint( "Unique.out", ios::out | ios::app,
    "Состояние очереди (используется левый указатель):" );

return 0;
}
```

Результаты тестирования шаблона классов для работы с универсальной очередью неограниченного размера приведены в листинге 11.3.

Листинг 11.3. Результаты работы программы

Тестирование работы с пустой очередью

```
Извлечение слева не выполнено - очередь пуста
Извлечение справа не выполнено - очередь пуста
Состояние очереди (используется правый указатель):
Предупреждение 70. Очередь пуста
Состояние очереди (используется левый указатель):
Предупреждение 40. Очередь пуста
```

Тестирование работы с непустой очередью

Занесли справа '3', '4', '5', слева '2', '1', '0',
извлекли по одному элементу слева и справа.

Состояние очереди (используется правый указатель):

4
3
2
1

Состояние очереди (используется левый указатель):

1
2
3
4

В [3] (см. разд. 14.4) рассмотрены основные аспекты, связанные с реализацией динамических структур с помощью массивов. В частности, на базе массива можно реализовать универсальную очередь. При этом используемый массив можно размещать в динамической памяти, и тогда мы получаем универсальную очередь ограниченного размера.

11.2.2. Универсальная очередь ограниченного размера

Универсальная очередь ограниченного размера, в отличие от универсальной очереди неограниченного размера, более экономно использует память. Очередь реализуется на базе массива и для каждого элемента очереди хранится только полезная информация. Связи между элементами универсальной очереди ограниченного размера и указателями на концы очереди определяются путем довольно остроумных манипуляций с индексами массива. В соответствии с [3], для реализации универсальной очереди ограниченного размера, наряду с самим массивом, достаточно использовать две переменных целого типа `Right` и `Left` — для хранения индексов элементов массива, являющихся началом (левый конец) и концом (правый конец) очереди. Кроме того, членами класса следует сделать максимальный размер очереди `MaxSize`, определяемый размером используемого массива и текущий размер очереди `Size`. Набор операций над универсальной очередью остается таким же, как в предыдущем разделе — меняется только их реализация. Рассмотрим иллюстрирующий пример, в котором операции над очередью также реализованы с помощью шаблонного класса (листинги 11.4, 11.5).

Листинг 11.4. Файл `UnQLimSz.h`

/*

Содержит стандартные и пользовательские включаемые файлы, а также определение шаблонного класса для работы с универсальной очередью ограниченного размера на базе массива.

В шаблоне классов реализованы следующие операции с очередью:

* инициализация очереди (конструктор);

```

* разрушение очереди с освобождением занятой динамической памяти (деструктор);
* занесение элемента с левого конца;
* занесение элемента с правого конца;
* извлечение элемента с левого конца;
* извлечение элемента с правого конца;
* печать состояния очереди с использованием указателя на левый конец очереди;
* печать состояния очереди с использованием указателя на правый конец очереди.

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/

// Предотвращение многократного включения данного файла
#ifndef __UnQLimSz_H
#define __UnQLimSz_H

    // Класс для открытия-закрытия файлов на базе стандартного класса
    //  fstream и подключение перегруженных операций << и >>. Файл
    //  IOFile.h приведен в разд. 5.6
    #include "IOFile.h"

    // Объявление шаблонного класса для работы с универсальной очередью
    //  ограниченного размера на базе массива (Т - тип значений,
    //  хранимых в очереди)
    template< class Т >
    class UnQueueLimSize
    {
        // Данные

    protected:

        Т      *pQueue; // Указатель на начало массива, используемого
                        //  очередью

        unsigned int
            Right,      // Индекс правого конца очереди
            Left,       // Индекс левого конца очереди
            MaxSize,    // Максимальный размер очереди
            Size;       // Текущий размер очереди

        // Методы

    public:

        // Конструктор
        UnQueueLimSize( unsigned int MaxSz );

        // Деструктор (подставляемая функция)
        ~UnQueueLimSize( void )

```

```

{
    delete [ ] pQueue;
}

// Добавление элемента в правый конец очереди
bool AddRight( const T &add );

// Добавление элемента в левый конец очереди
bool AddLeft( const T &add );

// Извлечение элемента из левого конца очереди
bool OutLeft( T &out );

// Извлечение элемента из правого конца очереди
bool OutRight( T &out );

// Печать содержимого очереди с использованием
// указателя на левый конец
void LeftPrint( char FileNameOut[ ] = "UniLimQ.out",
               int mode = ios::app, char Header[ ] = "" );

// Печать содержимого очереди с использованием указателя на
// правый конец
void RightPrint( char FileNameOut[ ] = "UniLimQ.out",
                int mode = ios::app, char Header[ ] = "" );

}; // Конец объявления шаблонного класса

// *****
// Конструктор
template< class T >
UnQueueLimSize< T > :: UnQueueLimSize(
    unsigned int
        MaxSz ) // Максимальный размер очереди
{
    // Проверяем допустимость заданного максимального размера очереди
    if( MaxSz<1 )
    {
        cout << "\n Error 10. The size of queue should exceed 1 "
              << endl;
        exit( 10 );
    }

    // Размещаем массив для очереди в динамической памяти
    pQueue = new T[ MaxSz ];
    if( !pQueue )

```

```

    {
        cout << "\n Error 20. Queue is not placed in dynamic of"
              " memory " << endl;
        exit( 20 );
    }

    // Инициализируем массив
    for( unsigned int i=0; i<MaxSz; i++ )
    {
        pQueue[ i ] = ( T )0;
    }

    // Инициализируем остальные данные
    MaxSize = MaxSz; Right = MaxSize-1; Left = Right;
    Size = 0;
}

// *****
// Добавление элемента в правый конец очереди
template< class T >
bool UnQueueLimSize< T > :: AddRight(
    // false - элемент не добавлен

    const T
        &add )    // Данное для добавляемого элемента
{
    // Очередь заполнена?
    if( Size>=MaxSize )
    {
        return false;
    }

    // Добавляем элемент в очередь
    Right = ( Right+1 ) % MaxSize; pQueue[ Right ] = add;
    Size++;

    return true;
}

// *****
// Добавление элемента в левый конец очереди
template< class T >
bool UnQueueLimSize< T > :: AddLeft(
    // false - элемент не добавлен

    const T
        &add )    // Данное для добавляемого элемента
{

```

```

    // Очередь заполнена?
    if( Size>=MaxSize )
    {
        return false;
    }

    // Добавляем элемент в очередь
    pQueue[ Left ] = add;
    Left = ( Left+MaxSize-1 ) % MaxSize; Size++;

    return true;
}

// *****
// Извлечение элемента из левого конца очереди
template< class T >
bool UnQueueLimSize< T > :: OutLeft(
    T      &out )    // false - извлечение не выполнено
                    // Значение, которое извлекли
{
    // Очередь пуста?
    if( !Size )
    {
        return false;
    }

    // Извлекаем элемент из очереди
    Left = ( Left+1 ) % MaxSize; out = pQueue[ Left ];
    Size--;

    return true;
}

// *****
// Извлечение элемента из правого конца очереди
template< class T >
bool UnQueueLimSize< T > :: OutRight(
    T      &out )    // false - извлечение не выполнено
                    // Значение, которое извлекли
{
    // Очередь пуста?
    if( !Size )
    {
        return false;
    }

```

```

    }

    // Извлекаем элемент из очереди
    out = pQueue[ Right ];
    Right = ( Right+MaxSize-1 ) % MaxSize;
    Size--;

    return true;
}

// *****
// Печать содержимого очереди с использованием указателя на левый
// конец
template< class T >
void UnQueueLimSize< T > :: LeftPrint(
    // Файл вывода
    char   FileNameOut[ ],
    int    mode,      // Режим открытия файла
    // Заголовок для печати
    char   Header[ ] )
{
    IOFILE FOut;      // Файловый объект для вывода

    // Открытие файла
    FOut.open_f( FileNameOut, mode, 30 );

    // Вывод заголовка
    FOut << Header << endl;

    // Печать содержимого очереди
    if( !Size )
    {
        FOut << "Предупреждение 40. Очередь пуста " << endl;
    }
    unsigned int
        i = Left;
    for( unsigned int k=1; k<=Size; k++ )
    {
        i = ( i+1 ) % MaxSize;
        FOut << pQueue[ i ] << endl;
    }
    FOut << endl;

    // Закрытие файла
    FOut.close_f( FileNameOut, 50 );

```

```

    return;
}

// *****
// Печать содержимого очереди с использованием указателя на правый
// конец
template< class T >
void UnQueueLimSize< T > :: RightPrint(
    // Файл вывода
    char   FileNameOut[ ],
    int    mode,          // Режим открытия файла
    // Заголовок для печати
    char   Header[ ] )
{
    IOFILE FOut;          // Файловый объект для вывода

    // Открытие файла
    FOut.open_f( FileNameOut, mode, 60 );

    // Вывод заголовка
    FOut << Header << endl;

    // Печать содержимого очереди
    if( !Size )
    {
        FOut << "Предупреждение 70. Очередь пуста " << endl;
    }
    unsigned int
        i = Right;
    for( unsigned int k=1; k<=Size; k++ )
    {
        FOut << pQueue[ i ] << endl;
        i = ( i+MaxSize-1 ) % MaxSize;
    }

    // Закрытие файла
    FOut.close_f( FileNameOut, 80 );

    return;
}

#endif

```

Листинг 11.5. Файл TestUnQLimSz.cpp

```

/*
    Работа с универсальной очередью ограниченного размера с использованием
    шаблонного класса UnQueueLimSize, определенного в файле UnQLimSz.h. Шаблон
    классов использует массив. Для открытия-закрытия файлов используется класс IOFILE,
    определение которого приведено во включаемом файле IOFile.h.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Стандартные, нестандартные заголовочные файлы и определение шаблона
// классов UnQueueLimSize для работы с универсальной очередью
// ограниченного размера
#include "UnQLimSz.h"

// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    // Определение uq классом UnQueueLimSize< int >
    UnQueueLimSize< int >
        uqlz( 4 );

    int      out;           // Для извлекаемого значения
    IOFILE    FOut;         // Файловый объект для вывода

    // Тестирование работы с пустой очередью

    // Открываем файловый объект для записи
    FOut.open_f( "UnQLimSz.out", ios::out, 90 );
    FOut << "Тестирование работы с пустой очередью" << endl
        << endl;

    // Извлечение из левого конца очереди
    if( !uqlz.OutLeft( out ) )
    {
        FOut << "Предупреждение 100. Очередь пуста - извлечение элемента"
            " \n из ее левого конца не выполнено " << endl;
    }
    else
    {
        FOut << " Из левого конца очереди извлечен элемент со значением "
            << out << endl;
    }

    // Извлечение из правого конца очереди
    if( !uqlz.OutRight( out ) )
    {
        FOut << "Предупреждение 110. Очередь пуста - извлечение элемента"
            " \n из ее правого конца не выполнено " << endl;
    }
}

```

```

}
else
{
    FOut << " Из правого конца очереди извлечен элемент со"
        " значением " << out << endl;
}
// Закрываем файловый объект
FOut.close_f( "UnQLimSz.out", 120 );
// Печать содержимого очереди с помощью правого указателя
uqlz.RightPrint( "UnQLimSz.out", ios::out | ios::app,
    "Состояние очереди (используется правый указатель):" );
// Печать содержимого очереди с помощью левого указателя
uqlz.LeftPrint( "UnQLimSz.out", ios::out | ios::app,
    "Состояние очереди (используется левый указатель):" );

// Тестирование работы с непустой очередью

// Открываем файловый объект для дозаписи
FOut.open_f( "UnQLimSz.out", ios::out | ios::app, 130 );
FOut << "Тестирование работы с непустой очередью" << endl << endl;
// Добавление в правый конец очереди
for( int i=3; i<5; i++ )
{
    if( !uqlz.AddRight( i ) )
    {
        FOut << "Предупреждение 140. Очередь заполнена - добавление"
            " \n элемента в правый конец очереди не выполнено "
            << endl;
    }
}
// Добавление в левый конец очереди
for( i=2; i>=0; i-- )
{
    if( !uqlz.AddLeft( i ) )
    {
        FOut << "Предупреждение 150. Очередь заполнена - добавление"
            " \n элемента в левый конец очереди не выполнено "
            << endl;
    }
}
// Извлечение из левого конца очереди
if( !uqlz.OutLeft( out ) )
{
    FOut << "Предупреждение 160. Очередь пуста - извлечение элемента"
        " \n из ее левого конца не выполнено " << endl;
}

```

```

else
{
    FOut << "Из левого конца очереди извлечен элемент со значением "
        << out << endl;
}
// Закрываем файловый объект
FOut.close_f( "UnQLimSz.out", 170 );
// Печать содержимого очереди с помощью правого указателя
uqlz.RightPrint( "UnQLimSz.out", ios::out | ios::app,
    "Занесли справа 3, 4, слева 2, 1, 0, извлекли один \n"
    " элемент слева (размер очереди 4 элемента).\n"
    "Состояние очереди (используется правый указатель):" );
// Печать содержимого очереди с помощью левого указателя
uqlz.LeftPrint( "UnQLimSz.out", ios::out | ios::app,
    " Состояние очереди (используется левый указатель):" );

return 0;
}

```

Результаты тестирования шаблона классов для универсальной очереди ограниченно-го размера приведены в листинге 11.6.

Листинг 11.6. Результаты работы программы

Тестирование работы с пустой очередью

```

Предупреждение 100. Очередь пуста - извлечение элемента
из ее левого конца не выполнено
Предупреждение 110. Очередь пуста - извлечение элемента
из ее правого конца не выполнено
Состояние очереди (используется правый указатель):
Предупреждение 70. Очередь пуста
Состояние очереди (используется левый указатель):
Предупреждение 40. Очередь пуста

```

Тестирование работы с непустой очередью

```

Предупреждение 150. Очередь заполнена - добавление
элемента в левый конец очереди не выполнено
Из левого конца очереди извлечен элемент со значением 1
Занесли справа 3, 4, слева 2, 1, 0, извлекли один
элемент слева (размер очереди 4 элемента).
Состояние очереди (используется правый указатель):
4
3
2

```

Состояние очереди (используется левый указатель) :

2
3
4

Советуем вам внимательно изучить этот проект, обратив внимание на *изящность* манипуляций с индексами массива. Приведем некоторые комментарии к программе. Универсальная очередь ограниченного размера, являющаяся обобщением всевозможных очередей, *может быть реализована в виде массива*. Это обеспечивает экономию памяти, по сравнению с универсальной очередью на базе двунаправленного линейного списка.

Инициализация (начальная подготовка) очереди. Инициализация очереди выполняется конструктором шаблона классов и состоит в размещении массива в динамической памяти, присвоении его элементам нулевых значений и задании начальных значений данных шаблона классов (рис. 11.5).

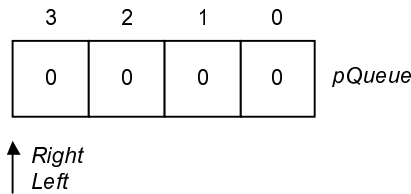


Рис. 11.5. Инициализация универсальной очереди ограниченного размера

Добавление элементов в правый конец очереди. Занесение элемента в очередь возможно только при $Size < MaxSize$. Состояния очереди после занесения в нее с правого конца элементов со значениями 3 и 4 показаны на рис. 11.6.

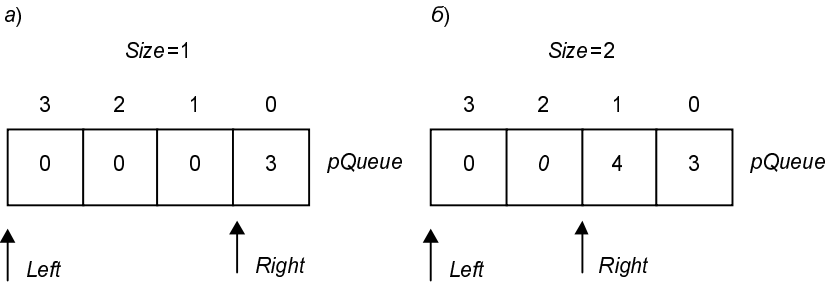


Рис. 11.6. Состояние очереди после занесения в нее с правого конца:
а) элемента со значением 3; б) элемента со значением 4

Добавление элементов в левый конец очереди. Занесение элемента в очередь также возможно только при $Size < MaxSize$. Состояния очереди после занесения в нее с левого конца элементов со значениями 2, 1 и 0 показаны на рис. 11.7 соответственно.

ПРИМЕЧАНИЕ

Обратите внимание, что элемент со значением 0 в очередь не будет помещен, так как к этому времени очередь уже будет заполнена.

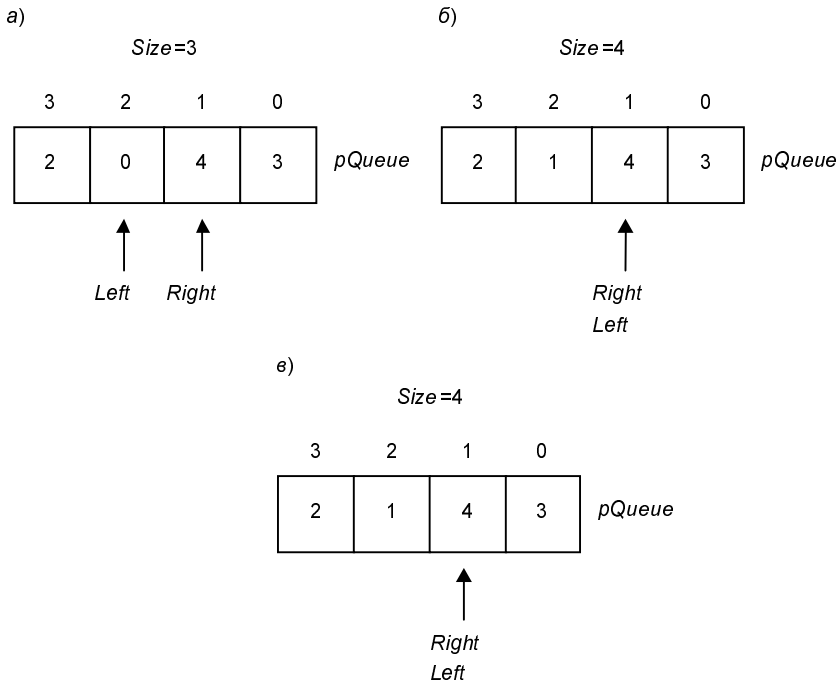


Рис. 11.7. Состояние очереди после занесения в нее с левого конца:
а) элемента со значением 2; б) элемента со значением 1; в) элемента со значением 0

Извлечение элемента из левого конца очереди. Извлечение элемента из очереди возможно только при $Size > 0$. Состояние очереди после извлечения из нее элемента с левого конца показано на рис. 11.8.

Печать содержимого очереди с использованием указателя на правый конец. При $Size = 0$ очередь пуста и печатается предупреждение об этом. Если же очередь не пуста, то для печати текущего состояния очереди можно использовать индекс *Right* элемента правого конца очереди. Следует заметить, что индекс *Right* указывает после занесения или извлечения элемента справа на элемент правого конца очереди. Сказанное иллюстрирует рис. 11.9.

Аналогичным образом выполняется печать содержимого очереди с использованием индекса элемента левого конца очереди. При этом имеются два отличия:

- ☐ перебор элементов очереди производится в противоположном порядке;
- ☐ индекс *Left* указывает после занесения или извлечения элемента слева на ближайший незаполненный элемент очереди.

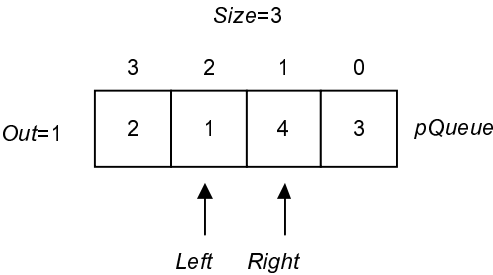


Рис. 11.8. Состояние очереди после извлечения элемента из ее левого конца

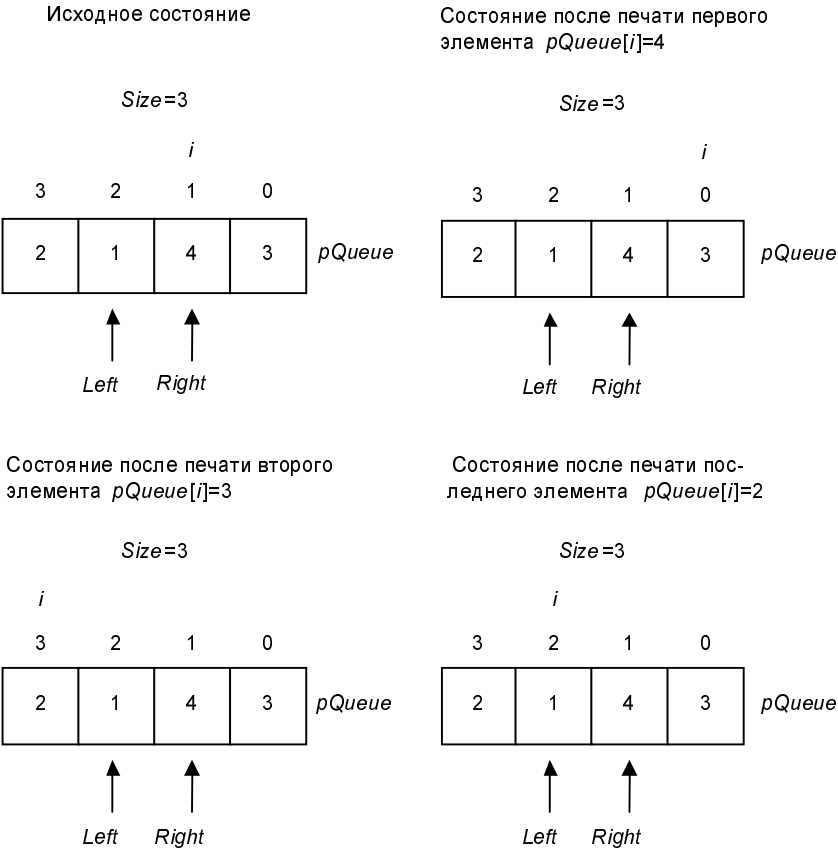


Рис. 11.9. Печать содержимого очереди с использованием указателя на правый конец

Заметим, что из двух рассмотренных реализаций универсальной очереди (очередь с неограниченным и ограниченным размерами) в качестве частных случаев легко получаются более простые варианты очередей, приведенные в табл. 11.1.

- Ограниченная очередь с двумя концами с занесением элемента только с одного конца и извлечением с обоих концов (из определения шаблонного класса универсальной очереди достаточно исключить либо операцию занесения элемента с левого конца, либо операцию занесения элемента с правого конца очереди).
- Ограниченная очередь с двумя концами с извлечением элемента только с одного конца и занесением элементов с обоих концов (из определения шаблонного класса универсальной очереди достаточно исключить либо операцию извлечения элемента с левого конца, либо операцию извлечения элемента с правого конца очереди).
- Очередь FIFO (из определения шаблонного класса универсальной очереди достаточно исключить либо операции извлечения элемента с левого конца и занесения с правого конца очереди, либо операции извлечения элемента с правого конца и занесения с левого конца очереди).

11.2.3. Динамический стек

Как уже указывалось, *стек* является частным случаем универсальной очереди, когда занесение элементов в очередь и извлечение их из очереди выполняются только с одного конца (очередь типа LIFO). Как и в случае универсальной очереди, стек можно реализовать либо на базе линейного списка (динамический стек неограниченного размера), либо на базе массива (стек ограниченного размера). Сопоставление этих вариантов реализации выполнено ранее. Стек ограниченного размера на базе массива неоднократно обсуждался в [3]. Поэтому рассмотрим только динамический стек и его особенности (листинги 11.7, 11.8). Обратите внимание на то, что для реализации динамического стека достаточно использовать обычный однонаправленный линейный список.

Листинг 11.7. Файл Stack.h

```
/*
    Содержит включаемые файлы, определение структуры для элемента стека
    и определение шаблонного класса для работы с динамическим стеком
    неограниченного размера.

    В шаблонном классе реализованы следующие операции со стеком:
    * инициализация стека (конструктор);
    * разрушение стека с освобождением занятой динамической памяти (деструктор);
    * занесение элемента в стек;
    * извлечение элемента из стека;
    * печать состояния стека.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Предотвращение многократного включения данного файла
#ifndef __Stack_H
```

```

#define __Stack_H

// Класс для открытия-закрытия файлов на базе стандартного класса
// fstream и подключение перегруженных операций << и >>. Файл
// IOFile.h приведен в разд. 5.6
#include "IOFile.h"

// Объявление шаблонного класса для работы с динамическим стеком
// неограниченного размера (T - тип значений, хранимых в стеке)
template< class T >
class STACK
{
    // Локальные типы

    struct ELEM        // Структура для элемента стека
    {
        T data;        // Данное, хранимое в элементе
        ELEM
            *next;      // Указатель на следующий элемент
    };

    // Данные

protected:

    ELEM *pTop;        // Указатель на вершину стека

    // Методы

public:

    // Конструктор (подставляемый метод)
    STACK( void )
    {
        // Вначале стек пуст
        pTop = NULL;
    }

    // Деструктор
    ~STACK( void );

    // Добавление элемента в стек
    void push( const T &add );

    // Извлечение элемента из стека
    bool pop( T &out );

```

```

        // Печать содержимого стека
        void PrintStack( char FileNameOut[ ] = "stack.out",
            int mode = ios::app, char Header[ ] = "" );

};                                     // Конец объявления шаблонного класса

// *****
// Деструктор
template< class T >
STACK< T > :: ~STACK( void )
{
    ELEM    *del;        // Указатель на удаляемый элемент

    while( pTop )
    {
        del = pTop; pTop = pTop->next; delete del;
    }
}

// *****
// Добавление элемента в стек
template< class T >
void STACK< T > :: push(
    const T
        &add )        // Данное для добавляемого элемента
{
    // Указатель на добавляемый элемент стека
    ELEM    *temp;

    temp = new ELEM; // Динамическое размещение элемента стека
    if( !temp )
    {
        cout << "\n Error 10. The unit of the stack is not placed "
            << endl;
        exit( 10 );
    }

    temp->data = add; // Занесение данного для хранения

    // Связываем новый элемент с остальной частью стека
    temp->next = pTop; pTop = temp;

    return;
}

```

```

// *****
// Извлечение элемента из стека
template< class T >
bool STACK< T > :: pop(
    // false - извлечение не выполнено
    T      &out )    // Значение, которое извлекли
{
    ELEM   *del;      // Указатель на удаляемый элемент

    // Стек пуст?
    if( !pTop )
    {
        return false;
    }

    // В стеке один элемент?
    if( !pTop->next )
    {
        out = pTop->data; delete pTop; pTop = NULL;
        return true;
    }

    // В стеке больше, чем один элемент
    del = pTop; out = pTop->data;
    pTop = pTop->next; delete del;

    return true;
}

// *****
// Печать содержимого стека
template< class T >
void STACK< T > :: PrintStack(
    // Файл вывода
    char   FileNameOut[ ],
    int    mode,      // Режим открытия файла
    // Заголовок для печати
    char   Header[ ] )
{
    IOFILE FOut;      // файловый объект для вывода

    // Открытие файла
    FOut.open_f( FileNameOut, mode, 20 );

    // Печать содержимого очереди
    ELEM   *pCur = pTop;

```

```

    if( !pCur )
    {
        FOut << "Предупреждение 30. Стек пуст " << endl;
    }
    else
    {
        // Вывод заголовка
        FOut << Header << endl;

        while( pCur )
        {
            FOut << pCur->data << endl;
            pCur = pCur->next;
        }

        // Закрытие файла
        FOut.close_f( FileNameOut, 40 );

        return;
    }

#endif

```

Листинг 11.8. Файл TestStack.cpp

```

/*
    Работа с динамическим стеком неограниченного размера с использованием шаблонного
    класса STACK, определенного в файле StackUniqeue.h. Для открытия-закрытия файлов
    используется класс IOFILE, определение которого приведено во включаемом файле
    IOFile.h.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Стандартные и нестандартные заголовочные файлы, структура элемента
// стека и определение шаблона классов STACK для работы с динамическим
// стеком неограниченного размера
#include "Stack.h"

// Тестирование
int main( void ) // Возвращает 0 при успехе
{
    // Определение s классом STACK< char >
    STACK< char >
        s;

    // Тестирование работы с пустым стеком

```

```

IOFILE      FOut;
// Открытие файлового объекта для записи
FOut.open_f( "Stack.out", ios::out, 50 );
FOut << "Тестирование работы с пустым стеком" << endl << endl;
// Извлечение из стека
char        ch;
if( !s.pop( ch ) )
{
    FOut << "Предупреждение 60. Стек пуст - извлечение\n элемента из"
          " него не выполнено " << endl;
}
else
{
    FOut << "Из стека извлечен элемент со значением " << ch << endl;
}
// Закрытие файлового объекта
FOut.close_f( "Stack.out", 70 );
// Печать содержимого стека
s.PrintStack( "Stack.out", ios::out | ios::app,
              "Состояние стека:" );
s.push( '2' );          // Добавление в стек '2'

// Тестирование работы с непустым стеком

s.push( '1' );          // Добавление в стек '1'
s.push( '0' );          // Добавление в стек '0'
// Извлечение из стека
s.pop( ch );

// Печать содержимого стека
s.PrintStack( "Stack.out", ios::out | ios::app,
              "\nТестирование работы с непустым стеком\n\n"
              "Занесли '2', '1', '0', извлекли '0'."
              "\nСостояние стека:" );

return 0;
}

```

Результаты тестирования шаблона классов для динамического стека неограниченно-го размера показаны в листинге 11.9.

Листинг 11.9. Результаты работы программы

Тестирование работы с пустым стеком

Предупреждение 60. Стек пуст - извлечение

элемента из него не выполнено
Предупреждение 30. Стек пуст

Тестирование работы с непустым стеком

Занесли '2', '1', '0', извлекли '0'.
Состояние стека:

1
2

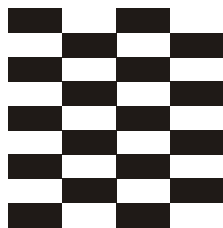
Приведенная программа тривиальна и не требует пояснений, так как является простым частным случаем универсальной очереди неограниченного размера.

11.3. Вопросы для самопроверки

1. Что собой представляют обобщенные связанные списки и когда их следует применять?
2. Перечислите разновидности списков.
3. На базе какой разновидности списков целесообразно реализовать динамический стек?
4. Укажите преимущества циклических списков по сравнению с линейными списками.
5. Какими преимуществами обладают двунаправленные списки?
6. Перечислите разновидности очередей.
7. Какие операции определены над очередью?

Ответы на эти вопросы приведены в разд. П1.10 приложения 1.

Глава 12



Сортировка файлов

Рассмотренные в [3] в *разд. 15* и в *гл. 8* данной книги достаточно эффективные алгоритмы сортировки массивов не применимы, если сортируемые данные имеют большой размер и не помещаются в оперативной памяти, а, например, расположены на *внешнем запоминающем устройстве* с последовательным доступом. В этом случае сортируемые данные представляются в виде *последовательного файла*, который характеризуется тем, что в каждый момент времени *в процессе перебора* имеется доступ к одному и только одному элементу данных. Это ограничение обуславливает необходимость применения других методов сортировки, отличных от методов сортировки массивов.

Основной метод сортировки файлов — это *слияние*, означающее объединение двух (или более) *упорядоченных* последовательностей в одну *упорядоченную* последовательность при помощи циклического выбора элементов, доступных в данный момент. Слияние — один из этапов сортировки файлов, используемый совместно с процессом, называемым *распределением*.

12.1. Сортировка файлов простым слиянием

Один из методов сортировки файлов слиянием, называемый *простым слиянием*, состоит в следующем:

1. Сортируемый файл *c* разбивается (распределяется) на две половины (два файла): файлы *a* и *b*. Этот процесс принято называть фазой распределения.
2. Файлы *a* и *b* сливаются снова в файл *c* при помощи объединения их отдельных элементов в упорядоченные пары (фаза слияния).
3. Шаги 1 и 2 (проходы сортировки) циклически повторяются, и на каждом проходе длина отсортированных фрагментов увеличивается в два раза. Так, на втором проходе получаются упорядоченные четверки элементов, на третьем проходе — упорядоченные восьмерки элементов и т. д. (рис. 12.1).

Для выполнения сортировки требуется три файла, и поэтому процесс сортировки называется *трехфайловым слиянием*.

Анализ сортировки слиянием. Поскольку на каждом проходе длина отсортированной последовательности удваивается, то сортировка требует $\log_2 n$ проходов и на каждом

проходе все множество из n элементов копируется ровно два раза. Следовательно, общее число пересылок

$$M = 2 * n * \log_2(n), \text{ т. е. } M \approx n * \log_2(n)$$

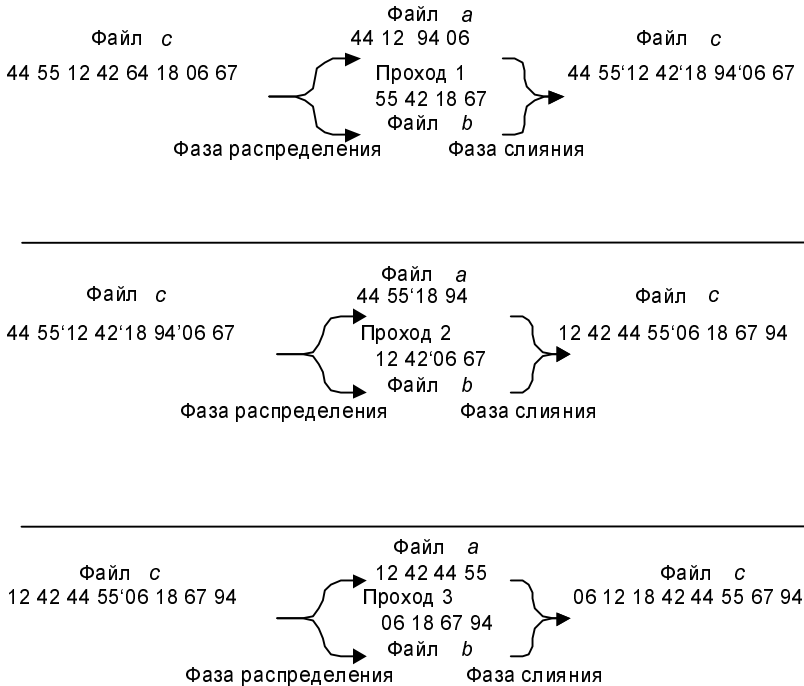


Рис. 12.1. Сортировка файла простым слиянием

При сортировке файлов стоимость операций пересылки (длительность) на несколько порядков превышает стоимость операции сравнения. Поэтому анализ числа сравнений при сортировке файлов не представляет практического интереса.

Показатель эффективности сортировки файлов простым слиянием соответствует показателям эффективности сложных сортировок массивов, но для массивов такая сортировка не приемлема, так как она не обеспечивает сортировки "на месте" (стоимость единицы оперативной памяти гораздо дороже, чем стоимость единицы внешней памяти).

12.2. Сортировка файлов естественным слиянием

В случае простого слияния мы ничего не выигрываем, если файл уже частично отсортирован. Фактически можно было бы сразу же сливать упорядоченные последовательности длиной m и l элементов в одну последовательность из $m + l$ элементов. Метод сортировки, при котором каждый раз сливаются две самые длинные возможные подпоследовательности, называется *естественным слиянием*.

Назовем подпоследовательность a_i, \dots, a_j такую, что

$$a_{i-1} > a_i \leq a_{i+1} \leq \dots \leq a_j > a_{j+1}$$

минимальной серией или, короче, *серией*. Сортировка файлов естественным слиянием объединяет не последовательности фиксированной, заранее заданной длины, а максимальные серии. Серии имеют то свойство, что при слиянии двух последовательностей, каждая из которых содержит r серий, создается одна последовательность, содержащая ровно r серий. Таким образом, на каждом проходе число серий уменьшается вдвое и число необходимых пересылок в худшем случае (когда серий нет — серии одиночной длины)

$$M = 2 * n * \log_2(n),$$

а в обычном случае даже меньше.

Пусть исходная последовательность элементов задана в виде файла c , который в конце работы должен содержать результат сортировки. Используются два вспомогательных файла a и b . Каждый проход состоит из фазы распределения, которая распределяет серии поровну из c в a и b , и из фазы слияния, которая сливает серии из a и b в c (рис. 12.2).

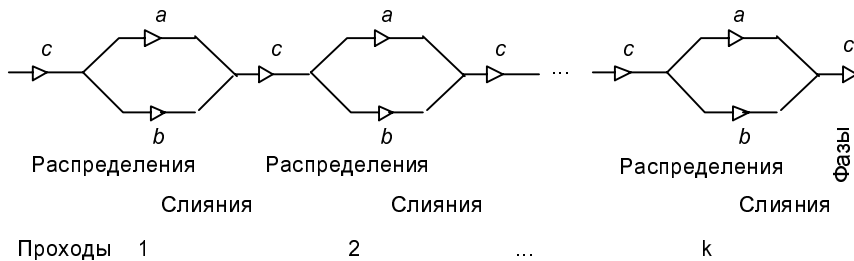


Рис. 12.2. Фазы сортировки файла

В качестве примера далее приведены результаты сортировки файла c естественным слиянием в исходном состоянии (строка 1) и после проходов 1—3 (строки 2—4):

```

17 31 ' 5 59 ' 13 41 43 67 ' 11 23 29 47 ' 3 7 71 ' 2 19 57 ' 37 61
5 17 31 59 ' 11 13 23 29 41 43 47 67 ' 2 3 7 19 57 71 ' 37 61
5 11 13 17 23 29 31 41 43 47 59 67 ' 2 3 7 19 37 57 61 71
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 57 59 61 67 71

```

В рассмотренном примере в естественном слиянии участвуют 20 чисел и требуются только 3 прохода, а не 5, как нужно было бы при простом слиянии. Сортировка заканчивается при числе серий в c , равном 1.

12.2.1. Спецификация класса для сортировки файлов

Как и при сортировке массивов, при проектировании спецификации класса необходимо решить следующие вопросы:

- выбрать целесообразную иерархию классов;

- ☐ определить, достаточным ли является использование обычных классов или следует применить шаблоны классов;
- ☐ спроектировать структуру каждого из классов иерархии (определить состав членов класса, их функциональное назначение и доступность);
- ☐ спроектировать файловую структуру класса (размещать ли целиком определения классов в заголовочных файлах или объявления классов помещать в заголовочные файлы, а реализацию методов классов — в файлы с расширением `cpp`).

Иерархия шаблонных или обычных классов. Поскольку сортировка файлов является хотя и важной, но все же частной задачей, являющейся одним из этапов более сложной задачи, в которой требуется использовать отсортированный файл, то представляется целесообразным реализовать сортировку файлов в *отдельном классе*. Этот класс можно использовать в качестве *базового* при решении задач, использующих отсортированные файлы.

При сортировке файлов обычно используется следующий набор операций: инициализация данных, удаление временных файлов по окончании сортировки (на рис. 12.2 они обозначены как a и b), печать содержимого файла, сортировка файла естественным слиянием. В свою очередь, для сортировки файла требуется выполнить ряд вспомогательных операций: распределение серий из исходного файла в два временных файла, слияние всех серий из двух временных файлов в исходный файл, слияние двух очередных серий из временных файлов в исходный файл, копирование серии из одного файла в другой, копирование элемента из одного файла в другой с проверкой достижения конца серии.

Анализируя этот набор операций, нетрудно заметить, что часть операций (инициализация данных, удаление временных файлов по окончании сортировки, печать содержимого файла, сортировка файла естественным слиянием) является *интерфейсной* и используется *явным образом*. Остальные операции являются *вспомогательными* и используются *неявным образом*. Вполне очевидно, что элементы сортируемых файлов могут быть различного типа. Поэтому для сортировки файлов следует использовать шаблонный класс. Структура класса определяется составом членов класса, их функциональным назначением и доступностью. В целях общности в качестве сортируемого файла будем рассматривать *файл*, элементы которого имеют следующий тип:

```
struct EL // Структура для сортируемого файла
{
    T key; // Ключ сортировки
    // Прочие поля элемента
};
```

Таким образом, шаблон базовых классов может иметь следующий вид:

```
// Объявление шаблонного класса для трехленточной двухфазной сортировки
//   файлов естественным слиянием (T - тип значений, хранимых
//   в сортируемом файле)
template< class T >
class MergeSortFile
{
    // Локальные типы

    struct EL              // Структура для сортируемого файла
```

```

{
    T      key;          // Ключ сортировки
    // Прочие поля элемента
};

// Данные

protected:

    // Указатель на имя сортируемого файла
    char      *pNameSortFile;

    // Методы

public:

    // Конструктор (подставляемая функция) - инициализация данных класса
    MergeSortFile(
        char      *pNSF )    // Указатель на имя сортируемого файла
    {
        pNameSortFile = pNSF;
    }

    // Деструктор (подставляемая функция) - удаление вспомогательных
    // файлов
    ~MergeSortFile( void );

    // Трехленточная двухфазная сортировка файла естественным слиянием
    void NaturalMerge( void );

    // Печать содержимого сортируемого файла
    void PrintFile( char FileNameOut[ ], char Header[ ] = "",
        int mode = ios::app );

private:

    // Распределение серий из файла с указателем pNameSortFile во
    // временные файлы a.f и b.f
    void Distribute( void );

    // Копирование одной серии из файлового объекта FInp в файловый
    // объект FOut
    void CopySeries( IOFILE &FInp, IOFILE &FOut );

    // Пересылка одного элемента из файлового объекта FInp в файловый
    // объект FOut с определением, достигнут ли в файловом объекте FInp

```

```

// конец серии
void Copy( IOFILE &FInp, IOFILE &FOut, bool &EOS );

// Слияние серий из временных файлов a.f и b.f в исходный файл
// с указателем pNameSortFile
void Merge( int &NumSer );

// Слияние двух серий из файловых объектов fa и fb в одну серию,
// передаваемую в файловый объект fc
void MergeSeries( IOFILE &fa, IOFILE &fb, IOFILE &fc );

};

```

Файловая структура класса. Ранее было обосновано и показано (см. *разд. 1.6*), что определение шаблона классов должно целиком размещаться в заголовочном файле.

12.2.2. Шаблон классов для сортировки файла естественным слиянием

Далее приводится один из вариантов реализации шаблона классов для сортировки файла и пример тестирующей программы (листинги 12.1, 12.2). Отличительной особенностью этого варианта является использование мощных и изящных средств ввода-вывода языка C++.

Листинг 12.1. Файл MergSort.h

```

/*
    Содержит включаемые файлы, определение структуры для элемента сортируемого
    файла и определение шаблона классов для трехленточной двухфазной сортировки файлов
    естественным слиянием.

```

В шаблоне классов реализованы следующие операции:

- инициализация (конструктор);
- сортировка файла естественным слиянием;
- распределение серий из исходного файла в два временных файла (вспомогательная операция);
- копирование серии из одного файла в другой (вспомогательная операция);
- копирование элемента из одного файла в другой с проверкой достижения конца серии (вспомогательная операция);
- слияние всех серий из двух временных файлов в исходный файл (вспомогательная операция);
- слияние двух очередных серий из временных файлов в исходный файл (вспомогательная операция);
- печать содержимого файла.

Давыдов В., консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/

```

```
// Предотвращение многократного включения данного файла
#ifndef __MergeSort_H
#define __MergeSort_H

// Класс для открытия-закрытия файлов на базе стандартного класса
// fstream и подключение перегруженных операций << и >>. Файл
// IOFile.h приведен в разд. 5.6
#include "IOFile.h"

#include <stdio.h> // Для функции удаления файла
#include <iomanip> // Для манипуляторов с параметрами

// Объявление шаблонного класса для трехленточной двухфазной
// сортировки файлов естественным сливанием (Т - тип значений,
// хранимых в сортируемом файле)
template< class T >
class MergeSortFile
{
    // Локальные типы

    struct EL // Структура для сортируемого файла
    {
        T key; // Ключ сортировки
        // Прочие поля элемента
    };

    // Данные

protected:

    // Указатель на имя сортируемого файла
    char *pNameSortFile;

    // Методы

public:

    // Конструктор (подставляемая функция)
    MergeSortFile(
        char
            *pNSF ) // Указатель на имя сортируемого файла
    {
        pNameSortFile = pNSF;
    }
}
```

```

// Деструктор (подставляемая функция)
~MergeSortFile( void )
{
    // Удаление вспомогательных файлов a.f и b.f
    if( _unlink( "a.f" ) || _unlink( "b.f" ) )
    {
        cout << "\n Error 10. The auxiliary file a.f"
              " or b.f was not remote " << endl;
        exit( 10 );
    }
}

// Трехленточная двухфазная сортировка файла естественным
// слиянием
void NaturalMerge( void );

// Печать содержимого сортируемого файла
void PrintFile( char FileNameOut[ ], char Header[ ] = "",
               int mode = ios::out | ios::app );

private:

// Распределение серий из файла с указателем pNameSortFile во
// временные файлы a.f и b.f
void Distribute( void );

// Копирование одной серии из файлового объекта FInp в файловый
// объект FOut
void CopySeries( IOFILE &FInp, IOFILE &FOut );

// Пересылка одного элемента из файлового объекта FInp в файловый
// объект FOut с определением, достигнут ли в файловом объекте
// FInp конец серии
void Copy( IOFILE &FInp, IOFILE &FOut, bool &EOS );

// Слияние серий из временных файлов a.f и b.f в исходный файл
// с указателем pNameSortFile
void Merge( int &NumSer );

// Слияние двух серий из файловых объектов fa и fb в одну серию,
// передаваемую в файловый объект fc
void MergeSeries( IOFILE &fa, IOFILE &fb,
                 IOFILE &fc );

}; // Конец шаблона классов MergeSortFile

```

```

// *****
// Печать содержимого сортируемого файла
template< class T >
void MergeSortFile< T > :: PrintFile(
    // Файл для вывода результатов
    char    FileNameOut[ ],
    // Заголовок для печати
    char    Header[ ],
    int     mode )    // Режим открытия файла
{
    IOFILE FOut,      // Файловый объект для вывода
           FIn;      // Файловый объект для ввода
    T      data;      // Для прочитанного элемента

    // Открытие файлов для ввода и вывода
    FOut.open_f( FileNameOut, mode, 20 );
    FIn.open_f( pNameSortFile, ios::in, 30 );

    // Вывод заголовка
    FOut << Header << endl;

    // Печать содержимого файла
    unsigned int
        i = 0;
    while( 1 )
    {
        FIn >> data;
        if( ( !FIn ) || ( FIn.eof( ) ) )
            break;
        if( !( i%4 ) )
            FOut << endl;
        FOut << setw( 20 ) << data;
        i++;
    }
    FOut << endl << endl;

    // Закрытие файлов
    FOut.close_f( FileNameOut, 40 );
    FIn.clear( );
    FIn.close_f( pNameSortFile, 50 );

    return;
}

// *****
// Трехленточная двухфазная сортировка файла естественным слиянием

```

```

template< class T >
void MergeSortFile< T > :: NaturalMerge( void )
{
    // Число серий в сортируемом файле: при числе серий 1 сортировка
    // закончена
    int    numser;

    // Сортировка
    do
    {
        // Фаза распределения
        Distribute( );
        numser = 0;
        // Фаза слияния с подсчетом числа сливаемых серий
        Merge( numser );
    } while( numser > 1 );

    return;
}

// *****
// Распределение серий из файла с указателем pNameSortFile во
// временные файлы a.f и b.f
template< class T >
void MergeSortFile< T > :: Distribute( void )
{
    IOFILE FC,          // файловый объект для чтения
           FA,          // файловый объект для записи
           FB;          // файловый объект для записи

    // Открываем файл с указателем pNameSortFile для чтения, а файлы
    // a.f и b.f - для записи
    FC.open_f( pNameSortFile, ios::in, 60 );
    FA.open_f( "a.f", ios::out, 70 );
    FB.open_f( "b.f", ios::out, 80 );

    // Распределение серий по вспомогательным файлам a.f и b.f
    do
    {
        // Копируем серию из файла с указателем pNameSortFile в файл
        // a.f
        CopySeries( FC, FA );
        // Если файл с указателем pNameSortFile не закончился, то
        // следующую его серию копируем в файл b.f
        if( !FC.eof( ) )
            CopySeries( FC, FB );
    }
}

```

```

    } while( !FC.eof( ) );

    // Закрываем открытые файлы
    FC.clear( );
    FC.close_f( pNamesSortFile, 90 );
    FA.close_f( "a.f", 100 );
    FB.close_f( "b.f", 110 );

    return;
}

// *****
// Копирование одной серии из файлового объекта Finp в файловый
// объект FOut
template< class T >
void MergeSortFile< T > :: CopySeries(
    IOFILE &FInp,      // Файловый объект-источник
    IOFILE &FOut )     // Файловый объект-приемник
{
    bool    eos;        // true - конец серии в файловом объекте-
                        // источнике

    do
    {
        // Копируем один элемент из файлового объекта-источника
        // в файловый объект-приемник и одновременно вырабатываем
        // признак конца серии в файловом объекте-источнике
        Copy( Finp, FOut, eos );
    } while( !eos );

    return;
}

// *****
// Пересылка одного элемента из файлового объекта Finp в файловый
// объект FOut с определением, достигнут ли в файловом объекте Finp
// конец серии
template< class T >
void MergeSortFile< T > :: Copy(
    IOFILE &FInp,      // Файловый объект-источник
    IOFILE &FOut,      // Файловый объект-приемник
    bool    &EOS )     // true - достигнут конец серии
{
    EL      buf;        // Пересылаемый элемент файла
    EL      buf1;       // Элемент, следующий за buf

```

```

// Пересылка одного элемента
FInp >> buf.key;
if( !FInp )
{
    cout << " Error 120 readings from a file - source " << endl;
    exit( 120 );
}
FOut << buf.key << ' ';

// Проверка - достигнут ли конец серии?
if( FInp.eof( ) )
{
    // Достигнут конец файла, а значит, и конец серии
    EOS = true; return;
}

// Сравниваем пересланный элемент со следующим за ним - если
// следующий элемент имеет меньшее значение ключа, то также
// достигнут конец серии. Определяем текущую позицию указателя
// в файловом объекте-источнике относительно его начала
long l = FInp.tellg( );
FInp >> buf1.key;
if( !FInp.eof( ) )
{
    if( !FInp )
    {
        cout << " Error 130 readings from a file - source "
            << endl;
        exit( 130 );
    }
}
// Возврат файлового указателя в позицию l относительно его
// начала
FInp.seekg( l, ios::beg );
// Конец серии, если следующий элемент меньше предыдущего
EOS = ( buf.key > buf1.key );

return;
}

// *****
// Слияние серий из временных файлов a.f и b.f в исходный файл
// с указателем pNameSortFile
template< class T >
void MergeSortFile< T > :: Merge(
    int      &NumSer ) // Число серий в файле с указателем
                    // pNameSortFile
{

```

```

IOFILE FC,          // Файловый объект для записи
    FA,             // Файловый объект для чтения
    FB;            // Файловый объект для чтения
EL    bufb;         // Для проверки достижения конца файла b.f

// Открываем файл с указателем pNameSortFile для записи, а файлы
//   a.f и b.f - для чтения
FC.open_f( pNameSortFile, ios::out, 140 );
FA.open_f( "a.f", ios::in, 150 );
FB.open_f( "b.f", ios::in, 160 );

// Проверяем достижение конца файла b.f. Получаем текущее
//   значение файлового указателя в файле b.f
long    ll = FB.tellg( );
FB >> bufb.key;
if( !FB.eof( ) )
{
    // Конец файла еще не достигнут
    // Возвращаем файловый указатель на прежнее место
    FB.seekg( ll, ios::beg );
    // Слияние серий
    while( !FB.eof( ) )
    {
        MergeSeries( FA, FB, FC ); NumSer++;
    }
}
// В общем случае в файле a.f может быть на одну серию больше и
//   тогда эту серию следует скопировать в файл c.f
if( !FA.eof( ) )
{
    CopySeries( FA, FC ); NumSer++;
}

// Закрываем открытые файлы
FA.clear( ); FB.clear( );
FC.close_f( pNameSortFile, 170 );
FA.close_f( "a.f", 180 );
FB.close_f( "b.f", 190 );

return;
}

// *****
// Слияние двух серий из файловых объектов fa и fb в одну серию,
//   передаваемую в файловый объект fc
template< class T >
void MergeSortFile< T > :: MergeSeries(
    IOFILE &fa,          // Файловый объект-источник

```

```

IOFILE &fb,      // Файловый объект-источник
IOFILE &fc )     // Файловый объект-приемник
{
    EL    bufa,    // Элемент, прочитанный из файла a.f
          bufb;    // Элемент, прочитанный из файла b.f
    bool  EOS;     // true - конец серии
    long  l,       // Текущее значение файлового
          ll;      // указателя файла a.f и b.f

    do
    {
        // Получаем текущее значение файлового указателя файла a.f
        l = fa.tellg( );
        // Читаем очередной элемент из файла a.f
        fa >> bufa.key;
        if( !fa )
        {
            cout << " Error 200 readings from a file - source "
                    << endl;
            exit( 200 );
        }
        // Восстанавливаем текущее значение файлового указателя файла
        // a.f
        fa.seekg( l, ios::beg );

        // Получаем текущее значение файлового указателя файла b.f
        ll = fb.tellg( );
        // Читаем очередной элемент из файла a.f
        fb >> bufb.key;
        if( !fb )
        {
            cout << " Error 210 readings from a file - source "
                    << endl;
            exit( 210 );
        }
        // Восстанавливаем текущее значение файлового указателя файла
        // a.f
        fb.seekg( ll, ios::beg );

        if( bufa.key <= bufb.key )
        {
            // Копируем элемент с меньшим значением ключа из файла
            // a.f в файл с указателем pNameSortFile
            Copy( fa, fc, EOS );
            // Если при этом в файле a.f достигнут конец серии, то
            // остаток серии из файла b.f копируется в файл
            // с указателем pNameSortFile

```

```

        if( EOS )
            CopySeries( fb, fc );
    }
    else
    {
        // Копируем элемент с меньшим значением ключа из файла
        //   b.f в файл с указателем pNameSortFile
        Copy( fb, fc, EOS );
        // Если при этом в файле b.f достигнут конец серии, то
        //   остаток серии из файла a.f копируется в файл
        //   с указателем pNameSortFile
        if( EOS )
            CopySeries( fa, fc );
    }
} while( !EOS );

return;
}

#endif // Конец файла MergSort.h

```

Листинг 12.2. Файл MergSort.cpp

```

/*
    Сортировка файла на базе шаблона классов, использующего трехфайловую двухфазную
    сортировку естественным слиянием и определенного MergSort.h. Для открытия-закрытия
    файлов используется класс IOFILE, определение которого приведено во включаемом
    файле IOFile.h.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

// Стандартные и нестандартные заголовочные файлы, определение структуры
//   элемента файла и определение шаблона классов для трехфайловой
//   двухфазной сортировки файлов естественным слиянием
#include "MergSort.h"

// Тестирование
int main( void ) // Возвращает 0 при успехе
{
    // Определение sf классом MergeSortFile< int >: c.f - сортируемый
    //   файл
    MergeSortFile< int >
        sf( "c.f" );

    sf.PrintFile( "MergSort.out",
        "
            Состояние файла до сортировки:", ios::out );

```

```
// Сортировка
sf.NaturalMerge( );

sf.PrintFile( "MergSort.out",
"                Состояние файла после сортировки:" );

return 0;
}
```

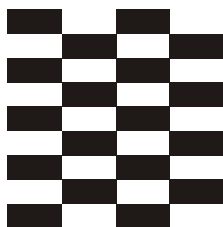
Дадим краткие пояснения к программе.

Трехленточная двухфазная сортировка файла естественным слиянием. Для выполнения сортировки файла служит общедоступный метод `NaturalMerge()`. В этом методе две фазы сортировки выражаются вызовом соответствующих методов — `Distribute` (фаза распределения) и `Merge` (фаза слияния). Эти методы вызываются последовательно и циклически, причем на фазе слияния подсчитывается количество полученных серий. Сортировка заканчивается при получении единственной серии. Отметим также, что методы `Distribute()` и `Merge()` являются служебными и поэтому имеют модификатор доступа `private`.

Фаза распределения. Фаза распределения серий из файлового объекта-источника в файловые объекты-приемники реализуется с помощью закрытого метода `Distribute()`, который, в свою очередь, использует закрытый метод `CopySeries()`, выполняющий копирование одной серии из сортируемого файла во вспомогательные файлы `a.f` и `b.f`. При таком способе распределения в файлах `a.f` и `b.f` оказывается либо одинаковое число серий, либо файл `a.f` содержит на одну серию больше, чем файл `b.f`. При копировании очередной серии метод `CopySeries()` циклически вызывает закрытый метод `Copy()`. Метод `Copy()` копирует один элемент из файлового объекта-источника в файловый объект-приемник и одновременно вырабатывает признак конца серии в файловом объекте-источнике. В этом методе для определения конца серии сопоставляются значения ключей двух соседних элементов файла: текущего (`buf.key`) и следующего за ним (`buf.l.key`). Требуемое "заглядывание вперед" достигается использованием методов `seekg()` и `tellg()` стандартного класса `fstream`. Таким образом, признаком конца серии является либо выполнение условия `buf.key > buf.l.key`, либо, в частном случае, достижение конца файла в объекте источнике.

Фаза слияния. Фаза слияния серий из файловых объектов-источников в файловый объект-приемник реализуется с помощью закрытого метода `Merge()`. Слияние серий выполняется циклически с помощью закрытого метода `MergeSeries()`. Поскольку в файле `a.f` может быть на одну серию больше, чем в файле `b.f`, то после слияния *пар серий* в файле `a.f` может оказаться лишняя серия, которую следует просто скопировать в файловый объект-приемник. При слиянии серий в методе `MergeSeries()` используются ранее использовавшиеся на фазе распределения серий закрытые методы `CopySeries()` и `Copy()`.

В заключение отметим, что наряду с сортировками файлов простым и естественным слиянием существуют более сложные и эффективные методы сортировки файлов. К их числу относится, например, сортировка файлов, использующая сбалансированное многопутевое слияние. Рассмотрение этих методов выходит за рамки данной книги.



Часть III

Стандартная библиотека языка C++ [9, 10]. Обобщенное программирование

Глава 13. Строки [9, 10]

Глава 14. Строковые потоки

Глава 15. Контейнерные классы

Глава 16. Итераторы и функциональные объекты

Глава 17. Алгоритмы

Процесс стандартизации языка C++ начался в 1989 и закончился в 1998 году публикацией стандарта языка. Результатом этой работы стало справочное руководство, насчитывающее около 750 страниц и опубликованное международной организацией по стандартизации (ISO). Стандарт получил название "Information Technology — Programming Languages — C++", ему был присвоен номер ISO/IEC 14882-1998 (см. файл CppStd.pgf на прилагаемом компакт-диске).

Стандарт стал важной вехой на пути развития языка C++. Строгие, формализованные определения синтаксиса и правил поведения языка C++ упрощают преподавание языка, написание программ, а также их адаптацию для других платформ. В результате пользователь получает свободу выбора между различными реализациями языка C++. От повышения надежности и переносимости программ выигрывают разработчики реализаций библиотеки и дополнительного инструментария. Благодаря наличию стандарта прикладные программисты, использующие язык C++, работают быстрее и эффективнее. При этом сокращаются затраты времени и сил на сопровождение программных продуктов.

Одной из важных составляющих стандарта является *стандартная библиотека* языка C++. Ее можно разделить на две части:

- функции, макросы, типы и константы, унаследованные из стандартной библиотеки языка C;
- стандартные классы и другие средства языка C++.

Функции, унаследованные из стандартной библиотеки языка C, в соответствии с их назначением, можно разделить на функции ввода-вывода, обработки строк, математические функции, функции для работы с динамической памятью, для поиска, сортировки и т. п. Описание средств, унаследованных из стандартной библиотеки языка C, приведено на компакт-диске в *приложениях 6 (константы, макросы и типы данных)* и *7 (функции)*. Вторая часть стандартной библиотеки языка C++ содержит классы, шаблоны и другие средства для ввода, вывода, хранения и обработки данных как стандартных, так и пользовательских типов.

В соответствии с их назначением *стандартные классы языка C++* можно разбить на следующие группы.

- *Потоковые классы* для управления потоками данных между оперативной памятью и внешними устройствами (дисками, клавиатурой, экраном монитора), а также в пределах оперативной памяти. Описание потоковых классов для ввода-вывода, кроме потоковых классов для работы со строками, приведено ранее в *гл. 5*.
- *Строковый класс* для удобной и защищенной от ошибок работы с символьными строками.
- *Контейнерные классы, алгоритмы и итераторы*. *Контейнерные классы* реализуют наиболее распространенные структуры для хранения данных: списки, вектора, множества и др. В состав стандартной библиотеки входят также *алгоритмы*, использующие и преобразующие контейнеры. *Итераторы* обеспечивают унифицированный доступ к элементам контейнерных классов.
- *Математические классы* для эффективной обработки массивов с плавающей точкой и комплексных данных.
- *Диагностические классы* для динамической идентификации типов и объектно-ориентированной обработки ошибок.

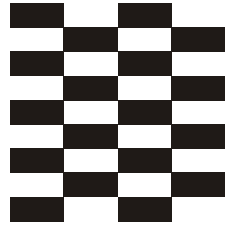
□ *Остальные классы* для динамического распределения памяти, адаптации к локальным особенностям и т. п.

Часть стандартной библиотеки, в которую входят контейнерные классы, алгоритмы и итераторы, называют *стандартной библиотекой шаблонов* (Standard Template Library, STL). STL занимает центральное место в стандартной библиотеке языка C++ и оказывает наибольшее влияние на ее общую архитектуру. Стандартная библиотека шаблонов содержит унифицированные средства для работы с *наборами объектов (коллекциями)* с применением современных и эффективных алгоритмов. Благодаря STL программисты могут пользоваться новыми разработками в области структур данных и алгоритмов, не разбираясь в принципах их работы. С точки зрения программиста, STL содержит набор классов коллекций для различных применений, а также поддерживает ряд эффективных алгоритмов для работы с этими коллекциями.

Все компоненты STL оформлены в виде шаблонов и могут использоваться с произвольными типами элементов. Но библиотека STL делает еще больше — она позволяет включать другие, новые классы коллекций и алгоритмов в уже существующие. Это поднимает язык C++ на новый уровень абстракции, позволяя отказаться от самостоятельного программирования динамических массивов, связанных списков, двоичных деревьев, разных алгоритмов поиска, сортировки и т. п. Для применения коллекции нужного типа программист просто определяет соответствующий контейнер, а затем вызывает для него нужные методы и алгоритмы для обработки данных. Однако за гибкость и мощь STL приходится расплачиваться тем, что библиотека получилась большой, сложной для изучения и освоения и нетривиальной.

Для использования средств стандартной библиотеки в программу с помощью директивы `#include` следует подключить соответствующие стандартные заголовочные файлы. Как уже указывалось ранее, элементы заголовочных файлов без расширения `h` определены в *стандартном пространстве имен* `std`, а одноименные файлы с расширением `h` — в *глобальном пространстве имен*. Список заголовочных файлов стандартной библиотеки приведен на компакт-диске в *приложении 5*.

Имена стандартных заголовочных файлов языка C++ для функций языка C, определенные в пространстве имен `std`, начинаются с буквы `c` и не имеют расширения, например, `<cstdio>`, `<cstring>`, `<cstdlib>`. Для каждого заголовочного файла вида `<cx>` (например, `<cstdio>`) существует файл `<x.h>` (например, `<stdio.h>`), определяющий те же имена в глобальном пространстве имен.



Глава 13

Строки [9, 10]

Язык C++ не предусматривает строкового типа данных. Вместо этого он поддерживает символьные массивы, завершаемые нуль-символом. Для работы с такими массивами библиотека языка содержит строковые функции, унаследованные от языка C и описанные в заголовочном файле `<string.h>` (`<cstring>`). Эти функции кратко описаны в *приложении 7* на компакт-диске. Строковые функции языка C обладают следующими отличительными особенностями:

- высокое быстродействие;
- неудобный интерфейс;
- опасность их использования, поскольку выход за границы строки в функциях не контролируется.

Последних двух недостатков лишены классы `string` и `wstring` стандартной библиотеки языка C++, которые являются специализациями шаблонного класса `basic_string` для типов `char` и `wchar_t`:

```
typedef basic_string<char> string;  
typedef basic_string<wchar_t> wstring;
```

Класс `wstring` позволяет работать со строками, содержащими символы в многобайтовой кодировке (например, в кодировке Unicode или азиатских кодировках), что для наших условий не актуально. По этой причине далее рассматривается только класс `string`.

Но, в соответствии с житейской мудростью — "за удовольствия надо платить", использование для работы со строками класса `string` приводит к некоторому понижению быстродействия. Тем не менее, использование этого класса для работы со строками весьма целесообразно и является хорошим стилем программирования.

При работе со строками могут возникать и часто возникают недоразумения. Это происходит из-за того, что термин "строка" может означать совершенно разные вещи — обычный символьный массив типа `char *` (с модификатором `const` или без него) или экземпляр (объект) класса `string`. Термин "строка" также может быть обобщающим названием для объектов, которые содержат строковую информацию. Для разрешения этой коллизии в дальнейшем под термином "строка" будем понимать объект любого из строковых типов (`string` или `wstring`). "Традиционные" же строки типов `char *` и `const char *` будем называть *C-строками*. Попутно заметим, что в стандарте языка C++ тип строковых литералов (например, "Привет!") был заменен

на `const char *`. Вместе с тем для обеспечения совместимости с языком C поддерживается неявное (хотя и не желательное) преобразование к типу `char *`.

Строковые классы стандартной библиотеки языка C++ позволяют работать со строками как с обычными предопределенными типами, не создающими проблем для пользователей. Это означает, что строки, как и объекты предопределенных типов, можно копировать, присваивать, складывать, сравнивать и т. п., используя традиционные операции. Современная обработка данных во многом ориентирована на работу с текстом. Поэтому указанное ранее особенно важно для программистов с опытом работы на языках C, FORTRAN и других языках, в которых обработка строк реализована весьма неудобно.

Основные действия со строками выполняются в классе `string` с помощью операций и методов, а длина строки изменяется динамически в соответствии с реальными потребностями. Это делает работу со строками эффективной по занятой памяти. Для использования класса `string` в программу нужно включить заголовочный файл `<string>`. Проиллюстрируем сказанное примером (листинги 13.1, 13.2).

Листинг 13.1. Файл `str1.cpp`

```
/*
Сравнение C-строк и возможностей стандартного класса string.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <cstring>           // C-строки в пространстве имен std
#include <string>            // Стандартный класс string
#include <iostream>         // Поточковый ввод-вывод C++

using namespace            // Используем стандартное
    std;                  // пространство имен

// Работа с C-строками и строками C++
int main( void )          // Возвращает 0 при успехе
{
    // Символьные массивы для C-строк
    char    c1[ 80 ], c2[ 80 ], c3[ 80 ];
    // Пустые строки
    string  s1, s2, s3;

    // Присваивание
    // Язык C
    strcpy( c1, "C:string1 " ); strcpy( c2, c1 );
    // C++
    s1 = "C++:string1 "; s2 = s1;

    // Сложение
    // Язык C
```

```

strcpy( c3, c1 ); strcat( c3, c2 );
// C++
s3 = s1 + s2;

// Сравнение строк и их печать
if( strcmp( c2, c3 ) < 0 )
    // C:string1 C:string1 (вид первой строки печати)
    cout << c3 << endl;
else
    cout << c2 << endl;
if( s2 < s3 )
    // C++:string1 C++:string1 (вид второй строки печати)
    cout << s3 << endl;
else
    cout << s2 << endl;
cout << endl;

return 0;
}

```

Листинг 13.2. Результаты выполнения программы

```

C:string1 C:string1
C++:string1 C++:string1

Press any key to continue

```

Комментарии излишни. Конечно же, работать со строками, используя стандартный класс `string` проще, удобнее и, что важнее всего, надежнее.

13.1. Создание строк. Конструкторы и деструктор строк

В классе `string` определены несколько конструкторов, которые позволяют создавать объекты-строки различными способами и весьма удобно (листинги 13.3, 13.4).

Листинг 13.3. Файл `str2.cpp`

```

/*
    Создание строковых объектов. Конструкторы класса string.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <string>           // Стандартный класс string
#include <iostream>         // Поточковый ввод-вывод

```

```
using namespace          // Используем стандартное
                        std;      // пространство имен

int main( void )         // Возвращает 0 при успехе
{
    // Пустая строка и ее печать. Используется конструктор умолчания
    // вида: string( void );
    string    s1;
    cout << "s1:" << s1 << endl;

    // Используется конструктор вида: string( const char *str );
    // Строка, инициализированная C-строкой, и ее печать
    string    s2( "Татьяна" );
    cout << "s2:" << s2 << endl;

    // Используется конструктор вида: string( const char *str,
    // size_type len ); Строка, инициализированная C-строкой str
    // с ограничением длины строки len, и ее печать
    string    s3( "Татьяна", 3 );
    cout << "s3:" << s3 << endl;

    // Используется конструктор копирования вида:
    // string( const string &str );
    // Строка, инициализированная строкой-аргументом, и ее печать
    string    s4( s2 );
    cout << "s4:" << s4 << endl;

    // Используется конструктор вида:
    // string( const string &str, size_type stridx, size_type len );
    // Строка, инициализированная не более, чем len символами строки
    // str, начиная с индекса stridx, и ее печать.
    string    s5( s2, 4, 2 );
    cout << "s5:" << s5 << endl;

    // Используется конструктор вида: string( size_type num, char c );
    // Строка, инициализированная num символами "с", и ее печать
    string    s6( 4, 't' );
    cout << "s6:" << s6 << endl;

    // Используется конструктор вида:
    // string( iterator beg, iterator end );
    // Строка, инициализированная всеми символами интервала,
    // заданного итераторами beg и end, и ее печать
    string    s7( s2.begin() + 4, s2.end() - 2 );
    cout << "s7:" << s7 << endl;
```

```

    cout << endl;

    return 0;
}

```

Листинг 13.4. Результаты выполнения программы

```

s1:
s2:Татьяна
s3:Тат
s4:Татьяна
s5:ян
s6:tttt
s7:я

```

Press any key to continue

В классе `string` имеется *деструктор*, который вызывается в программе автоматически, уничтожая все символы строки и освобождая динамическую память.

13.2. Операции над строками

Для объектов класса `string` определены следующие операции: "=" (присваивание), "+" (конкатенация), "==" (равенство), "!=" (неравенство), "<" (меньше), "<=" (меньше или равно), ">" (больше), ">=" (больше или равно), "[]" (индексация), "<<" (вывод), ">>" (ввод) и "+=" (добавление).

Синтаксис этих операций и их действие очевидны. Использование многих из них иллюстрируют два последних примера. При выполнении операций строчный размер автоматически устанавливается так, чтобы объект мог содержать присваиваемое ему значение. Отличительной особенностью строк является то, что для них не соблюдается соответствие (эквивалентность) между адресом первого элемента строки и именем объекта, как это было в случае С-строк (значение `&s[0]` не равно `s`).

Рассмотрим еще один иллюстрирующий пример (листинги 13.5, 13.6).

Листинг 13.5. Файл `str3.cpp`

```

/*
    Операции над строковыми объектами класса string.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <fstream>           // файловый ввод-вывод
#include <string>             // Стандартный класс string
#include <iostream>          // Поточковый ввод-вывод

```

```
using namespace      // Используем стандартное
                    std;      // пространство имен

int main( void )      // Возвращает 0 при успехе
{
    string    s1( "Ольга " ), s2, s3, s4, s5, s6;

    // Операция присваивания. Прототип используемой функции:
    //  string & operator=( const string & );
    s2 = s1;
    cout << "s2:" << s2 << endl;

    // Операция присваивания. Прототип используемой функции:
    //  string & operator=( const char * );
    s3 = "Лена";
    cout << "s3:" << s3 << endl;
    s4 = "Т";
    cout << "s4:" << s4 << endl;

    // Операция присваивания. Прототип используемой функции:
    //  string & operator=( char );
    s5 = 'Т';
    cout << "s5:" << s5 << endl;

    // Индексация
    cout << "s3[ 1 ]:" << s3[ 1 ] << endl;
    cout << "s3.at( 1 ):" << s3.at( 1 ) << endl;
    cout << "&s3[ 1 ]:" << &s3[ 1 ] << endl;
    cout << "&s3.at( 1 ):" << &s3.at( 1 ) << endl;

    // Ввод из файла (файл содержит строку "Татьяна")
    ifstream    inp( "inp.dat" );
    inp >> s6;
    if( !inp )
    {
        cout << "Ошибка ввода" << endl;
        exit( 1 );
    }
    cout << "s6:" << s6 << endl;

    // Добавление
    s1 += s6;
    cout << "s1:" << s1 << endl;

    cout << endl;

    return 0;
}
```

Листинг 13.6. Результаты выполнения программы

```
s2:Ольга
s3:Лена
s4:T
s5:T
s3[ 1 ]:e
s3.at( 1 ):e
&s3[ 1 ]:ена
&s3.at( 1 ):ена
s6:Татьяна
s1:Ольга Татьяна
```

```
Press any key to continue
```

Анализ примера показывает, что кроме ненадежной операции индексации для доступа к элементу строки определен снабженный контролем нарушения индексации метод `at()`. Если индекс приводит к выходу за пределы строки, то порождается исключение `out_of_range`.

Операции сравнения определены только над полными строками и C-строками. Сравнение символов строк производится в лексикографическом порядке в соответствии с их текущими трактовками. Операции сравнения возвращают значение с типом `bool` (`true` или `false`).

Для работы со строками целиком перечисленных ранее операций достаточно. Для обработки частей строк в классе `string` имеется множество методов. Рассмотрим наиболее употребительные из них. Для удобства рассмотрения разобьем методы на следующие подгруппы:

- ☐ методы присваивания и добавления частей строк;
- ☐ методы преобразования строк;
- ☐ методы поиска подстрок;
- ☐ методы сравнения и получения характеристик строк.

13.2.1. Присваивание и добавление частей строк

В классе `string` для присваивания части одной строки другой строке можно использовать метод `assign()`, а для добавления части одной строки к другой — метод `append()`. Далее приводится пример, иллюстрирующий возможности этих методов (листинги 13.7, 13.8).

Листинг 13.7. Файл `str4.cpp`

```
/*
Операции над строковыми объектами класса string. Присваивание и добавление частей строк с помощью методов класса string.
```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/

#include <string>           // Стандартный класс string
#include <iostream>         // Поточковый ввод-вывод

using namespace           // Используем стандартное
    std;                  // пространство имен

int main( void )          // Возвращает 0 при успехе
{
    string    s1( "Ольга+Михаил" ), s2( "Юрий" );

    // Присваиваем строке часть строки str. Прототип используемого
    // метода: string & assign(const string &str, size_type pos,
    // size_type n );
    s2.assign( s1, 0, 5);
    // Строка s2 получит значение подстроки s1, начиная с позиции 0
    // длиной 5 символов. Если второй аргумент в вызове выводит за
    // пределы строки s1, то порождается исключение out_of_range. Если
    // третий аргумент в вызове выводит за пределы строки s1, то
    // строке s2 присваивается значение строки s1. В нашем примере
    // строка s2 получит значение "Ольга"
    cout << "s2:" << s2 << endl;

    // Присваиваем строке часть C-строки str длиной n. Прототип
    // используемого метода:
    // string & assign( const char *str, size_type n );
    s2.assign( "Ольга+Михаил", 5);
    // Строка s2 будет содержать первые 5 символов строки "Ольга+Михаил".
    // В нашем примере строка s2 получит также значение "Ольга"
    cout << "s2:" << s2 << endl;

    // Добавляем к строке часть строки str, содержащую не более n
    // символов, начиная с позиции pos. Прототип используемого метода:
    // string & append(const string &str, size_type pos, size_type n );
    s2 = "Юрий+";
    s2.append( s1, 0, 5);
    // К строке s2 добавится подстрока s1, начиная с позиции 0 длиной 5
    // символов. Если второй аргумент в вызове выводит за пределы
    // строки s1, то порождается исключение out_of_range. Если третий
    // аргумент в вызове выводит за пределы строки s1, то к строке s2
    // добавится строка s1. Если длина результата получится больше
    // максимально допустимой длины строки, порождается исключение
    // length_error. В нашем примере строка s2 получит значение
    // "Юрий+Ольга"
    cout << "s2:" << s2 << endl;
}

```

```
// Добавляем к строке часть C-строки str длиной n. Прототип
//   используемого метода:
//   string & append( const char *str, size_type n );
s2 = "Юрий+";
s2.append( "Ольга+Михаил", 5);
// К строке s2 добавятся первые 5 символов строки "Ольга+Михаил".
// В нашем примере строка s2 получит значение "Юрий+Ольга"
cout << "s2:" << s2 << endl << endl;

return 0;
}
```

Листинг 13.8. Результаты выполнения программы

```
s2:Ольга
s2:Ольга
s2:Юрий+Ольга
s2:Юрий+Ольга
```

Press any key to continue

13.2.2. Преобразования строк

В классе `string` для преобразования строк можно использовать следующие методы:

- ☐ `insert()` — вставка в одну строку части другой строки;
- ☐ `erase()` — удаление части строки;
- ☐ `clear()` — очистка строки;
- ☐ `replace()` — замена части строки;
- ☐ `swap()` — обмен содержимого двух строк;
- ☐ `substr()` — выделение части строки;
- ☐ `c_str()` — преобразование объекта типа `string` в C-строку;
- ☐ `data()` — преобразование объекта типа `string` в массив символов (в отличие от предыдущего метода в конец массива символ `'\0'` не помещается);
- ☐ `copy()` — копирование в C-строку части объекта с типом `string`.

Строки и C-строки. Между объектами класса `string` и C-строками существует тесная связь. C-строки могут использоваться практически в любых операциях вместе со строками, и рассмотренные ранее примеры это подтверждают. В частности, поддерживается автоматическое преобразование типа `const char *` в строку. С другой стороны, *не поддерживается* автоматическое преобразование строковых объектов в C-строки. Возможность такого преобразования была исключена по соображениям безопасности. Вместо этого в классе `string` были определены специальные методы для создания и записи/копирования C-строк. Следует помнить, что в строках *не существует* специальной интерпретации символа `'\0'`, который в C-строках является признаком конца строки. Символ `'\0'` может входить в строки наравне с любым другим символом.

Преобразование содержимого строки в массив символов или в С-строку осуществляется тремя методами класса `string`.

- ❑ `c_str()`. Возвращает содержимое строки в формате С-строки (то есть с завершающим символом `'\0'`).
- ❑ `data()`. Возвращает содержимое строки в виде массива символов. Возвращаемое значение не является С-строкой, поскольку к нему не присоединяется символ `'\0'`.
- ❑ `copy()`. Копирует содержимое строки в символьный массив, передаваемый при вызове метода. При этом завершающий символ `'\0'` также не присоединяется.

Обычно в программе следует работать со строками, а их преобразование в С-строки или символьные массивы должно производиться непосредственно перед тем, как вам потребуется содержимое строки в виде типа `const char *`. При этом имейте в виду, что возвращаемое значение методов `c_str()` и `data()` остается действительным *только* до следующего вызова не константного метода для той же строки.

А теперь приведем пример, иллюстрирующий использование перечисленных методов (листинги 13.9, 13.10).

Листинг 13.9. Файл `str5.cpp`

```
/*
    Операции над строковыми объектами класса string. Преобразования строк с помощью
    методов класса string.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <string>           // Стандартный класс string
#include <iostream>         // Поточковый ввод-вывод

using namespace           // Используем стандартное
    std;                  // пространство имен

int main( void )          // Возвращает 0 при успехе
{
    // Метод insert( ) - вставка в одну строку части другой строки
    //*****

    string    s1( "Ольхаил" ), s2( "га+Ми" );

    s1.insert( 3, s2 );

    // Вставляем в строку s1, начиная с позиции 3, строку s2. Прототип
    // используемого метода: string & insert( size_type pos1,
    // const string &str );
    // Если первый аргумент в вызове выводит за пределы строки s1, то
    // порождается исключение out_of_range. Если длина результата
    // получится больше максимально допустимой длины строки,
    // порождается исключение length_error. В нашем примере строка s1
```

```

// получит значение "Ольга+Михаил"
cout << "s1:" << s1 << endl;

string s3( "Ах! Ольга+Михаил" );
s1 = "Ольхаил";

s1.insert( 3, s3, 7, 5 );
// Вставляем в строку s1, начиная с позиции 3, 5 символов из строки
// s3, начиная с позиции 7. Прототип используемого метода:
// string & insert( size_type pos, const string &str,
//               size_type pos1, size_type n );
// Если первый или третий аргумент в вызове выводит за пределы
// соответствующей строки, то порождается исключение out_of_range.
// Если четвертый аргумент выводит за пределы строки s3, то
// вставляется вся строка s3. Если длина результата получится
// больше максимально допустимой длины строки, порождается
// исключение length_error. В нашем примере строка s1 получит
// значение "Ольга+Михаил"
cout << "s1:" << s1 << endl;

s1 = "Ольхаил";

s1.insert( 3, "га+Михаил", 5 );
// Вставляем в строку s1, начиная с позиции 3, 5 символов из C-строки
// "га+Михаил". Прототип используемого метода:
// string & insert( size_type pos, const char *s, size_type n );
// Если первый аргумент в вызове выводит за пределы соответствующей
// строки, то порождается исключение out_of_range. Если третий
// аргумент в вызове выводит за пределы строки старого стиля, то
// вставляется вся эта строка s3, а "хвост" строки s1 теряется.
// Если длина результата получится больше максимально допустимой
// длины строки, порождается исключение length_error. В нашем
// примере строка s1 получит значение "Ольга+Михаил"
cout << "s1:" << s1 << endl;

// Метод erase( ) - удаление части строки
//*****

s3.erase( 9, 7 );
// Удаляем из строки s3 7 СИМВОЛОВ, начиная с позиции 9. Прототип
// используемого метода:
// string & erase( size_type pos = 0, size_type n = npos );
// Здесь npos - статический член класса string с самым большим
// значением для типа size_type. Если n не указано, то удаляется
// весь остаток строки. Если не указано ни pos, ни n, то очищается
// вся строка. В нашем примере строка s3 получит значение

```

```
// "Ах! Ольга"
cout << "s3:" << s3 << endl;

// Метод replace( ) - замена части строки
//*****

s1 = "Михаил";
s3.replace( 4, 5, s1 );
// Заменяем часть строки s3, начинающуюся с позиции 4 длиной 5
// символов, содержимым строки s1. Прототип используемого метода:
// string & replace( size_type pos, size_type n, const string &s );
// Если первый аргумент в вызове (соответствует параметру pos)
// выводит за пределы строки s3, то порождается исключение
// out_of_range. Если длина результата больше максимально
// допустимой длины строки, то порождается исключение length_error.
// В нашем примере строка s3 получит значение "Ах! Михаил"
cout << "s3:" << s3 << endl;

s3 = "Ах! xxxxxxл";
s1 = "Михаил";
s3.replace( 4, 6, s1, 0, 5 );
// Заменяем часть строки s3, начинающуюся с позиции 4 длиной 6
// символов, содержимым части строки s1, начиная с позиции 0 длиной
// 5 символов. Прототип используемого метода:
// string & replace( size_type pos1, size_type n1, const string &s,
//                  size_type pos2, size_type n2 );
// Если первый аргумент в вызове (соответствует параметру pos1)
// выводит за пределы строки s3 или четвертый аргумент в вызове
// (соответствует параметру pos2) выводит за пределы строки s1, то
// порождается исключение out_of_range. Если длина результата
// больше максимально допустимой длины строки, то порождается
// исключение length_error. В нашем примере строка s3 получит
// значение "Ах! Михаил"
cout << "s3:" << s3 << endl;

s3 = "Ах! xxxxxxл";
s3.replace( 4, 6, "Михаил", 5 );
// Заменяем часть строки s3, начинающуюся с позиции 4 длиной 6
// символов, первыми 5 символами С-строки, указанной третьим
// аргументом. Прототип используемого метода:
// string & replace( size_type pos1, size_type n1, const char *s,
//                  size_type n2 );
// Если первый аргумент в вызове (соответствует параметру pos1)
// выводит за пределы строки s3, то порождается исключение
// out_of_range. Если длина результата больше максимально
// допустимой длины строки, то порождается исключение length_error.
```

```
// В нашем примере строка s3 получит значение "Ах! Михаил"
cout << "s3:" << s3 << endl;

// Метод swap( ) - обмен содержимого двух строк
//*****

s3 = "Строка 3";
s1 = "Строка 1";
s3.swap( s1 );
// Обмен содержимого строк s3 и s1. Прототип используемого метода:
// swap( string &s );
// В нашем примере строка s3 получит значение "Строка 1", а строка
// s1 - значение "Строка 3"
cout << "s3:" << s3 << endl;

// Метод substr( ) - выделение части строки
//*****

s3 = "Строка 3";
s1 = "Ах! Ольга+Михаил";
s3 = s1.substr( 4, 5 );
// Возвращает подстроку строки s1 длиной 5 символов начинающуюся
// с позиции 4. Прототип используемого метода:
// string substr( size_type pos = 0, size_type n = npos ) const;
// Здесь npos - статический член класса string с самым большим
// значением для типа size_type. Если первый аргумент в вызове
// (соответствует параметру pos) выводит за пределы строки s1, то
// порождается исключение out_of_range. Если второй аргумент в
// вызове (соответствует параметру n) выводит за пределы строки s1,
// то возвращается вся оставшаяся часть строки s1. В нашем примере
// строка s3 получит значение "Ольга"
cout << "s3:" << s3 << endl;

// Метод c_str( ) - преобразование объекта типа string в C-строку
//*****

s1 = "Ах! Ольга+Михаил";
cout << "s1.c_str( ):" << s1.c_str( ) << endl;
// Преобразует строку s1 в C-строку. Возвращает константный указатель
// на оканчивающуюся нуль-символом C-строку. Эту строку нельзя
// изменить. Указатель, который на нее ссылается, может стать
// некорректным после любой неконстантной операции над строкой-
// источником. Прототип используемого метода:
// const char * c_str( void ) const;
// В нашем примере возвращается указатель на строку
// "Ах! Ольга+Михаил"
```

```

// Метод copy( ) - копирование в C-строку части объекта с типом
// string
//*****

char    arr[ 20 ];
s1 = "Ах! Ольга+Михаил";
unsigned Retcode = s1.copy( arr, 8, 10 );
// Копирует в символьный массив arr 8 символов из строки s1, начиная
// с позиции 10. Возвращает количество скопированных символов.
// Ноль-символ в массив arr не заносится. Прототип используемого
// метода:
// size_type copy( char *s, size_type n, size_type pos = 0 ) const;
// Если третий аргумент в вызове (соответствует параметру pos)
// выводит за пределы строки-источника, то порождается исключение
// out_of_range. Если второй аргумент в вызове (соответствует
// параметру n) выводит за пределы строки-источника, то копируется
// оставшаяся часть строки-источника. В нашем примере возвращается
// 6, а в массиве arr первые 6 элементов содержат соответственно
// символы <Михаил>, а содежимое остальных элементов этого массива
// не определено
cout << "Retcode: " << Retcode << endl;

// Преобразует строку s1 в символьный массив. Возвращает константный
// указатель на символьный массив. Этот массив нельзя изменить.
// Указатель, который на него ссылается, может стать некорректным
// после любой неконстантной операции над строкой-источником.
// Прототип используемого метода: const char * data( void ) const;
// В нашем примере печатается содержимое некоторых ячеек полученного
// символьного массива
s1 = "Ах! Ольга+Михаил";
const char *ptr = s1.data( );
cout << "ptr[ 0 ]: " << ptr[ 0 ] << endl
    << "ptr[ 10 ]: " << ptr[ 10 ] << endl << endl;

return 0;
}

```

Листинг 13.10. Результаты выполнения программы

```

s1:Ольга+Михаил
s1:Ольга+Михаил
s1:Ольга+Михаил
s3:Ах! Ольга
s3:Ах! Михаил
s3:Ах! Михаил
s3:Ах! Михаил

```

```
s3:Строка 1
s3:Ольга
s1.c_str( ):Ах! Ольга+Михаил
Retcode: 6
ptr[ 0 ]: А
ptr[ 10 ]: М
```

Press any key to continue

13.2.3. Поиск подстрок

В классе `string` для поиска подстрок предусмотрено большое разнообразие методов. Далее перечислены основные из этих методов.

Поиск самого левого вхождения заданной строки (подстроки) или символа в другую строку.

```
size_type find( const string &str, size_type pos = 0 ) const;
```

Ищет самое левое вхождение строки `str` в вызывающую строку, начиная с позиции `pos` вызывающей строки. Возвращает позицию найденной строки или `npos` (наибольшую возможную длину строкового объекта — наибольшее значение для типа `size_type`), если строка не найдена.

```
size_type find( char c, size_type pos = 0 ) const;
```

Ищет самое левое вхождение символа `c` в вызывающую строку, начиная с позиции `pos` вызывающей строки. Возвращает позицию найденного символа или `npos`, если символ не найден.

```
size_type find( const char *s, size_type pos = 0 ) const;
```

Ищет самое левое вхождение C-строки `s` в вызывающую строку, начиная с позиции `pos` вызывающей строки. Возвращает позицию найденной строки или `npos`, если строка не найдена.

```
size_type find( const char *s, size_type pos, size_type len ) const;
```

Ищет самое левое вхождение подстроки длиной не более `len` символов C-строки `s` в вызывающую строку, начиная с позиции `pos` вызывающей строки. Возвращает позицию найденной подстроки или `npos`, если подстрока не найдена.

Примеры использования этих методов приведены далее в программном проекте `str6.cpp` (листинг 13.11).

Поиск самого правого вхождения заданной строки (подстроки) или символа в другую строку.

```
size_type rfind( const string &str, size_type pos = npos ) const;
```

Ищет самое правое вхождение строки `str` в вызывающую строку до позиции `pos` вызывающей строки. Возвращает позицию найденной строки или `npos`, если строка не найдена.

```
size_type rfind( char c, size_type pos = npos ) const;
```

Ищет самое правое вхождение символа `c` в вызывающую строку до позиции `pos` вызывающей строки. Возвращает позицию найденного символа или `npos`, если символ не найден.

```
size_type rfind( const char *s, size_type pos = npos ) const;
```

Ищет самое правое вхождение С-строки *s* в вызывающую строку до позиции *pos* вызывающей строки. Возвращает позицию найденной строки или *npos*, если строка не найдена.

```
size_type rfind( const char *s, size_type pos, size_type len ) const;
```

Ищет самое правое вхождение подстроки длиной не более *len* символов С-строки *s* в вызывающую строку до позиции *pos* вызывающей строки. Возвращает позицию найденной подстроки или *npos*, если подстрока не найдена.

Примеры использования этих методов приведены далее в программном проекте *str6.cpp* (листинг 13.11).

Поиск самого левого вхождения любого символа заданной строки в другую строку.

```
size_type find_first_of( const string &str, size_type pos = 0 ) const;
```

Ищет самое левое вхождение любого символа строки *str* в вызывающую строку, начиная с позиции *pos* вызывающей строки. Возвращает позицию найденного символа или *npos*, если вхождение символа не найдено.

```
size_type find_first_of( ( const char *s, size_type pos = 0 ) const;
```

Ищет самое левое вхождение любого символа С-строки *s* в вызывающую строку, начиная с позиции *pos* вызывающей строки. Возвращает позицию найденного символа или *npos*, если вхождение символа не найдено.

```
size_type find_first_of( ( const char *s, size_type pos, size_type len )  
    const;
```

Ищет самое левое вхождение любого символа подстроки длиной не более *len* символов С-строки *s* в вызывающую строку, начиная с позиции *pos* вызывающей строки. Возвращает позицию найденного символа или *npos*, если вхождение символа не найдено.

Примеры использования этих методов приведены далее в программном проекте *str6.cpp* (листинг 13.11).

Поиск самого правого вхождения любого символа заданной строки в другую строку.

```
size_type find_last_of( const string &str, size_type pos = npos ) const;
```

Ищет самое правое вхождение любого символа строки *str* в вызывающую строку, до позиции *pos* вызывающей строки. Возвращает позицию найденного символа или *npos*, если вхождение символа не найдено.

```
size_type find_last_of( ( const char *s, size_type pos = npos ) const;
```

Ищет самое правое вхождение любого символа С-строки *s* в вызывающую строку, до позиции *pos* вызывающей строки. Возвращает позицию найденного символа или *npos*, если вхождение символа не найдено.

```
size_type find_last_of( const char *s, size_type pos, size_type len )  
    const;
```

Ищет самое правое вхождение любого символа подстроки длиной не более *len* символов С-строки *s* в вызывающую строку, до позиции *pos* вызывающей строки. Возвращает позицию найденного символа или *npos*, если вхождение символа не найдено.

Примеры использования этих методов приведены далее в программном проекте *str6.cpp* (листинг 13.11).

Поиск самого левого символа заданной строки, который не входит в другую строку.

```
size_type find_first_not_of( const string &str, size_type pos = 0 )
    const;
```

Ищет самый левый символ вызывающей строки, начиная с позиции `pos` вызывающей строки, который не входит в строку `str`. Возвращает позицию найденного символа или `npos` в противном случае.

```
size_type find_first_not_of( const char *s, size_type pos = 0 ) const;
```

Ищет самый левый символ вызывающей строки, начиная с позиции `pos` вызывающей строки, который не входит в С-строку `s`. Возвращает позицию найденного символа или `npos` в противном случае.

```
size_type find_first_not_of( const char *s, size_type pos,
                             size_type len ) const;
```

Ищет самый левый символ вызывающей строки, начиная с позиции `pos` вызывающей строки, который не входит в подстроку длиной `len` символов С-строки `s`. Возвращает позицию найденного символа или `npos` в противном случае.

Примеры использования этих методов приведены далее в программном проекте `str6.cpp` (листинг 13.11).

Поиск самого правого символа заданной строки, который не входит в другую строку.

```
size_type find_last_not_of( const string &str, size_type pos = npos )
    const;
```

Ищет самый правый символ вызывающей строки, до позиции `pos` вызывающей строки, который не входит в строку `str`. Возвращает позицию найденного символа или `npos` в противном случае.

```
size_type find_last_not_of( const char *s, size_type pos = npos ) const;
```

Ищет самый правый символ вызывающей строки, до позиции `pos` вызывающей строки, который не входит в С-строку `s`. Возвращает позицию найденного символа или `npos` в противном случае.

```
size_type find_last_not_of( const char *s, size_type pos, size_type len )
    const;
```

Ищет самый правый символ вызывающей строки, до позиции `pos` вызывающей строки, который не входит в подстроку длиной `len` символов С-строки `s`. Возвращает позицию найденного символа или `npos` в противном случае.

Примеры использования этих методов приведены в листингах 13.11, 13.12.

Листинг 13.11. Файл `str6.cpp`

```
/*
    Операции над строковыми объектами класса string. Поиск подстрок с помощью методов класса string.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <string>           // Стандартный класс string
```

```

#include <iostream>          // Поточковый ввод-вывод

using namespace             // Используем стандартное
    std;                   // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    string    s1( "C++ - мощный, мощный язык" ),
              s2( "мо" );

    // Методы find( ) и rfind( ) - поиск самого левого или правого
    // вхождения заданной строки (подстроки) или символа в другую
    // строку
    //*****

    cout << "Тестирование методов find( ) и rfind( )" << endl
         << "*****" << endl << endl;

    string::size_type
        left = s1.find( s2 );
    cout << "Самое левое вхождение строки s2=<" << s2 << "> в строку"
         << "\ns1=<" << s1 << "> имеет позицию " << left << endl;
    cout << "В этой позиции находится символ: <" << s1[ left ] <<
         ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
         << ">" << endl;

    string::size_type
        right = s1.rfind( s2 );
    cout << "\nСамое правое вхождение строки s2=<" << s2 << "> в строку"
         << "\ns1=<" << s1 << "> имеет позицию " << right << endl;
    cout << "В этой позиции находится символ: <" << s1[ right ] <<
         ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]
         << ">" << endl;

    left = s1.find( 'н' );
    cout << "\nСамое левое вхождение символа <" << 'н' << "> в строку"
         << "\ns1=<" << s1 << "> имеет позицию " << left << endl;
    cout << "В этой позиции находится символ: <" << s1[ left ] <<
         ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
         << ">" << endl;

    right = s1.rfind( 'н' );
    cout << "\nСамое правое вхождение символа <" << 'н' << "> в строку"
         << "\ns1=<" << s1 << "> имеет позицию " << right << endl;
    cout << "В этой позиции находится символ: <" << s1[ right ] <<
         ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]

```

```

    << ">" << endl;

    left = s1.find( "мо" );
    cout << "\nСамое левое вхождение C-строки <" << "мо" << "> в строку"
        << "\ns1=<" << s1 << "> имеет позицию " << left << endl;
    cout << "В этой позиции находится символ: <" << s1[ left ] <<
        ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
        << ">" << endl;

    right = s1.rfind( "мо" );
    cout << "\nСамое правое вхождение C-строки <" << "мо" <<
        "> в строку" << "\ns1=<" << s1 << "> имеет позицию " << right
        << endl;
    cout << "В этой позиции находится символ: <" << s1[ right ] <<
        ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]
        << ">" << endl;

    left = s1.find( "мощный", 0, 2 );
    cout << "\nСамое левое вхождение подстроки длиной 2 символа"
        " C-строки\n<" << "мощный" << "> в строку " << "s1=<" << s1
        << ">,\nначиная с позиции 0, имеет позицию " << left << endl;
    cout << "В этой позиции находится символ: <" << s1[ left ] <<
        ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
        << ">" << endl;

    right = s1.rfind( "мощный", 20, 2 );
    cout << "\nСамое правое вхождение подстроки длиной 2 "
        "символа C-строки\n<" << "мощный" << "> в строку " << "s1=<"
        << s1 << ">,\nначиная с позиции 20, имеет позицию " << right
        << endl;
    cout << "В этой позиции находится символ: <" << s1[ right ] <<
        ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]
        << ">" << endl;

    // Методы find_first_of( ) и find_last_of( ) - поиск самого левого
    // или правого вхождения любого символа заданной строки (подстроки)
    // в другую строку
    //*****

    cout << "\nТестирование методов find_first_of( ) и find_last_of( )"
        << "\n*****"
        << endl;

    s2 = "май";
    left = s1.find_first_of( s2 );
    cout << "\nСамое левое вхождение любого символа строки s2=<" << s2

```

```

    << ">\nв строку " << "s1=<" << s1 << "> имеет позицию " << left
    << endl;
cout << "В этой позиции находится символ: <" << s1[ left ] <<
    ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
    << ">" << endl;

right = s1.find_last_of( s2 );
cout << "\nСамое правое вхождение любого символа строки s2=<" << s2
    << ">\nв строку " << "s1=<" << s1 << "> имеет позицию " << right
    << endl;
cout << "В этой позиции находится символ: <" << s1[ right ] <<
    ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]
    << ">" << endl;

left = s1.find_first_of( "май" );
cout << "\nСамое левое вхождение любого символа C-строки \n<"
    << "май" << "> в строку " << "s1=<" << s1 << "> имеет\nпозицию "
    << left << endl;
cout << "В этой позиции находится символ: <" << s1[ left ] <<
    ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
    << ">" << endl;

right = s1.find_last_of( "май" );
cout << "\nСамое правое вхождение любого символа C-строки \n<"
    << "май" << "> в строку " << "s1=<" << s1 << "> имеет\nпозицию "
    << right << endl;
cout << "В этой позиции находится символ: <" << s1[ right ] <<
    ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]
    << ">" << endl;

left = s1.find_first_of( "яма", 0, 1 );
cout << "\nСамое левое вхождение любого символа подстроки длиной\n"
    "1 символ C-строки <" << "яма" << "> в строку \n" << "s1=<"
    << s1 << "> имеет позицию " << left << endl;
cout << "В этой позиции находится символ: <" << s1[ left ] <<
    ">,\nс этой позиции начинается подстрока: \n<" << &s1[ left ]
    << ">" << endl;

right = s1.find_last_of( "мая", 24, 2 );
cout << "\nСамое правое вхождение любого символа подстроки длиной\n"
    "2 символа C-строки <" << "мая" << "> в строку \n" << "s1=<"
    << s1 << "> до позиции 24\nимеет позицию " << right << endl;
cout << "В этой позиции находится символ: <" << s1[ right ] <<
    ">,\nс этой позиции начинается подстрока: \n<" << &s1[ right ]
    << ">" << endl;

```

```

// Методы find_first_not_of( ) и find_last_not_of( ) - поиск самого
// левого или правого символа одной строки, который не входит
// в другую строку
//*****

cout << "\nТестирование методов find_first_not_of()"
      " и find_last_not_of()"
      << "\n*****"
      "*****" << endl;

s2 = "Сакля";
left = s1.find_first_not_of( s2 );
cout << "\nСамый левый символ строки s1=<" << s1 <<
      ">\nкоторый не входит в строку " << "s2=<" << s2 <<
      ">, имеет позицию " << left << endl;
cout << "В этой позиции находится символ: <" << s1[ left ] <<
      ">, с этой позиции \nначинается подстрока: <" << &s1[ left ]
      << ">" << endl;

right = s1.find_last_not_of( s2 );
cout << "\nСамый правый символ строки s1=<" << s1 <<
      ">\nкоторый не входит в строку " << "s2=<" << s2 <<
      "> имеет позицию " << right << endl;
cout << "В этой позиции находится символ: <" << s1[ right ] <<
      ">, с этой позиции \nначинается подстрока: <" << &s1[ right ]
      << ">" << endl;

left = s1.find_first_not_of( "Сакля" );
cout << "\nСамый левый символ строки s1=<" << s1 <<
      ">\nкоторый не входит в С-строку <" << "Сакля" <<
      "> имеет позицию " << left << "." << endl;
cout << "В этой позиции находится символ: <" << s1[ left ] <<
      ">, с этой позиции \nначинается подстрока: <" << &s1[ left ]
      << ">" << endl;

right = s1.find_last_not_of( "Сакля" );
cout << "\nСамый правый символ строки s1=<" << s1 <<
      ">, \nкоторый не входит в строку в стиле С <" << "Сакля" <<
      "> имеет \nпозицию " << right << "." << endl;
cout << "В этой позиции находится символ: <" << s1[ right ] <<
      ">, с этой позиции \nначинается подстрока: <" << &s1[ right ]
      << ">" << endl << endl;

return 0;
}

```

Листинг 13.12. Результаты выполнения программы

Тестирование методов find() или rfind()

Самое левое вхождение строки s2=<мо> в строку
s1=<C++ - мощный, мощный язык> имеет позицию 6
В этой позиции находится символ: <м>,
с этой позиции начинается подстрока:
<мощный, мощный язык>

Самое правое вхождение строки s2=<мо> в строку
s1=<C++ - мощный, мощный язык> имеет позицию 14
В этой позиции находится символ: <м>,
с этой позиции начинается подстрока:
<мощный язык>

Самое левое вхождение символа <н> в строку
s1=<C++ - мощный, мощный язык> имеет позицию 9
В этой позиции находится символ: <н>,
с этой позиции начинается подстрока:
<ный, мощный язык>

Самое правое вхождение символа <н> в строку
s1=<C++ - мощный, мощный язык> имеет позицию 17
В этой позиции находится символ: <н>,
с этой позиции начинается подстрока:
<ный язык>

Самое левое вхождение C-строки <мо> в строку
s1=<C++ - мощный, мощный язык> имеет позицию 6
В этой позиции находится символ: <м>,
с этой позиции начинается подстрока:
<мощный, мощный язык>

Самое правое вхождение C-строки <мо> в строку
s1=<C++ - мощный, мощный язык> имеет позицию 14
В этой позиции находится символ: <м>,
с этой позиции начинается подстрока:
<мощный язык>

Самое левое вхождение подстроки длиной 2 символа C-строки
<мощный> в строку s1=<C++ - мощный, мощный язык>,
начиная с позиции 0, имеет позицию 6
В этой позиции находится символ: <м>,
с этой позиции начинается подстрока:

<мощный, мощный язык>

Самое правое вхождение подстроки длиной 2 символа C-строки

<мощный> в строку s1=<C++ - мощный, мощный язык>,

начиная с позиции 20, имеет позицию 14

В этой позиции находится символ: <м>,

с этой позиции начинается подстрока:

<мощный язык>

Тестирование методов find_first_of() и find_last_of()

Самое левое вхождение любого символа строки s2=<май>

в строку s1=<C++ - мощный, мощный язык> имеет позицию 6

В этой позиции находится символ: <м>,

с этой позиции начинается подстрока:

<мощный, мощный язык>

Самое правое вхождение любого символа строки s2=<май>

в строку s1=<C++ - мощный, мощный язык> имеет позицию 19

В этой позиции находится символ: <й>

с этой позиции начинается подстрока:

<й язык>

Самое левое вхождение любого символа C-строки

<май> в строку s1=<C++ - мощный, мощный язык> имеет

позицию 6

В этой позиции находится символ: <м>,

с этой позиции начинается подстрока:

<мощный, мощный язык>

Самое правое вхождение любого символа C-строки

<май> в строку s1=<C++ - мощный, мощный язык> имеет

позицию 19

В этой позиции находится символ: <й>

с этой позиции начинается подстрока:

<й язык>

Самое левое вхождение любого символа подстроки длиной

1 символ C-строки <яма> в строку

s1=<C++ - мощный, мощный язык> имеет позицию 21

В этой позиции находится символ: <я>,

с этой позиции начинается подстрока:

<язык>

Самое правое вхождение любого символа подстроки длиной

2 символа C-строки <мая> в строку
 s1=<C++ - мощный, мощный язык> до позиции 24
 имеет позицию 14
 В этой позиции находится символ: <м>,
 с этой позиции начинается подстрока:
 <мощный язык>

Тестирование методов find_first_not_of() и find_last_not_of()

Самый левый символ строки s1=<C++ - мощный, мощный язык>,
 который не входит в строку s2=<Сакля>, имеет позицию 1
 В этой позиции находится символ: <+>, с этой позиции
 начинается подстрока: <++ - мощный, мощный язык>

Самый правый символ строки s1=<C++ - мощный, мощный язык>,
 который не входит в строку s2=<Сакля>, имеет позицию 23
 В этой позиции находится символ: <ы>, с этой позиции
 начинается подстрока: <ык>

Самый левый символ строки s1=<C++ - мощный, мощный язык>,
 который не входит в C-строку <Сакля>, имеет позицию 1.
 В этой позиции находится символ: <+>, с этой позиции
 начинается подстрока: <++ - мощный, мощный язык>

Самый правый символ строки s1=<C++ - мощный, мощный язык>,
 который не входит в строку в стиле C <Сакля>, имеет
 позицию 23.
 В этой позиции находится символ: <ы>, с этой позиции
 начинается подстрока: <ык>

Press any key to continue

Значение *npos*. Если поиск не дает результатов, то поисковые методы возвращают значение `string::npos`. Будьте внимательны при использовании этого значения и его типа. При проверке возвращаемого значения поисковых функций не забывайте, что тип переменной для возвращаемого значения должен быть `string::size_type`, но не `int` или `unsigned`. Иначе сравнение возвращаемого значения с `string::npos` даст неверные результаты.

13.2.4. Сравнение частей строк

В классе `string` для сравнения частей строк можно использовать метод `compare()`, который имеет следующие разновидности.

```
int compare( const string &s ) const;
```

Сравнивает строку `s` с вызывающей строкой. Возвращает отрицательное значение, если вызывающая строка лексикографически меньше строки `s`; возвращает 0 при равенстве строк и положительное значение в остальных случаях.

ЗАМЕЧАНИЕ

Если сравниваются строки разной длины, то перед сравнением длина короткой строки выравнивается до длины более длинной строки добавлением пробельных символов и затем производится сравнение. Эта особенность относится ко всем разновидностям метода `compare()`.

```
int compare( size_type p, size_type n, const string &s ) const;
```

Сравнивает строку `s` с `n` символами вызывающей строки, начиная с позиции `p`. Если параметр `p` выводит за пределы вызывающей строки, то порождается исключение `out_of_range`. Если параметр `n` выводит за пределы вызывающей строки, то в сравнении будет участвовать подстрока вызывающей строки от позиции `p` до ее конца. Возвращает отрицательное значение, если выбранная часть вызывающей строки лексикографически меньше строки `s`; возвращает 0 при равенстве строк и положительное значение в остальных случаях.

```
int compare( size_type p, size_type n, const string &s,  
            size_type p1, size_type n1 ) const;
```

Сравнивает подстроку строки `s` длиной `n1` символов, начинающуюся с позиции `p1`, с `n` символами вызывающей строки, начиная с позиции `p`. Если параметры `p` или `p1` выводят за пределы соответствующих строк, то порождается исключение `out_of_range`. Если параметры `n` или `n1` выводят за пределы соответствующих строк, то в сравнении будут участвовать подстроки соответствующих строк от позиции `p` или `p1` до их конца. Возвращает отрицательное значение, если выбранная часть вызывающей строки лексикографически меньше выбранной подстроки строки `s`; возвращает 0 при равенстве подстрок и положительное значение в остальных случаях.

```
int compare( const char *s ) const;
```

Сравнивает C-строку `s` с вызывающей строкой. Возвращает отрицательное значение, если вызывающая строка лексикографически меньше строки `s`; возвращает 0 при равенстве строк и положительное значение в остальных случаях.

```
int compare( size_type p, size_type n, const char *s,  
            size_type len = npos ) const;
```

Сравнивает подстроку из `len` символов C-строки `s` с `n` символами вызывающей строки, начиная с позиции `p`. Если параметр `p` выводит за пределы вызывающей строки, то порождается исключение `out_of_range`. Если параметр `n` выводит за пределы вызывающей строки, то в сравнении будет участвовать подстрока вызывающей строки от позиции `p` до ее конца. Возвращает отрицательное значение, если выбранная часть вызывающей строки лексикографически меньше подстроки `s`; возвращает 0 при равенстве подстрок и положительное значение в остальных случаях.

ВНИМАНИЕ!

Методы могут давать неверные результаты, если в строке используются символы русского алфавита 'Ё' или 'ё'.

Примеры использования этих методов приведены в листингах 13.13, 13.14.

Листинг 13.13. Файл str7.cpp

```

/*
    Операции над строковыми объектами класса string. Сравнение частей строк
    с помощью методов класса string.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <string>           // Стандартный класс string
#include <iostream>         // Поточковый ввод-вывод

using namespace           // Используем стандартное
    std;                  // пространство имен

int main( void )          // Возвращает 0 при успехе
{
    string    s1( "Ап" ), s2( "Апельсин" );

    // Методы compare( ) - сравнение подстрок
    //*****

    cout << "s1=" << s1 << "\ns2=" << s2 << endl;
    int      rc = s1.compare( 0, 2, s2 );
    if( rc<0 )
        cout << "s1[ 0-1 ]<s2" << endl << endl;
    else if( !rc )
        cout << "s1[ 0-1 ]==s2" << endl << endl;
    else
        cout << "s1[ 0-1 ]>s2" << endl << endl;

    s1 = "Апельсин";
    cout << "s1=" << s1 << "\ns2=" << s2 << endl;
    rc = s1.compare( s2 );
    if( rc<0 )
        cout << "s1<s2" << endl << endl;
    else if( !rc )
        cout << "s1==s2" << endl << endl;
    else
        cout << "s1>s2" << endl << endl;

    s2 = "Селсин";
    cout << "s1=" << s1 << "\ns2=" << s2 << endl;
    rc = s1.compare( 5, 3, s2, 3, 3 );
    if( rc<0 )

```

```

    cout << "s1[ 5-7 ]<s2[ 3-5 ]" << endl << endl;
else if( !rc )
    cout << "s1[ 5-7 ]==s2[ 3-5 ]" << endl << endl;
else
    cout << "s1[ 5-7 ]>s2[ 3-5 ]" << endl << endl;

const char *s = "Апельсин";
cout << "s1=" << s1 << "\ns=" << s << endl;
rc = s1.compare( 0, 8, s );
if( rc<0 )
    cout << "s1[ 0-7 ]<s" << endl << endl;
else if( !rc )
    cout << "s1[ 0-7 ]==s" << endl << endl;
else
    cout << "s1[ 0-7 ]>s" << endl << endl;

cout << "s1=" << s1 << "\ns=" << s << endl;
rc = s1.compare( 0, 8, s, 4 );
if( rc<0 )
    cout << "s1[ 0-7 ]<s[ 0-3 ]" << endl << endl;
else if( !rc )
    cout << "s1[ 0-7 ]==s[ 0-3 ]" << endl << endl;
else
    cout << "s1[ 0-7 ]>s[ 0-3 ]" << endl << endl;

return 0;
}

```

Листинг 13.14. Результаты выполнения программы

```

s1=Ап
s2=Апельсин
s1[ 0-1 ]<s2

```

```

s1=Апельсин
s2=Апельсин
s1==s2

```

```

s1=Апельсин
s2=Селсин
s1[ 5-7 ]==s2[ 3-5 ]

```

```

s1=Апельсин
s=Апельсин
s1[ 0-7 ]==s

```

```
s1=Апельсин
s=Апельсин
s1[ 0-7 ]>s[ 0-3 ]
```

Press any key to continue

13.2.5. Получение характеристик строк

В классе `string` определено несколько методов, позволяющих получить количество элементов строки, длину строки и объем памяти, занимаемой строковым объектом.

Получение количества элементов строки.

```
size_type size( void ) const;
size_type length( void ) const;
```

Методы эквивалентны и возвращают количество элементов строки.

Получение максимальной длины строки.

```
size_type max_size( void ) const;
```

Возвращает максимально возможную длину строки.

Получение текущей емкости строки.

```
size_type capacity( void ) const;
```

Возвращает текущую емкость строки, то есть количество символов, которые можно сохранить в строке без перераспределения ее внутренней памяти. Достаточная емкость строки важна по двум причинам. Во-первых, в результате перераспределения памяти становятся недействительными все ссылки, указатели и итераторы, ссылающиеся на символы строки. Во-вторых, на перераспределение памяти тратится время.

Проверка строки.

```
bool empty( void ) const;
```

Возвращает `true`, если строка пустая.

Примеры использования этих методов приведены в листингах 13.15, 13.16.

Листинг 13.15. Файл `str8.cpp`

```
/*
   Операции над строковыми объектами класса string. Получение характеристик строк
   с помощью методов класса string.
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <string>           // Стандартный класс string
#include <iostream>         // Поточковый ввод-вывод

using namespace           // Используем стандартное
    std;                 // пространство имен

int main( void )          // Возвращает 0 при успехе
```

```

{
    // Методы size( ) и length( ) - получение количества элементов строки
    //*****

    string    s1( "Елка" );

    cout << "s1 = <" << s1 << ">, содержит s1.size( ) = " << s1.size( )
        << " элементов" << endl;
    cout << "s1 = <" << s1 << ">, содержит s1.length( ) = " <<
        s1.length( ) << " элементов" << endl;

    // Метод max_size( ) - получение максимально
    //    возможного количества элементов строки
    //*****

    cout << "\ns1 = <" << s1 << ">, максимальное число "
        "элементов строки м.б. \ns1.max_size( ) = "
        << s1.max_size( ) << " элементов" << endl;

    // Метод capacity( ) - получение текущей емкости строки
    //*****

    cout << "\ns1 = <" << s1 << ">, текущая емкость "
        "строки \ns1.capacity( ) = " << s1.capacity( ) << endl;

    // Метод empty( ) - проверка строки
    //*****

    string    s2( "" );

    cout << "\ns2 = <" << s2 << ">" << endl;
    if( s2.empty( ) )
        cout << "Строка s2 пустая" << endl << endl;
    else
        cout << "Строка s2 непустая" << endl << endl;

    return 0;
}

```

Листинг 13.16. Результаты выполнения программы

```

s1 = <Елка>, содержит s1.size( ) = 4 элементов
s1 = <Елка>, содержит s1.length( ) = 4 элементов

```

```

s1 = <Елка>, максимальное число элементов строки м.б.

```

```
s1.max_size( ) = 4294967293 элементов
```

```
s1 = <Елка>, текущая емкость строки
s1.capacity( ) = 31
```

```
s2 = <>
```

```
Строка s2 пустая
```

```
Press any key to continue
```

В классе `string` имеется еще ряд полезных методов.

Изменение размера строки.

```
void resize( size_type n );
void resize( size_type n, char c );
```

Оба метода изменяют размер строки и делают его равным `n`. Если заданный размер отличается от текущего значения (`size()`), метод присоединяет или удаляет символы в конце строки в соответствии с новым размером. При увеличении количества символов новые символы инициализируются значением `c`. Если в вызове метода отсутствует аргумент, соответствующий параметру `c`, то добавленные символы инициализируются `'\0'`. Если параметр `n` равен `npos`, то генерируется исключение `length_error`.

Резервирование внутренней памяти для строки.

```
void reserve( size_type res_arg = 0 );
```

Метод резервирует внутреннюю память, по крайней мере, для `res_arg` символов. Если `res_arg` меньше текущей емкости, вызов метода интерпретируется как запрос на сокращение емкости, не обязательный для выполнения. Если `res_arg` меньше текущего количества символов, вызов метода интерпретируется как запрос на сокращение емкости по текущему размеру, не обязательный для выполнения. При вызове без аргумента метод всегда интерпретируется как запрос на сокращение емкости по текущему размеру, не обязательный для выполнения. Емкость строки никогда не сокращается до величины, меньшей текущего количества символов. При каждом перераспределении памяти становятся недействительными все ссылки, указатели и итераторы, связанные со строкой. Кроме того, перераспределение требует времени. Поэтому предварительный вызов этого метода повышает скорость работы программы и сохраняет ссылки, указатели и итераторы.

Присоединение символа к строке.

```
string & operator+=( char c );
void push_back( const char c );
```

Методы присоединяют к строке символ `c`. Если размер полученной строки превышает максимально допустимый, то генерируется исключение `length_error`.

Сложение строк.

```
string & operator+=( const string &str );
string & operator+=( const char &s );
```

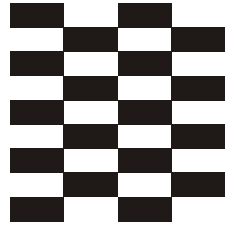
Методы присоединяют к вызывающей строке строку `str` или `s`. Если размер полученной строки превышает максимально допустимый, то генерируется исключение `length_error`.

13.3. Вопросы для самопроверки

1. Чем отличаются C-строки и объекты классов `string` и `wstring` стандартной библиотеки языка C++? Чем отличаются объекты классов `string` и `wstring`?
2. Сравните строковые функции языка C и методы классов `string` и `wstring` стандартной библиотеки языка C++.
3. Перечислите разновидности конструкторов класса `string` и приведите примеры их использования. Роль деструктора класса `string`.
4. Какие операции предусмотрены над строковыми объектами класса `string`?
5. Какие группы методов предусмотрены для обработки частей строк в классе `string`?

Ответы на эти вопросы приведены в *разд. П1.11 приложения 1*.

Глава 14



Строковые потоки

Строковые потоки позволяют считывать и записывать информацию из областей оперативной памяти точно так же, как из файла и в файл, с клавиатуры или на экран. В стандартной библиотеке языка C++ определено три класса строковых потоков:

- ❑ `istream` — входные строковые потоки (наследуется от класса `istream`). Этот класс является специализацией шаблонного класса `basic_istream`:
`typedef basic_istream<char> istream;`
- ❑ `ostream` — выходные строковые потоки (наследуется от класса `ostream`). Этот класс является специализацией шаблонного класса `basic_ostream`:
`typedef basic_ostream<char> ostream;`
- ❑ `stringstream` — двунаправленные строковые потоки (наследуется от класса `iostream`). Этот класс является специализацией шаблонного класса `basic_stringstream`:
`typedef basic_stringstream<char> stringstream;`

Данные классы определяются в заголовочном файле `<sstream>` и, как только что было отмечено, являются производными от классов `istream`, `ostream` и `iostream`. По этой причине они наследуют перегруженные операции "`<<`" и "`>>`", флаги и методы форматирования, манипуляторы, состояния потоков и т. п. (см. гл. 5).

Участки памяти, с которыми выполняются операции чтения и записи, в соответствии со стандартом определяются как строки языка C++ (класс `string`, см. гл. 13). Строковые потоки создаются и связываются с этими участками памяти с помощью конструкторов:

```
// Класс istream
explicit istream( int mode = ios::in );
explicit istream( const string &name, int mode = ios::in );

// Класс ostream
explicit ostream( int mode = ios::out );
explicit ostream( const string &name, int mode = ios::out );

// Класс stringstream
explicit stringstream( int mode = ios::in | ios::out );
```

```
explicit stringstream( const string &name,
                      int mode = ios::in | ios::out );
```

Указанные конструкторы объявлены как **explicit** для того, чтобы они не являлись *конструкторами преобразования типа*. Ключевое слово **explicit** указывает на то, что конструктор будет вызываться явно *только* при создании объекта с типом соответствующего класса.

Можно считать, что строковые потоки являются аналогами библиотечных функций `sscanf()` и `sprintf()` языка C и могут применяться для преобразования данных, когда они сначала заносятся в некоторый участок памяти, а затем считываются оттуда в величины требуемых типов.

Во всех строковых потоках имеется метод `str()`, возвращающий копию строки или устанавливающий ее значение:

```
string str( ) const;      // Возвращает копию строки
// Устанавливает значение строки
void str( const string &s );
```

Проверять строковые потоки на переполнение не требуется, поскольку размер строки изменяется динамически. Рассмотрим пример, иллюстрирующий использование строковых потоков (листинги 14.1, 14.2). Внимательно изучите этот пример — в нем очень много полезной информации!

Листинг 14.1. Файл `stringstream.cpp`

```
/*
   Работа со строковыми потоками istream, ostream и stringstream.
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <sstream>          // Строковые потоки
#include <iostream>         // Поток ввoд-вывoд

using namespace std;      // Используем стандартное
                           // пространство имен

int main( void )          // Возвращает 0 при успехе
{
    // Входные строковые потоки и работа с ними
    // *****

    // Создание входных строковых потоков
    istream
        is, is1; // Пустые потоки
    // Непустой поток
    istream
        is2( "Входной_поток 1 1.5" );

    string    s( "Строка" );
```

```

is1.str( s );           // Инициализируем пустой is1
// Вывод значений созданных строковых потоков на экран
cout << "Работаем с входными строковыми потоками" << endl;
cout << "*****" << endl;
cout << "Пустой поток is:" << is.str( ) << endl;
cout << "Непустой поток is1:" << is1.str( ) << endl;
cout << "Непустой поток is2:" << is2.str( ) << endl;

// Чтение из входного потока is2 и вывод прочитанного
string      s1;
int         i;
double      d;
is2 >> s1 >> i >> d;
if( is2.eof( ) )
    cout << "При чтении в is2 достигнут конец файла" << endl;
if( !is2 )
{
    cout << "Ошибка 10 чтения из is1" << endl;
    exit( 10 );
}
cout << "Прочитали из is2 строку s1:" << s1 << endl;
cout << "Прочитали из is2 целое i:" << i << endl;
cout << "Прочитали из is2 данное с типом double d:" << d << endl;

// Выходные строковые потоки и работа с ними
// *****

// Создание выходных строковых потоков
ostream
    os;           // Пустой поток
// Непустой поток
ostream
    os1( "Выходной строковый поток" );

// Вывод значений созданных строковых потоков на экран
cout << "\nРаботаем с выходными строковыми потоками" << endl;
cout << "*****" << endl;
cout << "Пустой поток os:" << os.str( ) << endl;
cout << "Непустой поток os1:" << os1.str( ) << endl;

// Запись в выходной поток os1 с его начала и вывод записанного
os1 << s1 << " " << i << " " << d << " ";
cout << "Получили после вывода в начало потока os1:\n" << os1.str( )
    << endl;
cout << "Вывод проведен поверх предыдущего вывода с начала потока"
    << endl;

```

```

// Запись в конец выходного потока os1 и вывод записанного
os1.seekp( 0, ios::end );
os1 << " " << s1;
cout << "Получили после вывода в конец потока os1:\n" << os1.str( )
    << endl;

// Двухнаправленные строковые потоки и работа с ними
// *****

// Создание двухнаправленных строковых потоков
stringstream
    ds;                // Пустой поток
// Непустой поток
stringstream
    ds1( "Двухнаправленный строковый поток" );

// Вывод значений созданных строковых потоков на экран
cout << "\nРаботаем с двухнаправленными строковыми потоками" << endl;
cout << "*****" << endl;
cout << "Пустой поток ds:" << ds.str( ) << endl;
cout << "Непустой поток ds1:" << ds1.str( ) << endl;

// Запись в двухнаправленный поток ds1 с его начала и вывод
//   записанного
ds1 << i << " " << d << " ";
cout << "Получили после вывода в начало потока ds1:\n" << ds1.str( )
    << endl;
cout << "Вывод проведен поверх с начала потока" << endl;

// Запись в конец двухнаправленного потока ds1 и вывод записанного
ds1.seekp( 0, ios::end );
ds1 << " " << i << " " << d << " ";
cout << "Получили после вывода в конец потока ds1:\n" << ds1.str( )
    << endl;

// Запись в пустой двухнаправленный поток ds и вывод записанного
ds << " " << 21 << " " << 21.21 << " ";
cout << "Получили после вывода в поток ds:\n" << ds.str( ) << endl;

// Чтение из двухнаправленного потока ds и вывод прочитанного
ds >> i >> d;
if( ds.eof( ) )
    cout << "При чтении в ds достигнут конец файла" << endl;
if( !ds )
{
    cout << "Ошибка 20 чтения из ds" << endl;
}

```

```

        exit( 20 );
    }
    cout << "Прочитали из ds целое i:" << i << endl;
    cout << "Прочитали из ds данное с типом double d:" << d << endl;

    cout << endl;
    return 0;
}

```

Листинг 14.2. Результат работы программы, выводимый на экран

```

Работаем с входными строковыми потоками
*****

Пустой поток is:
Непустой поток is1:Строка
Непустой поток is2:Входной_поток 1 1.5
При чтении в is2 достигнут конец файла
Прочитали из is2 строку s1:Входной_поток
Прочитали из is2 целое i:1
Прочитали из is2 данное с типом double d:1.5

Работаем с выходными строковыми потоками
*****

Пустой поток os:
Непустой поток os1:Выходной строковый поток
Получили после вывода в начало потока os1:
Входной_поток 1 1.5 оток
Вывод проведен поверх предыдущего вывода с начала потока
Получили после вывода в конец потока os1:
Входной_поток 1 1.5 оток Входной_поток

Работаем с двунаправленными строковыми потоками
*****

Пустой поток ds:
Непустой поток ds1:Двунаправленный строковый поток
Получили после вывода в начало потока ds1:
1 1.5 равленный строковый поток
Вывод проведен поверх с начала потока
Получили после вывода в конец потока ds1:
1 1.5 равленный строковый поток 1 1.5
Получили после вывода в поток ds:
21 21.21
Прочитали из ds целое i:21
Прочитали из ds данное с типом double d:21.21

Press any key to continue

```

Теперь, после того, как вы познакомились с этим важным и содержательным примером, в заключение сделаем еще несколько замечаний.

Исходные потоковые классы для строк в стандартной библиотеке C++ были заменены набором новых классов, указанных ранее. Сначала в строковых потоках данных для представления строк использовался тип `char *`. Теперь для этой цели используется только что рассмотренный тип `string`. Прежние классы строковых потоков данных также являлись частью стандартной библиотеки языка C++, но сейчас они считаются устаревшими. Они продолжают поддерживаться для обеспечения совместимости, но могут быть исключены из будущих версий стандарта. Прежние версии в новые программы включаться не будут, а в унаследованном коде произойдет их постепенная замена. Тем не менее, краткое описание "старых" строковых потоков следует привести.

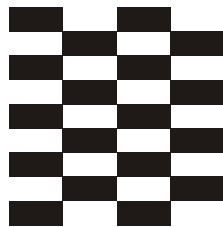
При работе со строковыми потоками данных часто допускают ошибку — забывают извлечь строку методом `str()` и вместо этого используют объект потока данных. С точки зрения компилятора это вполне разумно и допустимо, так как существует преобразование к `void *`. В результате состояние строкового потока выводится в виде адреса.

Как уже указывалось, "устаревшие" потоковые классы `char *` поддерживаются только в целях совместимости. Их интерфейс иногда порождает ошибки, с этими классами часто работают неправильно. Потоковые классы `char *` определены только для символического типа `char`. К этой категории относятся следующие классы:

- `istream` — для чтения из последовательностей символов;
- `ostream` — для записи в последовательности символов;
- `stringstream` — для чтения и записи в последовательности символов;
- `stringstreambuf` — используется как потоковый буфер для потоков `char *`.

Потоковые классы `char *` определяются в заголовочном файле `<sstream>`. Дальнейшее обсуждение "устаревших" строковых потоков `char *` не имеет особого смысла.

Глава 15



Контейнерные классы

Контейнер — это объект, содержащий набор других объектов, организованный определенным образом. Контейнеры предназначены для управления коллекциями объектов определенного типа. У каждой разновидности контейнеров имеются свои достоинства и недостатки, поэтому существование разных контейнеров отражает различие между требованиями к коллекциям в программах. Примерами контейнеров являются массивы (векторы и ассоциативные массивы) и списки (собственно списки, очереди, стеки). Как правило, в контейнер можно добавлять объекты и удалять их из него. Работа с контейнерами поддерживается в стандартной библиотеке с помощью контейнерных классов.

Для каждого типа контейнера в соответствующем контейнерном классе определены методы для работы с его элементами (объектами), не зависящие от конкретного типа объектов, которые хранятся в контейнере. По этой причине один и тот же вид контейнера можно использовать для хранения и работы с объектами различных типов. Эта возможность реализована с помощью *шаблонов классов*.

Зачем же нужны контейнеры, чем они хороши? Использование контейнеров позволяет значительно повысить *надежность* программ, их *переносимость* и *универсальность*. При этом одновременно *уменьшаются сроки* разработки и *стоимость* разработки таких программ. Но, повторяем еще раз, что "за удовольствия надо платить". Универсальность и безопасность контейнерных классов не могут не сказаться на быстродействии использующих их программ. Снижение быстродействия, в зависимости от реализации компилятора языка C++, может оказаться весьма значительным. Уместно еще раз заметить также, что на освоение стандартной библиотеки языка C++ придется потратить немалые время и усилия.

Контейнеры STL можно разделить на последовательные и ассоциативные контейнеры.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности и имеют следующие разновидности:

- ☐ векторы (`vector`);
- ☐ двусторонние очереди или, иначе, деки (`deque`);
- ☐ списки (`list`);
- ☐ стеки (`stack`);
- ☐ очереди (`queue`);
- ☐ очереди с приоритетами (`priority_queue`).

В последовательном контейнере каждый объект (элемент) занимает определенную позицию, которая зависит только от времени и места вставки элемента в контейнер, но не зависит от значения элемента. Каждый из перечисленных контейнеров обеспечивает свой набор действий над содержащимися в нем объектами. Выбор контейнера зависит от того, что требуется делать с объектами в программе. Например, если требуется часто вставлять и удалять элементы из середины последовательности, то следует использовать список. Если же включение или удаление объектов выполняется чаще в конце или начало последовательности, то лучше использовать двустороннюю очередь.

Первые три из перечисленных последовательных контейнеров являются основными, а остальные — вспомогательными. В реализации вспомогательных контейнеров используются основные контейнеры.

Ассоциативные контейнеры представляют собой *отсортированные коллекции*, в которых позиция объекта (элемента) зависит от его значения по определенному критерию сортировки. Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу и построены на основе сбалансированных деревьев. Существуют следующие разновидности ассоциативных контейнеров:

- словари (`map`);
- словари с дубликатами (`multimap`);
- множества (`set`);
- множества с дубликатами (`multiset`);
- битовые множества (`bitset`).

Обратите внимание на то, что перечисленные ассоциативные контейнеры полностью независимы друг от друга. Они имеют *разные* реализации и не являются производными друг от друга.

Автоматическая сортировка элементов в ассоциативных контейнерах не означает, что они специально предназначены для сортировки элементов. С таким же успехом можно отсортировать элементы последовательного контейнера. Основным достоинством автоматической сортировки в ассоциативных контейнерах является более высокая эффективность поиска в них элемента с заданными свойствами. При использовании в них двоичного поиска его сложность возрастает логарифмически, а не линейно. Например, для поиска в коллекции из 1000 элементов понадобится всего 10 (ближайшее большее целое от $\log_2 1000$) сравнений. Таким образом, автоматическая сортировка элементов является только полезным "побочным эффектом" реализации ассоциативного контейнера, спроектированного в расчете на повышение эффективности поиска.

На основе контейнерных классов, имеющихся в стандартной библиотеке языка C++, можно создавать собственные (пользовательские) контейнерные классы.

Различные контейнерные классы имеют много общего, и это очень удобно. В частности, контейнерные классы имеют *стандартизованный интерфейс*. Это означает, что смысл одноименных операций (методов) и данных-членов в различных контейнерных классах одинаков и применим ко всем типам контейнеров. Стандарт языка C++ определяет только интерфейс контейнеров. Поэтому разные реализации контейнеров могут сильно отличаться по эффективности.

Общие возможности контейнеров. Существуют три основных требования, которые должны выполняться всеми контейнерами STL.

- ❑ Контейнеры должны поддерживать семантику значений вместо ссылочной семантики. Это означает, что при вставке элемента контейнер должен создавать его внутреннюю копию, вместо того чтобы сохранять ссылку на внешний объект.
- ❑ Элементы в контейнере должны располагаться в определенном порядке. Это означает, что при повторном переборе элементов контейнера с применением итератора порядок перебора элементов должен остаться прежним. Итераторы представляют собой основной интерфейс для работы алгоритмов STL.
- ❑ В общем случае, операции с элементами контейнеров не безопасны. Вызывающая сторона должна проследить за тем, чтобы параметры операции соответствовали требованиям. Нарушение правил (например, использование недействительного индекса) приведет к непредсказуемым последствиям.

Унифицированные типы, используемые в контейнерах. Практически в любом контейнерном классе определены следующие унифицированные типы.

- ❑ Тип элемента контейнера (`value_type`).
- ❑ Тип индексов, счетчиков элементов и т. д. (`size_type`).
- ❑ Итератор (`iterator`). *Итератор* является аналогом указателя на элемент (объект), хранимый в контейнере (ведет себя подобно `value_type *`). Все, что требуется от итератора, — уметь ссылаться на элемент контейнера и реализовывать операцию перехода к его следующему элементу.
- ❑ Константный итератор (`const_iterator`). *Константный итератор* используется тогда, когда значения соответствующих элементов контейнера не изменяются (ведет себя подобно `const value_type *`).
- ❑ Обратный итератор (`reverse_iterator`). Просматривает контейнер в обратном порядке (ведет себя подобно `value_type *`).
- ❑ Константный обратный итератор (`const_reverse_iterator`). Просматривает контейнер в обратном порядке (ведет себя подобно `const value_type *`).
- ❑ Ссылка на элемент (`reference`). Ведет себя подобно `value_type &`.
- ❑ Константная ссылка на элемент (`const_reference`). Ведет себя подобно `const value_type &`.
- ❑ Тип ключа (`key_type`). Используется только для *ассоциативных* контейнеров.
- ❑ Тип критерия сравнения (`key_compare`). Используется только для *ассоциативных* контейнеров.

Общие операции и методы над контейнерами. При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах объектов, находящихся в контейнере. Для этого в каждом контейнере определены перечисленные далее методы.

```
iterator begin( );  
const_iterator begin( );
```

Указывают на первый элемент контейнера.

```
iterator end( );  
const_iterator end( );
```

Указывают на элемент контейнера, *следующий за последним* (но не последний элемент контейнера).

```
reverse_iterator rbegin( );  
const_reverse_iterator rbegin( );
```

Указывают на первый элемент контейнера в обратной последовательности.

```
reverse_iterator rend( );  
const_reverse_iterator rend( );
```

Указывают на элемент контейнера, следующий за последним, в обратной последовательности.

Во всех контейнерах имеются методы, позволяющие определить *характеристики контейнера*.

```
size_type size( ) const;
```

Возвращает число элементов контейнера.

```
size_type max_size( ) const;
```

Возвращает максимально возможное число элементов контейнера (чуть больше четырех миллиардов).

```
bool empty( ) const;
```

Возвращает **true**, если контейнер пуст.

Во всех контейнерах имеются конструкторы, позволяющие создавать и инициализировать объекты-контейнеры. К таким конструкторам относятся конструктор умолчания (без параметров), обычный конструктор с инициализацией элементов элементами другого контейнера, конструктор копирования и деструктор.

В контейнерных классах предусмотрены операции сравнения объектов-контейнеров ("==", "!=", "<", ">", "<=", ">="). Перечисленные операции сравнения определяются по следующим правилам:

- ☐ сравниваемые контейнеры должны относиться к одному типу;
- ☐ два контейнера равны, если их элементы совпадают и следуют в одинаковом порядке;
- ☐ отношение "меньше/больше" между контейнерами проверяется по лексикографическому критерию.

Вместе с тем заметим, что с использованием алгоритмов сравнения можно сравнивать разнотипные контейнеры.

Во всех контейнерных классах предусмотрены операция присваивания и метод `swap()`. В процессе присваивания все элементы контейнера-приемника удаляются и после этого все элементы контейнера-источника копируются в контейнер-приемник. Эта операция является дорогостоящей и имеет линейную сложность. Для повышения эффективности можно использовать метод `swap()`, который *меняет местами два контейнера*. Этот метод требует, чтобы контейнеры были одинакового типа. Метод `swap()` реализован так, что имеет не линейную, а постоянную сложность.

Во всех контейнерных классах имеется метод `clear()`, *удаляющий из контейнера все элементы*, но, в отличие от деструктора, этот метод не уничтожает контейнер.

Стандартная библиотека классов определена в следующих стандартных заголовочных файлах:

```
<algorithm> <deque> <functional> <iterator> <list> <map>
<memory> <numeric> <queue> <set> <stack>
<utility> <vector>
```

Последовательные контейнеры — векторы (`vector`), двусторонние очереди (`deque`) и списки (`list`) — поддерживают разные наборы операций, среди которых имеются совпадающие операции (табл. 15.1) с различной эффективностью выполнения.

Таблица 15.1. Совпадающие операции последовательных контейнеров и их эффективность

Операция	Метод	Вид последовательного контейнера		
		vector	deque	list
Вставка в начало	<code>push_front()</code>	Отсутствует	+	+
Удаление из начала	<code>pop_front()</code>	Отсутствует	+	+
Вставка в конец	<code>push_back()</code>	+	+	+
Удаление из конца	<code>pop_back()</code>	+	+	+
Вставка в произвольное место	<code>insert()</code>	(+)	(+)	+
Удаление из произвольного места	<code>erase()</code>	(+)	(+)	+
Произвольный доступ к элементу	<code>[]</code> или <code>at()</code>	+	+	Отсутствует

В таблице приняты следующие обозначения. Знак "+" означает, что соответствующая операция реализуется за постоянное время, не зависящее от количества *n* объектов, помещенных в контейнер (постоянная сложность). Знак "(+)" означает, что соответствующая операция реализуется за время, пропорциональное *n* (линейная сложность). Важно отметить, что для малых значений *n* время выполнения операций, отмеченных знаком "+", может превышать время выполнения операций, обозначенных знаком "(+)". В общем же случае это не так.

Анализ таблицы позволяет сделать следующие важные выводы относительно областей применения последовательных контейнеров.

- ❑ Последовательный контейнер *вектор* эффективно реализует произвольный доступ к элементам, добавление в конец и удаление элемента из конца.
- ❑ *Двусторонняя очередь (дек)* эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление элементов из обоих концов.
- ❑ *Список* эффективно реализует вставку и удаление элементов в произвольное место, но не имеет произвольного доступа к своим элементам.

15.1. Последовательные контейнеры.

Вектор (*vector*)

Вектором (одномерным массивом) называется абстрактная модель, имитирующая динамический массив при операциях с элементами. Однако стандарт не утверждает, что в реализации вектора должен использоваться именно динамический массив.

Определение шаблона классов `vector` приведено в стандартном заголовочном файле `<vector>` в стандартном пространстве имен `std` и представлено в листинге 15.1 (дана упрощенная запись).

Листинг 15.1. Шаблон классов `vector`

```
template < class T, class A = allocator < T > >
// T - тип элементов вектора, поддерживающий присваивание и копирование,
// необязательный второй параметр шаблона определяет модель памяти (по
// умолчанию используется модель allocator, определенная в стандартной
// библиотеке)
class vector
{
    public:

        // Определение типов
        typedef A                               allocator_type;
        typedef A::size_type                    size_type;
        typedef A::difference_type              difference_type;
        typedef typename A::reference            reference;
        // typename означает, что A::reference является типом
        typedef typename A::const_reference      const_reference;
        typedef typename A::pointer              pointer;
        typedef typename A::const_pointer        const_pointer;
        typedef T                                value_type;
        typedef A::pointer                       iterator;
        typedef A::const_pointer                  const_iterator;
        typedef std::reverse_iterator <const_iterator>
            const_reverse_iterator;
        typedef std::reverse_iterator<iterator>
            reverse_iterator;

        // Конструкторы
        // 1: конструктор умолчания
        explicit vector( const A &a1 = A( ) );
        // 2: разновидность обычного конструктора
        explicit vector( size_type n, const T &v = T( ),
            const A &a1 = A( ) );
```

```

// 3: конструктор копирования
vector( const vector < T, A > &x );
// 4: разновидность обычного конструктора
vector( const_iterator first, const_iterator last,
        const A &a1 = A( ) );

// Деструктор
~vector( );

// Интерфейсные методы
// 5: операция присваивания
vector < T, A > & operator=( const vector < T, A > &x );
// 13: резервирует память под вектор из n элементов
void reserve( size_type n );
// 12: под какое количество элементов зарезервирована память
size_type capacity( ) const;
iterator begin( );
const_iterator begin( ) const;
iterator end( );
const_iterator end( ) const;
reverse_iterator rbegin( );
const_reverse_iterator rbegin( ) const;
reverse_iterator rend( );
const_reverse_iterator rend( ) const;
// 14: устанавливает размер вектора n элементов
void resize( size_type n, const T &x = T( ) );
size_type size( ) const;
size_type max_size( ) const;
bool empty( ) const;
// Возвращает модель памяти контейнера
allocator_type get_allocator( ) const;
// 8: доступ к элементу вектора по индексу с проверкой выхода
// индекса за границу вектора
const_reference at( size_type pos ) const;
reference at( size_type pos );
// 9: доступ к элементу вектора по индексу без проверки выхода
// индекса за границу вектора
const_reference operator[( size_type pos ) const;
reference operator[( size_type pos );
// 10: доступ к первому элементу вектора
reference front( );
const_reference front( ) const;
// 11: доступ к последнему элементу вектора
reference back( );
const_reference back( ) const;

```

```

// 15: добавление элемента в конец вектора
void                push_back( const T &x );
// 16: удаление элемента из конца вектора
void                pop_back( );
// 6: копирование
void                assign( const_iterator first,
                           const_iterator last );
// 7: копирование
void                assign( size_type n, const T &x );
// 17: вставка элемента в заданное место вектора
iterator            insert( iterator pos, const T &x );
// 18: вставка нескольких элементов в заданное место вектора
void                insert( iterator pos, size_type m,
                           const T &x );
// 19: вставка группы элементов, заданных диапазоном итераторов,
//     в заданное место вектора
void                insert( iterator pos,
                           const_iterator first, const_iterator last );
// 20: удаление заданного элемента вектора
iterator            erase( iterator pos );
// 21: удаление нескольких элементов вектора, заданных диапазоном
//     итераторов
iterator            erase( iterator first,
                           iterator last );

// 22: очистка вектора
void                clear( );
// 23: обмен векторов
void                swap( vector < T, A > &x );

protected:

    A                allocator;
};

```

Элементы вектора копируются во внутренний динамический массив и хранятся в определенном порядке. Следовательно, вектор относится к категории *упорядоченных коллекций*. Вектор обеспечивает *произвольный доступ* к своим элементам. Это означает, что обращение к любому элементу вектора с известной позицией выполняется напрямую и с постоянным временем. Итераторы векторов являются итераторами прямого доступа и это позволяет применять к векторам все алгоритмы STL.

Операции присоединения и удаления элементов в конце вектора выполняются с высоким быстродействием. Если элементы вставляются или удаляются в середине или начале вектора, то быстродействие снижается, поскольку все элементы в последующих позициях приходится перемещать на новое место. Один из способов повышения быстродействия векторов заключается в выделении для вектора большего объема памяти, чем необходимо для хранения всех элементов. Чтобы эффективно и пра-

вильно использовать векторы, нужно понимать, как связаны между собой размер и емкость вектора.

Векторы поддерживают стандартные операции проверки размера `size()`, `max_size()` и `empty()`. К ним добавляется метод `capacity()`, возвращающий максимальное количество элементов, которые могут храниться в текущей выделенной памяти. Если количество элементов вектора превысит значение, возвращаемое методом `capacity()`, то вектору придется перераспределить свою внутреннюю память. Как уже указывалось ранее, это нежелательно по двум причинам:

- ❑ в результате перераспределения памяти становятся недействительными все ссылки, указатели и итераторы элементов вектора;
- ❑ на перераспределение памяти требуется время.

Следовательно, если программа поддерживает указатели, ссылки и итераторы вектора, а также при важности быстродействия нужно перед началом работы с вектором с помощью метода `reserve()` задать для него достаточную емкость (на наихудший случай или с запасом).

Концепция емкости векторов сходна с концепцией емкости для строк, но имеет отличие. Метод `reserve()` не может вызываться для уменьшения емкости векторов — его вызов с аргументом, меньшим текущей емкости вектора, *игнорируется*. Однако способ косвенного уменьшения емкости вектора все же существует — достаточно с помощью метода `swap()` поменять местами содержимое двух векторов. При этом меняются местами не только элементы векторов, но и их емкости. После вызова этого метода все ссылки, итераторы и указатели останутся действительными, но переключатся на новые контейнеры.

Конструкторы и получение характеристик векторов. Для создания вектора предусмотрена возможность использования нескольких конструкторов (см. ранее объявление шаблона классов `vector`). Обратите внимание на то, что служебное слово **explicit** используется в конструкторах для того, чтобы при создании объекта запретить неявное преобразование при присвоении создаваемому объекту-вектору значения другого типа.

Конструктор 1 является конструктором умолчания. С его помощью создается пустой вектор.

Конструктор 2 является разновидностью обычного конструктора. С его помощью создается вектор длиной `n` элементов. Конструктор инициализирует элементы вектора одинаковыми значениями `v`. Поскольку изменение размера вектора обходится дорого, то при его создании задавать начальный размер весьма полезно. При этом для встроенных типов производится инициализация каждого элемента значением `v`. Если оно не указано, то элементы глобальных векторов инициализируются нулем. Если тип элемента вектора определен пользователем, то начальное значение формируется с помощью конструктора по умолчанию для данного типа. На месте второго параметра можно написать вызов конструктора с параметрами, создав таким образом вектор элементов с требуемыми свойствами.

Конструктор 4 является другой разновидностью обычного конструктора. С его помощью вектор создается путем копирования указанного с помощью итераторов-параметров диапазона элементов другого вектора. Итераторы-параметры при этом должны быть константными.

Конструктор 3 является конструктором копирования.

Рассмотрим пример, иллюстрирующий использование перечисленных конструкторов (листинги 15.2, 15.3).

Листинг 15.2. Файл vec1.cpp

```
/*
    Последовательный контейнер vector. Конструкторы и получение характеристик
    объектов-контейнеров с помощью методов контейнера vector.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Потокковый ввод-вывод
#include <string>             // Стандартный класс string
#include <vector>             // Последовательный контейнер

using namespace             // Используем стандартное
    std;                    // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // 1: конструктор умолчания
    vector< int >
        v;
    cout << "Число элементов в v: " << v.size( ) << endl;
    if( v.empty( ) )
        cout << "Вектор v пустой" << endl;
    cout << "В v может храниться " << v.capacity( )
        << " элементов без перераспределения памяти" << endl;

    // 2: разновидности обычного конструктора

    // Создается вектор из 10 элементов целого типа с единичными
    // значениями
    vector< int >
        v1( 10, 1 );
    cout << "\nЧисло элементов в v1: " << v1.size( ) << endl;
    cout << "v1[ 0 ] = " << v1[ 0 ] << endl;
    cout << "v1[ 9 ] = " << v1[ 9 ] << endl;

    // Создается вектор из 5 элементов типа string (инициализацию
    // элементов выполняет конструктор умолчания string - пустыми
    // строками)
    vector< string >
        v2( 5 );
```

```

cout << "\nЧисло элементов в v2: " << v2.size( ) << endl;
cout << "v2[ 0 ] = " << v2[ 0 ] << endl;

// Создается вектор из 6 элементов типа string (элементы
// инициализируются строкой "Вася")
vector< string >
    v3( 6, string( "Вася" ) );
cout << "\nЧисло элементов в v3: " << v3.size( ) << endl;
cout << "v3[ 0 ] = " << v3[ 0 ] << endl;

// 4: разновидности обычного конструктора

// Создается вектор из первых трех элементов вектора v1
vector< int >
    v4( v1.begin( ), v1.begin( )+3 );
cout << "\nЧисло элементов в v4: " << v4.size( ) << endl;
cout << "v4[ 0 ] = " << v4[ 0 ] << endl;

// 3: конструктор копирования. Копируется вектор v1
vector< int >
    v5( v1 );
cout << "\nЧисло элементов в v5: " << v5.size( ) << endl;
cout << "v5[ 0 ] = " << v5[ 0 ] << endl << endl;

return 0;
}

```

Листинг 15.3. Результаты выполнения программы

```

Число элементов в v: 0
Вектор v пустой
В v может храниться 0 элементов без перераспределения памяти

Число элементов в v1: 10
v1[ 0 ] = 1
v1[ 9 ] = 1

Число элементов в v2: 5
v2[ 0 ] =

Число элементов в v3: 6
v3[ 0 ] = Вася

Число элементов в v4: 3
v4[ 0 ] = 1

Число элементов в v5: 10
v5[ 0 ] = 1

Press any key to continue

```

Присваивание и копирование векторов. Прототипы 5—7 методов для присваивания и копирования векторов приведены ранее в объявлении шаблона классов `vector`.

С помощью метода 5, перегружающего операцию присваивания, вектора можно присваивать друг другу точно так же, как стандартные типы данных или строки. После присваивания размер вектора (левого операнда) становится равным размеру правого операнда (вектора). Все элементы левого операнда удаляются и заменяются элементами правого операнда. Пример использования операции присваивания векторов приведен далее.

Формат 6 метода `assign()` осуществляет копирование в начало вызывающего вектора значений другого вектора из диапазона, определяемого итераторами `first` и `last`. Новый размер вызывающего вектора при этом будет равен `last-first`.

Формат 7 метода `assign()` осуществляет копирование в первые `n` (первый параметр) элементов в начале вызывающего вектора заданного значения `x` (второй параметр). После копирования новый размер вызывающего вектора становится равным `n`.

Пример использования этих методов приведен в листингах 15.4, 15.5.

Листинг 15.4. Файл `vec2.cpp`

```
/*
    Последовательный контейнер vector. Присваивание и копирование векторов с помощью
    методов контейнера vector.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <vector>            // Последовательный контейнер

using namespace            // Используем стандартное
    std;                  // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    // 5: присваивание векторов
    vector< int >
        v1( 5, 1 ), v2( 7, 5 ), v3( 10, 3 );
    v3 = v2 = v1;
    cout << "v3 = v2 = v1;" << endl;
    cout << "Число элементов в v1: " << v1.size() << ", v1[ 0 ] = "
        << v1[ 0 ] << ", v1[ 4 ] = " << v1[ 4 ] << endl;
    cout << "Число элементов в v2: " << v2.size() << ", v2[ 0 ] = "
        << v2[ 0 ] << endl;
    cout << "Число элементов в v3: " << v3.size() << ", v3[ 4 ] = "
        << v3[ 4 ] << endl;

    // 6, 7: копирование векторов
```

```

vector< int >
    v4( 4, 11 );
// Первым 3 элементам вектора v4 присваивается значение 12
v4.assign( 3, 12 );
cout << "\nv4.assign( 3, 12 );" << endl;
cout << "Число элементов в v4: " << v4.size( ) << ", v4[ 0 ] = "
    << v4[ 0 ] << ", v4[ 2 ] = " << v4[ 2 ] << endl;
vector< int >
    v5( 5, 1 ), v6( 7, 5 );
// Первым 2 элементам вектора v6 присваиваются значения v5[ 2 ],
//   v5[ 3 ]
v6.assign( v5.begin( )+2, v5.begin( )+4 );
cout << "\nv6.assign( v5.begin( )+2, v5.begin( )+4 );" << endl;
cout << "v5[ 2 ] = " << v5[ 2 ] << ", v5[ 3 ] = " << v5[ 3 ] << endl;
cout << "Число элементов в v6: " << v6.size( ) << ", v6[ 0 ] = "
    << v6[ 0 ] << ", v6[ 1 ] = " << v6[ 1 ] << endl << endl;

return 0;
}

```

Листинг 15.5. Результаты выполнения программы

```

v3 = v2 = v1;
Число элементов в v1: 5, v1[ 0 ] = 1, v1[ 4 ] = 1
Число элементов в v2: 5, v2[ 0 ] = 1
Число элементов в v3: 5, v3[ 4 ] = 1

v4.assign( 3, 12 );
Число элементов в v4: 3, v4[ 0 ] = 12, v4[ 2 ] = 12

v6.assign( v5.begin( )+2, v5.begin( )+4 );
v5[ 2 ] = 1, v5[ 3 ] = 1
Число элементов в v6: 2, v6[ 0 ] = 1, v6[ 1 ] = 1

Press any key to continue

```

Итераторы. Итераторы шаблона классов `vector` типичны для контейнеров и рассмотрены ранее.

Доступ к элементам вектора осуществляется с помощью операций и методов шаблона классов `vector` (см. в листинге 15.1 прототипы 8—11).

Операция `"[]"` с прототипами 9 осуществляет доступ к элементу вектора по индексу *без проверки* его выхода за границы вектора. Метод `at()` с прототипами 8 осуществляет аналогичную операцию, но *с проверкой* выхода индекса за границу вектора. При нарушении индексации метод порождает исключение `out_of_range`. Естественно, что метод `at()` работает медленнее, чем операция `"[]"`. Поэтому в случаях, когда

диапазон индексов задан явно, для доступа к элементу вектора предпочтительнее использование операции "[]".

Операции доступа к элементу вектора возвращают значение ссылки на элемент (reference) или константной ссылки (const_reference) в зависимости от того, применяются ли они к константному объекту или нет.

Методы front() с прототипами 10 и back() с прототипами 11 возвращают ссылки соответственно на первый и последний элемент вектора.

Пример использования методов и операций доступа к элементам вектора приводится в листингах 15.6, 15.7.

Листинг 15.6. Файл vec3.cpp

```
/*
    Последовательный контейнер vector. Доступ к элементам вектора с помощью методов
    и операций контейнера vector.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <vector>             // Последовательный контейнер

using namespace std;        // Используем стандартное
                             // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Операция [] и метод at( )

    vector< int >
        v1( 4, 1 );
    cout << "Возможное число элементов в v1 без его перераспределения:"
         << "\n v1.capacity( ) = " << v1.capacity( ) << endl;
    cout << "\nЧисло элементов в v1: v1.size( ) = " << v1.size( ) <<
         << ", v1[ 0 ] = " << v1[ 0 ] << ", \nv1.at( 1 ) = " << v1.at( 1 )
         << ", v1.at( 2 ) = " << v1.at( 2 ) << ", v1[ 3 ] = " << v1[ 3 ]
         << endl;
    v1[ 1 ] += 4;
    cout << "\nВыполняем операцию v1[ 1 ] += 4;" <<
         << "\nРезультат операции: v1[ 1 ] = " << v1[ 1 ] << endl;
    v1.at( 2 ) += 2;
    cout << "\nВыполняем операцию v1.at( 2 ) += 2;" <<
         << "\nРезультат операции: v1.at( 2 ) = " << v1.at( 2 ) << endl;

    // 10: метод front( ) - доступ к первому элементу вектора
    // 11: метод back( ) - доступ к последнему элементу вектора
```

```

v1.front( ) -= 10;
cout << "\nВыполняем операцию v1.front( ) -= 10;" <<
    "\nРезультат операции: v1.front( ) = v1[ 0 ] = " << v1.front( )
    << endl;
v1.back( ) -= 5;
cout << "\nВыполняем операцию v1.back( ) -= 5;" <<
    "\nРезультат операции: v1.back( ) = v1[ 3 ] = " << v1.back( )
    << endl << endl;

return 0;
}

```

Листинг 15.7. Результаты выполнения программы

Возможное число элементов в v1 без его перераспределения:

```
v1.capacity( ) = 4
```

Число элементов в v1: v1.size() = 4, v1[0] = 1,
v1.at(1) = 1, v1.at(2) = 1, v1[3] = 1

Выполняем операцию v1[1] += 4;
Результат операции: v1[1] = 5

Выполняем операцию v1.at(2) += 2;
Результат операции: v1.at(2) = 3

Выполняем операцию v1.front() -= 10;
Результат операции: v1.front() = v1[0] = -9

Выполняем операцию v1.back() -= 5;
Результат операции: v1.back() = v1[3] = -4

Press any key to continue

Получение размера вектора. Для получения количества элементов, под хранение которых в векторе выделена память, можно использовать метод `capacity()` шаблона классов `vector` (см. в листинге 15.1 прототип 12). Обратите внимание, что возвращаемое этим методом значение в общем случае больше или равно значению, возвращаемому методом `size()`. Пример, иллюстрирующий использование этих методов, только что рассмотрен (файл `vec3.cpp`).

Память под вектор выделяется динамически, но не под один элемент в каждый момент времени (это было бы неэффективным), а сразу же под группу элементов. Перераспределение памяти происходит только при превышении этого количества элементов. При этом объем выделенного пространства удваивается. После перераспределения памяти любые итераторы, ссылающиеся на элементы вектора, становятся

недействительными, так как вектор может быть перемещен в другой участок памяти. После этого нельзя ожидать, что связанные с вектором ссылки будут автоматически обновлены.

Резервирование места под возможное расширение вектора и изменение размера вектора.

Для размещения вектора из заданного количества элементов n можно использовать метод `reserve()` шаблона классов `vector` (см. в листинге 15.1 прототип 13). После вызова этого метода значение, возвращаемое методом `capacity()`, будет больше или равно n . Этот метод полезно применять тогда, когда размер вектора известен заранее.

Для изменения размеров существующего вектора можно использовать метод `resize()` шаблона классов `vector` (см. в листинге 15.1 прототип 14). Этот метод увеличивает или уменьшает размер вектора в зависимости от фактического размера вектора (этот размер равен значению, возвращаемому методом `size()`). При увеличении размера вектора в его конец добавляется соответствующее количество элементов, значение которого определяется вторым параметром x .

Проиллюстрируем использование этих методов примером из листингов 15.8, 15.9.

Листинг 15.8. Файл `vec4.cpp`

```
/*
    Последовательный контейнер vector. Резервирование памяти под вектор заданного
    размера и изменение размера вектора с помощью методов контейнера vector.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <vector>            // Последовательный контейнер

using namespace            // Используем стандартное
    std;                  // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    vector< int >
        v1, v2( 5, 2 );

    // 13: метод reserve( ) - резервирует память под вектор из 15
    // элементов

    v1.reserve( 15 );
    cout << "Выполняем операцию v1.reserve( 15 );" <<
        "\nРезультат операции - фактический размер v1: " << v1.size( )
        << "\nЗарезервированный размер v1: " << v1.capacity( ) << endl;
```

```
// 14: метод resize( ) - изменяет размер вектора v2 до 10 элементов

cout << "\nРазмер v2: " << v2.size( ) << endl;
v2.resize( 10, 21 );
cout << "Выполняем операцию v2.resize( 10, 21 );" <<
    "\nРезультат операции - фактический размер v2: " << v2.size( )
    << "\nv2[ 0 ] = " << v2[ 0 ] << ", v2[ 9 ] = " << v2[ 9 ]
    << endl << endl;

return 0;
}
```

Листинг 15.9. Результаты выполнения программы

```
Выполняем операцию v1.reserve( 15 );
Результат операции - фактический размер v1: 0
Зарезервированный размер v1: 15

Размер v2: 5
Выполняем операцию v2.resize( 10, 21 );
Результат операции - фактический размер v2: 10
v2[ 0 ] = 2, v2[ 9 ] = 21

Press any key to continue
```

Изменение вектора. Для изменения вектора в шаблоне классов `vector` имеется множество методов (см. в листинге 15.1 прототипы 15—23).

Метод `push_back()` с прототипом 15 добавляет элемент в конец вектора, а метод `pop_back()` с прототипом 16 удаляет элемент из конца вектора.

Методы `insert()` служат для вставки элемента или группы элементов в вектор. *Первая форма* метода `insert()` с прототипом 17 служит для вставки элемента со значением, указанным во втором аргументе, в место вектора, определяемое первым аргументом. Метод возвращает итератор, ссылающийся на вставленный элемент. Вставка в вектор занимает время, пропорциональное количеству сдвигаемых на новые позиции элементов. При этом если новый размер вектора превышает зарезервированный размер вектора, то происходит перераспределение памяти. Если при вставке перераспределения памяти не происходит, то все итераторы сохраняют свои значения. В противном случае итераторы становятся недействительными. *Вторая форма* метода `insert()` с прототипом 18 служит для вставки группы элементов со значением, указанным в третьем аргументе, в место вектора, определяемое первым аргументом. Количество вставляемых элементов задается вторым аргументом в вызове метода. *Третья форма* метода `insert()` с прототипом 19 служит для вставки группы элементов из диапазона, указанного итераторами (второй и третий аргументы), в место вектора, определяемое первым аргументом. Обратите внимание, что третий аргумент в вызове этой разновидности метода задает *не последний* вставляемый элемент, а элемент, *следующий* за ним.

Методы `erase()` служат для удаления одного элемента вектора (прототип 20) или нескольких элементов (прототип 21), диапазон которых задан итераторами. Каждый вызов метода `erase()`, так же как и в случае вставки элементов в вектор, занимает время, пропорциональное количеству сдвигаемых на новые позиции элементов вектора. Все итераторы и ссылки "правее" места удаления становятся недействительными.

Метод `clear()` с прототипом 22 выполняет очистку вектора, а метод `swap()` с прототипом 23 меняет векторы местами. Обратите внимание на то, что при обмене размеры векторов *могут* отличаться.

Использование этих методов иллюстрирует пример из листингов 15.10, 15.11.

Листинг 15.10. Файл `vec5.cpp`

```
/*
    Последовательный контейнер vector. Изменение вектора с помощью методов
    контейнера vector.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <vector>             // Последовательный контейнер

using namespace std;        // Используем стандартное
                             // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    vector< int >
        v1( 5, 2 );

    // 15: метод push_back( ) - добавляет элемент со значением 22 в конец
    // вектора

    cout << "Размер v1: " << v1.size( ) << endl;
    v1.push_back( 22 );
    cout << "Выполняем операцию v1.push_back( 22 );" <<
        "\nРезультат операции - размер v1: " <<
        v1.size( ) << ", v1[ 5 ] = " << v1[ 5 ] << endl;

    // 16: метод pop_back( ) - удаляет элемент из конца вектора

    v1.pop_back( );
    cout << "\nВыполняем операцию v1.pop_back( );" <<
        "\nРезультат операции - размер v1: " << v1.size( ) << endl;

    // 17: метод insert( ) - вставка элемента со значением 21 после
```

```

// первого элемента вектора v1

v1.insert( v1.begin( )+1, 21 );
cout << "\nВыполняем операцию " "v1.insert( v1.begin( )+1, 21 );"
    << "\nРезультат операции - размер v1: " << v1.size( ) <<
    ",\nv1[ 1 ] = " << v1[ 1 ] << ", v1[ 2 ] = " << v1[ 2 ] << endl;
// Вставка элемента со значением 25 после третьего элемента вектора
cout << "\nВыполняем операцию " "v1.insert( v1.begin( )+3, 25 );"
    << "\nv1.insert( v1.begin( )+3, 25 ) = " <<
    *v1.insert( v1.begin( )+3, 25 ) << endl;
cout << "Результат операции - размер v1: " << v1.size( ) <<
    ",\nv1[ 1 ] = " << v1[ 1 ] << ", v1[ 3 ] = " << v1[ 3 ] << endl;

// 18: метод insert( ) - вставка 3 элементов со значением 7 после
// четвертого элемента вектора

v1.insert( v1.begin( )+4, 3, 7 );
cout << "\nВыполняем операцию v1.insert( v1.begin( )+4, 3, 7 );"
    << "\nРезультат операции - размер v1: " << v1.size( ) <<
    ",\nv1[ 3 ] = " << v1[ 3 ] << ", v1[ 4 ] = " << v1[ 4 ] <<
    ", v1[ 6 ] = " << v1[ 6 ] << endl;

// 19: метод insert( ) - вставка третьего и четвертого элементов
// вектора v1 в начало вектора v2

vector< int >
    v2( 5, -1 );
cout << "\nРазмер v2: " << v2.size( ) << endl;
v2.insert( v2.begin( ), v1.begin( )+3, v1.begin( )+5 );
cout << "Выполняем операцию v2.insert( v2.begin( ), v1.begin( )+3, "
    "\n
        v1.begin( )+5 );"
    << "\nРезультат операции - размер v2: " << v2.size( ) <<
    ",\nv2[ 0 ] = " << v2[ 0 ] << ", v2[ 1 ] = " << v2[ 1 ] <<
    ", v2[ 2 ] = " << v2[ 2 ] << endl;
// Вставляем C-вектор vec3 в начало вектора v2
int    vec3[ 2 ] = { 11, 12 };
v2.insert( v2.begin( ), vec3, vec3+2 );
cout << "\nВыполняем операцию "
    "v2.insert( v2.begin( ), vec3, vec3+2 );"
    << "\nРезультат операции - размер v2: " <<
    v2.size( ) << ",\nv2[ 0 ] = " << v2[ 0 ] << ", v2[ 1 ] = "
    << v2[ 1 ] << ", v2[ 2 ] = " << v2[ 2 ] << endl;

// 20: метод erase( ) - удаление второго элемента вектора v2

v2.erase( v2.begin( )+1 );

```

```

cout << "\nВыполняем операцию v2.erase( v2.begin( )+1 );"
    << "\nРезультат операции - размер v2: " << v2.size( ) <<
    ", \nv2[ 0 ] = " << v2[ 0 ] << ", v2[ 1 ] = " << v2[ 1 ] <<
    ", v2[ 2 ] = " << v2[ 2 ] << endl;
// Удаляем первый и второй элементы вектора v2
v2.erase( v2.begin( ), v2.begin( )+2 );
cout << "\nВыполняем операцию "
    "v2.erase( v2.begin( ), v2.begin( )+2 );"
    << "\nРезультат операции - размер v2: " << v2.size( ) <<
    ", \nv2[ 0 ] = " << v2[ 0 ] << ", v2[ 1 ] = " << v2[ 1 ] <<
    ", v2[ 2 ] = " << v2[ 2 ] << endl;

// 22: метод clear( ) - очистка вектора v2

v2.clear( );
cout << "\nВыполняем операцию v2.clear( );"
    << "\nРезультат операции - размер v2: " << v2.size( ) << endl;

// 23: метод swap( ) - обмен векторов v3 и v4

vector< int >
    v3( 5, 2 ), v4( 4, -2 );
cout << "\nРазмер v3: " << v3.size( ) << ", v3[ 0 ] = " << v3[ 0 ]
    << endl;
cout << "Размер v4: " << v4.size( ) << ", v4[ 3 ] = " << v4[ 3 ]
    << endl;
v3.swap( v4 );           // Эquiv. v4.swap( v3 );
cout << "Выполняем операцию v3.swap( v4 );" << endl;
cout << "Размер v3: " << v3.size( ) << ", v3[ 0 ] = " << v3[ 0 ]
    << endl;
cout << "Размер v4: " << v4.size( ) << ", v4[ 4 ] = " << v4[ 4 ]
    << endl << endl;

return 0;
}

```

Листинг 15.11. Результаты выполнения программы

```

Размер v1: 5
Выполняем операцию v1.push_back( 22 );
Результат операции - размер v1: 6, v1[ 5 ] = 22

Выполняем операцию v1.pop_back( );
Результат операции - размер v1: 5

Выполняем операцию v1.insert( v1.begin( )+1, 21 );

```

Результат операции - размер v1: 6,
v1[1] = 21, v1[2] = 2

Выполняем операцию v1.insert(v1.begin()+3, 25);
v1.insert(v1.begin()+3, 25) = 25

Результат операции - размер v1: 7,
v1[1] = 21, v1[3] = 25

Выполняем операцию v1.insert(v1.begin()+4, 3, 7);
Результат операции - размер v1: 10,
v1[3] = 25, v1[4] = 7, v1[6] = 7

Размер v2: 5
Выполняем операцию v2.insert(v2.begin(), v1.begin()+3,
v1.begin()+5);

Результат операции - размер v2: 7,
v2[0] = 25, v2[1] = 7, v2[2] = -1

Выполняем операцию v2.insert(v2.begin(), vec3, vec3+2);
Результат операции - размер v2: 9,
v2[0] = 11, v2[1] = 12, v2[2] = 25

Выполняем операцию v2.erase(v2.begin()+1);
Результат операции - размер v2: 8,
v2[0] = 11, v2[1] = 25, v2[2] = 7

Выполняем операцию v2.erase(v2.begin(), v2.begin()+2);
Результат операции - размер v2: 6,
v2[0] = 7, v2[1] = -1, v2[2] = -1

Выполняем операцию v2.clear();
Результат операции - размер v2: 0

Размер v3: 5, v3[0] = 2
Размер v4: 4, v4[3] = -2
Выполняем операцию v3.swap(v4);
Размер v3: 4, v3[0] = -2
Размер v4: 5, v4[4] = 2

Press any key to continue

Операции отношений для векторов. В шаблоне классов `vector` для сравнения векторов предусмотрены следующие операции отношений: "=", "!=", "<", "<=", ">" и ">=".

Два вектора считаются равными, если равны их размеры и все соответствующие пары элементов. Один вектор меньше другого, если первый из элементов одного вектора, не равный соответствующему элементу другого, меньше его (используется

лексикографическое сравнение). Аналогичным образом определяются другие операции отношений над векторами.

ЗАМЕЧАНИЕ

Если сравниваемые векторы имеют разную длину, но одинаковые значения элементов, то более длинный вектор считается лексикографически большим.

Использование операций отношений для векторов иллюстрирует пример из листингов 15.12, 15.13.

Листинг 15.12. Файл vec6.cpp

```
/*
    Последовательный контейнер vector. Операции отношений над векторами.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <vector>             // Последовательный контейнер

using namespace std;        // Используем стандартное
                             // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    vector< int >
        v1( 5, 2 ), v2( 5, 2 ), v3( 6, 2 ), v4( 5, -2 ),
        v5( 6, -2 );

    cout << "\nv1: ";
    for( unsigned i=0; i<5; i++ ) cout << v1[ i ] << ' ';
    cout << "\nv2: ";
    for( i=0; i<5; i++ ) cout << v2[ i ] << ' ';
    cout << "\nv3: ";
    for( i=0; i<6; i++ ) cout << v3[ i ] << ' ';
    cout << "\nv4: ";
    for( i=0; i<5; i++ ) cout << v4[ i ] << ' ';
    cout << "\nv5: ";
    for( i=0; i<6; i++ ) cout << v5[ i ] << ' ';

    if( v1 == v2 )
        cout << "\n\nv1 равно v2" << endl;

    if( v1 == v3 )
        cout << "v1 равно v3" << endl;
```

```
else if( v1 < v3 )
    cout << "v1 меньше v3" << endl;

if( v1 != v3 )
    cout << "v1 не равно v3" << endl;

if( v1 <= v3 )
    cout << "v1 меньше или равно v3" << endl;

if( v3 >= v1 )
    cout << "v3 больше или равно v1" << endl;

if( v3 > v1 )
    cout << "v3 больше v1" << endl;

if( v5 > v4 )
    cout << "v5 больше v4" << endl << endl;

return 0;
}
```

Листинг 15.13. Результаты выполнения программы

```
v1: 2 2 2 2 2
v2: 2 2 2 2 2
v3: 2 2 2 2 2
v4: -2 -2 -2 -2 -2
v5: -2 -2 -2 -2 -2

v1 равно v2
v1 меньше v3
v1 не равно v3
v1 меньше или равно v3
v3 больше или равно v1
v3 больше v1
v5 больше v4
v5 больше v4

Press any key to continue
```

Для более эффективной работы с векторами в стандартной библиотеке определены шаблоны функций, названные *алгоритмами*. Они включают поиск значений, сортировку элементов и многое другое. Указанные алгоритмы будут рассмотрены далее.

15.2. Последовательные контейнеры.

Вектор логических значений (*vector<bool>*)

Для векторов, содержащих элементы логического типа, в стандартной библиотеке языка C++ определена специализация шаблона классов `vector<bool>`. Сделано это для минимизации затрат памяти — вектор логических значений в классе `vector<bool>` реализован таким образом, что каждый его элемент вместо 1 байта занимает 1 бит. Это в восемь раз экономит оперативную память. Конечно же, оптимизация оперативной памяти достигается не даром. В языке C++ минимальной адресуемой единицей памяти является 1 байт. Следовательно, в классе `vector<bool>` выполняется специальная обработка ссылок и итераторов.

Специализированная разновидность класса `vector<bool>` не удовлетворяет всем требованиям других векторов. Этот класс может уступать обычным реализациям векторов по скорости работы, так как операции с элементами приходится преобразовывать в операции с битами.

Имейте в виду, что класс `vector<bool>` представляет собой нечто большее, чем специализация класса `vector<>` для типа `bool` — в него включена поддержка специальных операций, упрощающих работу с флагами и наборами битов. Размер контейнера `vector<bool>` изменяется динамически, и поэтому его можно рассматривать как битовое поле с динамическим размером. Это означает, что при работе с контейнером можно добавлять и удалять биты. Вместе с тем, при работе с битовым полем, имеющим *статический* размер, вместо класса `vector<bool>` лучше использовать класс `bitset`, который будет рассмотрен далее.

Контейнер `vector<bool>` предусматривает следующие дополнительные операции:

```
c.flip( )
```

Инвертирует все логические элементы контейнера `c`.

```
c[ idx ].flip( )
```

Инвертирует логический элемент контейнера `c` с индексом `idx`.

```
c.front( ).flip( )
```

Инвертирует первый логический элемент контейнера `c`.

ПРИМЕЧАНИЕ

Метод `flip()`, предусмотренный стандартом, в версии языка Microsoft Visual C++ 6.0 не реализован.

Обычные операции с векторами, рассмотренные ранее, работают также для `vector<bool>` и сохраняют свой обычный смысл.

Рассмотрим иллюстрирующий пример (листинги 15.14, 15.15).

Листинг 15.14. Файл `vec_bool.cpp`

```
/*
Последовательный контейнер vector. Работа с векторами логических значений.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
```

```

#include <iostream>          // Потокoвый ввод-вывод
#include <vector>             // Последовательный контейнер

using namespace             // Используем стандартное
    std;                    // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Создание вектора логических значений из 10 элементов, по умолчанию
    // инициализированных 0
    vector< bool >
        v1( 10 );

    cout << "v1: ";
    // Обращение к элементам с использованием итератора - вывод на экран
    // значений элементов булевского вектора
    for( vector< bool >::iterator bi=v1.begin( ); bi != v1.end( ); bi++ )
        cout << *bi << " ";

    cout << "\n\nInput 10 value (0 or 1), separator ' ' " for v1:\n";
    // Обращение к элементам по индексу - ввод значений элементов
    // булевского вектора с клавиатуры
    for( unsigned i=0; i<v1.size( ); i++ )
    {
        cin >> v1[ i ];
        if( !cin )
        {
            cout << "Error input" << endl;
            exit( 1 );
        }
    }
    // Вывод на экран значений элементов булевского вектора
    cout << "\nAfter input v1: ";
    for( i=0; i<v1.size( ); i++ )
        cout << v1[ i ] << " ";
    cout << endl << endl;

    return 0;
}

```

Листинг 15.15. Результаты выполнения программы

```
v1: 0 0 0 0 0 0 0 0 0 0
```

```
Input 10 value (0 or 1), separator ' ' for v1:
```



```

// Конструкторы
explicit deque( const A &a1 = A( ) );
explicit deque( size_type n, const T &v = T( ),
               const A &a1 = A( ) );
deque( const deque < T, A > &x );
deque( const_iterator first, const_iterator last,
      const A &a1 = A( ) );

// Деструктор
~deque( );

// Интерфейсные методы
// *****

deque < T, A > &      operator=( const deque < T, A > &x );

// Итераторы
iterator             begin( );
const_iterator       begin( ) const;
iterator             end( );
const_iterator       end( ) const;
reverse_iterator     rbegin( );
const_reverse_iterator rbegin( ) const;
reverse_iterator     rend( );
const_reverse_iterator rend( ) const;

void                 resize( size_type n, T x = T( ) );
size_type            size( ) const;
size_type            max_size( ) const;
bool                 empty( ) const;
allocator_type       get_allocator( ) const;
const_reference       at( size_type pos ) const;
reference             at( size_type pos );
const_reference       operator[]( size_type pos ) const;
reference             operator[]( size_type pos );
reference             front( );
const_reference       front( ) const;
reference             back( );
const_reference       back( ) const;
void                 push_front( const T &x );
void                 pop_front( );
void                 push_back( const T &x );
void                 pop_back( );

```

```

void          assign( const_iterator first,
                     const_iterator last );
void          assign( size_type n, const T &x );
iterator      insert( iterator pos, const T &x );
void          insert( iterator pos, size_type m,
                     const T &x );
void          insert( iterator pos,
                     const_iterator first, const_iterator last );
iterator      erase( iterator pos );
iterator      erase( iterator first, iterator last );
void          clear( );
void          swap( deque < T, A > &x );

```

protected:

```

    A          allocator;
};

```

Обращаем внимание читателя на то, что разновидности конструкторов для создания двусторонних очередей, возможности конструкторов и способы создания очередей полностью аналогичны рассмотренным ранее конструкторам для создания векторов.

В последовательном контейнере `deque` определены операция присваивания, методы копирования, итераторы, операции отношений и методы доступа к элементам и изменения объектов, аналогичные соответствующим операциям и методам для работы с векторами.

Вставка и удаление элементов внутрь очереди, как и для векторов, выполняются за время, пропорциональное размеру очереди. Если эти операции выполняются над внутренними элементами, то все значения итераторов и ссылок на элементы очереди становятся недействительными. После операции добавления элемента в любой из концов очереди все значения итераторов становятся недействительными, а значения ссылок на элементы очереди сохраняются. После операции выборки из любого конца очереди недействительными становятся только значения итераторов и ссылок, связанных с этими элементами.

Последовательный контейнер `deque` отличается от контейнера `vector` следующим:

- ☐ дополнительно определены методы добавления и выборки из начала очереди: `push_front()` и `pop_front()` соответственно;
- ☐ отсутствуют методы `capacity()` и `reserve()`.

Чтобы убедиться, что работа с двусторонней очередью полностью аналогична работе с вектором, внимательно рассмотрите пример из листингов 15.17, 15.18.

Листинг 15.17. Файл `deque.cpp`

```
/*
```

Последовательный контейнер `deque`. Работа с двусторонней очередью с помощью методов контейнера `deque`.

```
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <deque>             // Последовательный контейнер

using namespace            // Используем стандартное
    std;                   // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    // Создается двусторонняя очередь из 10 элементов целого типа
    // с единичными значениями
    deque< int >
        v1( 10, 1 );

    // Печать состояния очереди
    cout << "Размер очереди v1: " << v1.size( )
        << "\nСостояние очереди v1: ";
    for( unsigned i=0; i<v1.size( ); i++ )
        cout << v1[ i ] << ' ';

    // Добавляем 2 элемента в начало и 1 элемент в конец очереди
    v1.push_front( 7 );
    v1.push_front( 12 );
    v1.push_back( 21 );
    // Добавляем один элемент после второго элемента с начала
    v1.insert( v1.begin( )+2, 33 );

    // Печать состояния очереди с использованием итератора
    cout << "\n\nРазмер очереди v1: " << v1.size( )
        << "\nСостояние очереди после добавления элементов 7"
        " и 12 в начало,\n21 в конец очереди и 33 в начало после"
        " второго элемента \nv1: ";
    for( deque< int >::iterator it = v1.begin( ); it != v1.end( ); it++ )
        cout << *it << ' ';

    // Удаляем из очереди все добавленные в нее элементы
    v1.erase( v1.begin( )+2 );
    v1.pop_front( );
    v1.pop_front( );
    v1.pop_back( );

    // Печать состояния очереди
    cout << "\n\nРазмер очереди v1: " << v1.size( )
        << "\nСостояние очереди после удаления добавленных "
```

```

    "элементов \nv1: ";
    for( i=0; i<v1.size( ); i++ )
        cout << v1.at( i ) << ' ';

    cout << endl << endl;

    return 0;
}

```

Листинг 15.18. Результаты выполнения программы

```

Размер очереди v1: 10
Состояние очереди v1: 1 1 1 1 1 1 1 1 1 1

Размер очереди v1: 14
Состояние очереди после добавления элементов 7 и 12 в начало,
21 в конец очереди и 33 в начало после второго элемента
v1: 12 7 33 1 1 1 1 1 1 1 1 1 1 21

Размер очереди v1: 10
Состояние очереди после удаления добавленных элементов
v1: 1 1 1 1 1 1 1 1 1 1

Press any key to continue

```

Возможности деков (двусторонних очередей) и области их применения. По своим возможностям деки имеют следующие отличия от векторов:

- ☐ вставка и удаление элементов выполняются с постоянным временем и быстро как в начале, так и в конце дека (для вектора эти операции выполняются быстро только в конце вектора);
- ☐ внутренняя структура дека содержит дополнительный уровень ссылок. За счет этого обращение к элементам и перемещение итератора в деках выполняются медленнее;
- ☐ деки не позволяют управлять емкостью и моментом перераспределения памяти. Поэтому при любых операциях вставки и удаления, выполняемых не в начале дека, становятся недействительными все указатели, ссылки и итераторы, ссылающиеся на элементы дека;
- ☐ перераспределение памяти в деках выполняется несколько быстрее, чем в векторах. Причиной этого являются особенности внутренней структуры дека, из-за которых удается копировать не все элементы.

На основании сказанного и сравнительного анализа деков и векторов можно сделать следующие рекомендации. Используйте дек, а не вектор, если выполняются следующие условия:

- ☐ вставка/удаление элементов выполняются с обоих концов очереди (классический пример двусторонней очереди, поэтому термины "дек" и "двусторонняя очередь" часто отождествляются);

- ❑ в программе не используются ссылки на элементы контейнера;
- ❑ важно, чтобы контейнер освобождал неиспользуемую память (хотя стандарт таких гарантий не дает).

К очередям можно применять алгоритмы стандартной библиотеки, которые мы рассмотрим далее.

15.4. Последовательные контейнеры. Список (*list*)

Отличительные признаки и возможности списков. По своей внутренней структуре списки *полностью отличаются* от векторов и деков. Перечислим наиболее важные особенности списков.

- ❑ Список *не предоставляет* произвольного доступа к своим элементам. Например, чтобы обратиться к четвертому элементу с начала списка, необходимо перебрать первые три элемента по цепочке ссылок. Поэтому обращение к произвольному элементу списка выполняется относительно медленно. Так как списки не поддерживают произвольный доступ к элементам, то в них не определены ни оператор индексирования "`[]`", ни метод `at()`.
- ❑ Вставка и удаление элементов в любое место списка выполняются *быстро* (за постоянное время), причем *не только* с его концов. Объясняется это тем, что указанные операции не требуют перемещения других элементов списка. Во внутренней реализации изменяются значения только нескольких указателей.
- ❑ После выполнения операций вставки и удаления элементов списка значения указателей, итераторов и ссылок, относящиеся к другим элементам, остаются действительными.
- ❑ Последовательный контейнер `list` реализован в STL в виде *двусвязного (двунаправленного) списка*, каждый узел (элемент) которого содержит ссылки на последующий и предыдущий элементы. Поэтому операции инкремента и декремента для итераторов списка выполняются за *постоянное* время, а продвижение на *n* элементов занимает время, пропорциональное *n*.
- ❑ Обработка удаления из списка реализована так, что практически *каждая* операция завершается успешно или не вносит изменений. Таким образом, список не может оказаться в промежуточном состоянии, в котором операция завершена только наполовину.
- ❑ Списки *не поддерживают* операции, связанные с емкостью и перераспределением памяти — такие операции просто не нужны. У каждого элемента списка имеется собственная память, которая остается действительной до удаления элемента.
- ❑ Списки поддерживают много *специальных методов* для перемещения элементов. Эти методы представляют собой оптимизированные версии одноименных универсальных алгоритмов. Оптимизация основана на замене указателей вместо копирования и перемещения значений.

Чтобы использовать список в программе, необходимо включить в нее заголовочный файл `<list>`:

```
#include <list>
```

Тип списка определяется как шаблонный класс в пространстве имен `std`:

```
namespace std
{
    template < class T, class A = allocator < T > >
    class list;
}
```

Элементы списка относятся к произвольному типу `T`, поддерживающему присваивание и копирование. Необязательный второй параметр определяет модель памяти. По умолчанию используется модель `allocator`, определенная в стандартной библиотеке языка C++.

Определение шаблона классов `list` представлено в листинге 15.19 (как и в предыдущих случаях, дана упрощенная запись).

Листинг 15.19. Шаблон классов `list`

```
// TEMPLATE CLASS list
template < class T, class A = allocator < T > >
class list
{
    public:

        // Определение типов
        typedef typename A::reference      reference;
        typedef typename A::const_reference const_reference;
        typedef implementation defined     iterator;
        typedef implementation defined     const_iterator;
        typedef A::size_type               size_type;
        typedef A::difference_type          difference_type;
        typedef T                           value_type;
        typedef A                           allocator_type;
        typedef typename A::pointer         pointer;
        typedef typename A::const_pointer   const_pointer;
        typedef std::reverse_iterator<iterator>
                                           reverse_iterator;
        typedef std::reverse_iterator< const_iterator>
                                           const_reverse_iterator;

        // Конструкторы
        explicit list( const A &al = A( ) );
        explicit list( size_type n, const T &v = T( ),
                       const A &al = A( ) );
        list( const list < T, A > &x );
        template < class InputIterator >
        list( InputIterator first, InputIterator last,
              const A &al = A( ) );
```

```

list( const_iterator first, const_iterator last,
      const A &a1 = A( ) );

// Деструктор
~list( );

// Интерфейсные методы
// *****

// Присваивание
list < T, A > &      operator=( const list < T, A > &x );

// Копирование
template < class InputIterator >
void              assign( InputIterator first,
                          InputIterator last);
void              assign( size_type n, const T &x );

allocator_type    get_allocator( ) const;

// Итераторы
iterator          begin( );
const_iterator    begin( ) const;
iterator          end( );
const_iterator    end( ) const;
reverse_iterator  rbegin( );
const_reverse_iterator rbegin( ) const;
reverse_iterator  rend( );
const_reverse_iterator rend( ) const;

// Получение сведений о списке
bool              empty( ) const;
size_type         size( ) const;
size_type         max_size( ) const;

// Изменение размера списка
void              resize( size_type n, T x = T( ) );

// Доступ к элементам списка
reference         front( );
const_reference   front( ) const;
reference         back( );
const_reference   back( ) const;

// Занесение/извлечение в начало/конец списка

```

```

void                push_front( const T &x );
void                pop_front( );
void                push_back( const T &x );
void                pop_back( );

// Изменение списка - занесение/извлечение в середину списка
iterator            insert( iterator pos, const T &x );
void                insert( iterator pos, size_type m,
                           const T &x );

template < class InputIterator >
void                insert( iterator pos,
                           InputIterator first, InputIterator last );
iterator            erase( iterator pos );
iterator            erase( iterator first, iterator last );

// Обмен
void                swap( list < T, A > &x );

// Очистка
void                clear( );

// Сцепка списков
void                splice( iterator pos, list < T, A > &x );
void                splice( iterator pos, list < T, A > &x,
                           iterator first );
void                splice( iterator pos, list < T, A > &x,
                           iterator first, iterator last );

// Удаление элемента по значению
void                remove( const T &v );
void                remove_if(
    binder2nd < not_equal_to < T > > ptr );

// Замена серии одинаковых элементов первым элементом серии
void                unique( );
void                unique(not_equal_to < T > ptr );

// Слияние списков
void                merge( list < T, A > &x );
void                merge( list < T, A > &x,
                           greater < T > ptr );

// Сортировка
void                sort( );

```

```

void                sort( greater < T > ptr );

// Изменение порядка следования элементов списка на обратный
void                reverse( );

protected:

    A                allocator;

};

```

Последовательный контейнер `list` поддерживает конструкторы, операцию присваивания, методы копирования, операции отношений, итераторы и методы изменения объектов (`insert()`, `erase()`, `swap()`, `clear()`), аналогичные имеющимся в последовательном контейнере `vector`.

Доступ к элементам списков ограничивается только методами `front()` и `back()`.

Для **занесения или извлечения элемента в начало или конец списка** определены методы `push_front()`, `pop_front()`, `push_back()` и `pop_back()`, аналогичные соответствующим методам очереди.

Для списка, как и для очереди, *не определен* метод `capacity()`. Объясняется это тем, что память под элементы списка выделяется по мере необходимости.

Для **изменения размера списка** можно использовать метод `resize()`. В результате его вызова произойдет удаление или добавление элементов в *конец* списка.

Ну, а теперь о главном. Для списков определено несколько *специфических методов*.

Сцепка списков (методы `splice()`) служит для перемещения элементов из одного списка в другой *без перераспределения* памяти, только за счет изменения указателей. При этом оба списка должны содержать элементы одинакового типа. Имеются следующие формы методов для сцепки списков:

```

void splice( iterator pos, list < T, A > &x );                // 1
void splice( iterator pos, list < T, A > &x, iterator posl ); // 2
void splice( iterator pos, list < T, A > &x, iterator first,
            iterator last );                                // 3

```

Первая форма метода вставляет в вызывающий список перед элементом, позиция которого указана первым аргументом в вызове метода, все элементы списка, указанного вторым аргументом. При этом вставляемый список становится *пустым*. Из сказанного следует, что *нельзя* вставлять список сам в себя.

Вторая форма метода переносит элемент, позицию которого указывает третий аргумент в вызове, из списка, заданного вторым аргументом. Место вставки в вызывающий список определяется первым аргументом. Допускается *переносить* элемент в пределах одного списка.

Третья форма метода аналогичным образом переносит из списка в список несколько элементов. Их диапазон задается третьим и четвертым аргументами. Если для одного и того же списка первый аргумент находится в диапазоне между третьим и четвертым аргументами, то результат операции *не определен*.

Удаление элемента списка по его значению (метод `remove()`) служит для удаления *всех* элементов списка, значения которых совпадают со значением аргумента `v` в вызове метода `remove()`:

```
void remove( const T &v );
```

Упорядочение элементов списка обеспечивает метод `sort()`:

```
void sort( );
```

При использовании этой версии метода список сортируется по возрастанию элементов. Порядок следования элементов, имеющих одинаковые значения, сохраняется. Время сортировки пропорционально $N * \log_2 N$, где N — количество элементов в списке.

Изменение порядка следования элементов списка на обратный обеспечивает метод `reverse()`. Время работы этого метода пропорционально количеству элементов в списке.

Версия метода

```
void unique( );
```

оставляет в списке только первый элемент из каждой серии идущих подряд одинаковых элементов.

Для **слияния списков** можно использовать метод

```
void merge( list < T, A > &x );
```

При этом оба списка должны быть *упорядочены по возрастанию*. В результате получится упорядоченный список. Если элементы в вызывающем списке и списке-аргументе совпадают, то первыми будут располагаться элементы из вызывающего списка.

Для практического освоения перечисленных методов рассмотрим иллюстрирующий пример (листинги 15.20, 15.21).

Листинг 15.20. Файл `list.cpp`

```
/*
   Последовательный контейнер list. Работа со списками с помощью методов контейнера
   list.
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <list>               // Последовательный контейнер

using namespace            // Используем стандартное
    std;                   // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Создание трех пустых списков
    list< int >
```

```
L1, L2, L3;

// Заполнение списков L1 и L2 значениями
for( unsigned i=0; i<5; i++ )
    L1.push_front( i );
for( i=0; i<3; i++ )
    L2.push_back( i+10 );

// Присваивание
L3 = L2;

// Печать состояния списков с использованием итераторов
list< int >::iterator
    it;
cout << "Размер списка L1: " << L1.size( ) <<
    "\nСостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';
cout << "\nРазмер списка L2: " << L2.size( ) <<
    "\nСостояние списка L2: ";
for( it = L2.begin( ); it != L2.end( ); it++ )
    cout << *it << ' ';
cout << "\nРазмер списка L3: " << L3.size( ) <<
    "\nСостояние списка L3: ";
for( it = L3.begin( ); it != L3.end( ); it++ )
    cout << *it << ' ';

// Вставка списка L2 перед третьим элементом списка L1
it = L1.begin( ); it++; it++;
// !!! Для итератора списка операция + не определена и записать
// it = L1.begin( )+2; нельзя (операции же ++ и -- определены)
L1.splice( it, L2 );
cout << "\n\nВставка списка L2 перед третьим элементом списка L1"
    << endl;
cout << "Размер списка L1: " << L1.size( ) <<
    "\nСостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';
cout << "\nРазмер списка L2: " << L2.size( ) <<
    "\nСостояние списка L2: ";
for( it = L2.begin( ); it != L2.end( ); it++ )
    cout << *it << ' ';

// Перемещение последнего элемента списка L1 в его начало
L1.splice( L1.begin( ), L1, --L1.end( ) );
cout << "\n\nПеремещение последнего элемента списка L1 в его начало"
```

```
<< endl;
cout << "Размер списка L1: " << L1.size( ) <<
      "\nСостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';

// Вставка 2, 3 элементов списка L3 в начало списка L1
L1.splice( L1.begin( ), L3, ++L3.begin( ),
          L3.end( ) );
cout << "\n\nВставка 2, 3 элементов списка L3 в начало списка L1"
      << endl;
cout << "Размер списка L1: " << L1.size( ) <<
      "\nСостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';
cout << "\n\nРазмер списка L3: " << L3.size( ) <<
      "\nСостояние списка L3: ";
for( it = L3.begin( ); it != L3.end( ); it++ )
    cout << *it << ' ';

// Удаление из списка L1 элементов со значением 12
L1.remove( 12 );
cout << "\n\nУдаление из списка L1 элементов со значением 12"
      << endl;
cout << "Размер списка L1: " << L1.size( ) <<
      "\nСостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';

// Сортировка элементов списка L1 по возрастанию
L1.sort( );
cout << "\n\nСортировка элементов списка L1 по возрастанию" << endl;
cout << "Размер списка L1: " << L1.size( ) <<
      "\nСостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';

// Слияние списков L1 и L3
L3.push_back( 11 );
cout << "\n\nРазмер списка L3: " << L3.size( ) <<
      "\nСостояние списка L3: ";
for( it = L3.begin( ); it != L3.end( ); it++ )
    cout << *it << ' ';
L1.merge( L3 );
cout << "\n\nСлияние списков L1 и L3" << endl;
cout << "Размер списка L1: " << L1.size( ) <<
```

```

        "\nCостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';

// Изменение порядка следования элементов списка L1 на
// противоположный
L1.reverse( );
cout << "\n\nИзменение порядка следования элементов "
    "\nCостояние списка L1 на противоположный" << endl;
cout << "Размер списка L1: " << L1.size( ) <<
    "\nCостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';

// Усечение серий элементов списка L1
L1.unique( );
cout << "\n\nУсечение серий элементов списка L1" << endl;
cout << "Размер списка L1: " << L1.size( ) <<
    "\nCостояние списка L1: ";
for( it = L1.begin( ); it != L1.end( ); it++ )
    cout << *it << ' ';

cout << endl << endl;

return 0;
}

```

Листинг 15.21. Результаты выполнения программы

```

Размер списка L1: 5
Состояние списка L1: 4 3 2 1 0
Размер списка L2: 3
Состояние списка L2: 10 11 12
Размер списка L3: 3
Состояние списка L3: 10 11 12

```

```

Вставка списка L2 перед третьим элементом списка L1
Размер списка L1: 8
Состояние списка L1: 4 3 10 11 12 2 1 0
Размер списка L2: 0
Состояние списка L2:

```

```

Перемещение последнего элемента списка L1 в его начало
Размер списка L1: 8
Состояние списка L1: 0 4 3 10 11 12 2 1

```

```

Вставка 2, 3 элементов списка L3 в начало списка L1

```

```
Размер списка L1: 10
Состояние списка L1: 11 12 0 4 3 10 11 12 2 1
Размер списка L3: 1
Состояние списка L3: 10
```

```
Удаление из списка L1 элементов со значением 12
Размер списка L1: 8
Состояние списка L1: 11 0 4 3 10 11 2 1
```

```
Сортировка элементов списка L1 по возрастанию
Размер списка L1: 8
Состояние списка L1: 0 1 2 3 4 10 11 11
```

```
Размер списка L3: 2
Состояние списка L3: 10 11
Слияние списков L1 и L3
Размер списка L1: 10
Состояние списка L1: 0 1 2 3 4 10 10 11 11 11
```

```
Изменение порядка следования элементов
списка L1 на противоположный
Размер списка L1: 10
Состояние списка L1: 11 11 11 10 10 4 3 2 1 0
```

```
Усечение серий элементов списка L1
Размер списка L1: 7
Состояние списка L1: 11 10 4 3 2 1 0
```

Press any key to continue

15.5. Адаптеры последовательных контейнеров. Стек (*stack*)

Помимо основных последовательных контейнеров стандартная библиотека языка C++ содержит специальные контейнерные *адаптеры*, предназначенные для особых целей.

- ☐ *Стеки* — контейнеры, элементы которых обрабатываются с использованием дисциплины обслуживания LIFO (последним занесен, первым извлечен).
- ☐ *Очереди* — контейнеры, элементы которых обрабатываются с использованием дисциплины обслуживания FIFO (первым занесен, первым извлечен). Иначе говоря, очередь представляет собой обычный буфер.
- ☐ *Приоритетные очереди* — контейнеры, каждому элементу которых соответствует приоритет, определяющий порядок выборки из очереди. По умолчанию он определяется с помощью операции `<` (`less`). Таким образом, при использовании умолчания из очереди каждый раз выбирается максимальный элемент.

Исторически адаптеры последовательных контейнеров считаются частью STL. Но с точки зрения программиста, это всего лишь специализированные контейнеры, которые используют общую архитектуру контейнеров, итераторов и алгоритмов, предоставленную STL.

В [3] отмечалось, что **стек** — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняется с одного конца, называемого *вершиной стека*. Другие операции со стеком не определены. В общем случае, стек можно реализовать на основе любого из рассмотренных последовательных контейнеров — вектора, двусторонней очереди или списка. Таким образом, стек является не новым типом контейнера, а вариантом имеющихся контейнеров (*адаптером* контейнеров).

Класс `stack<>` реализует стек, работающий по принципу "последним занесен, первым извлечен" (LIFO). Метод `push()` вставляет в стек произвольное количество элементов, а метод `pop()` удаляет элементы в порядке, обратном порядку их вставки.

Чтобы использовать стек в программе, необходимо включить в нее заголовочный файл `<stack>`:

```
#include <stack>
```

В файле `<stack>` шаблонный класс `<stack>` определяется в стандартном пространстве имен `std` следующим образом:

```
namespace std
{
    template < class T, class Container = deque < T > >
    class stack;
```

Первый параметр шаблона определяет тип элементов стека. Необязательный второй параметр определяет контейнер, который будет использован внутренней реализацией для хранения элементов стека. По умолчанию это дек, и объясняется это тем, что деки, в отличие от векторов, освобождают память при удалении элементов и обходятся без копирования всех элементов при перераспределении памяти.

Основной интерфейс стеков представлен следующими методами:

- ❑ `push()` — для вставки элемента в стек;
- ❑ `top()` — для получения верхнего элемента стека;
- ❑ `pop()` — для удаления элемента из стека.

Определение шаблона классов `stack` представлено в листинге 15.22 (дана упрощенная запись).

Листинг 15.22. Шаблон классов `stack`

```
// TEMPLATE CLASS stack
template < class T, class Container = deque < T > >
class stack
{
    public:
```

```

// Определения типов
// Тип элементов стека
typedef typename Container::value_type
                                value_type;

// Тип для значений размеров стека
typedef typename Container::size_type
                                size_type;

// Тип контейнера
typedef Container                container_type;

// Конструктор
explicit                        stack(
    const Container & = Container( ) );

// Интерфейсные методы
// *****

bool                            empty( ) const
{
    return c.empty( );
}

size_type                      size( ) const
{
    return c.size( );
}

value_type &                   top( )
{
    return c.back( );
}

const value_type &             top( ) const
{
    return c.back( );
}

void                           push( const value_type &x )
{
    c.push_back( x );
}

void                           pop( )
{
    c.pop_back( );
}

```

protected:

Container

```

        C;

};

// Следующие далее операции отношений также определены в стандартном
// пространстве имен std
template < class T, class Container >
    bool operator==( const stack < T, Container > &x,
                     const stack < T, Container > &y );
template < class T, class Container >
    bool operator!=( const stack < T, Container > &x,
                     const stack < T, Container > &y );
template < class T, class Container >
    bool operator<( const stack < T, Container > &x,
                   const stack < T, Container > &y );
template < class T, class Container >
    bool operator>( const stack < T, Container > &x,
                   const stack < T, Container > &y );
template < class T, class Container >
    bool operator>=( const stack < T, Container > &x,
                    const stack < T, Container > &y );
template < class T, class Container >
    bool operator<=( const stack < T, Container > &x,
                    const stack < T, Container > &y );

```

Кратко прокомментируем приведенное объявление шаблонного класса.

Конструкторы. С помощью конструктора умолчания (при создании объекта-стека после его идентификатора список аргументов отсутствует — см. иллюстрирующий пример далее) создается пустой стек. С помощью обычного конструктора можно создать стек, инициализированный элементами контейнера-аргумента, которые копируются в стек.

Проверка стека. Для проверки стека можно использовать метод `size()`, который возвращает текущее количество элементов в стеке, или метод `empty()`, возвращающий значение `true`, если стек пуст. Для проверки отсутствия элементов в стеке лучше использовать именно этот метод — он работает быстрее.

Занесение/извлечение элементов. Занесение элемента в стек выполняет метод `push()`, который вставляет в стек копию своего аргумента (его реализация показана в листинге 15.22). Метод `top()` возвращает значение элемента из вершины стека (этот элемент был помещен в стек последним). Перед вызовом данного метода необходимо убедиться в том, что стек содержит хотя бы один элемент, иначе его вызов может привести к непредсказуемым последствиям. Не константная форма этого метода позволяет модифицировать верхний элемент при нахождении его в стеке. Метод выборки из стека `pop()` удаляет из стека верхний элемент. Метод не имеет возвращаемого значения и, чтобы обработать значение верхнего элемента, нужно предварительно вызвать метод `top()`. Перед вызовом метода `top()` также необходимо

убедиться в том, что стек содержит хотя бы один элемент, иначе его вызов может привести к непредсказуемым последствиям.

При работе со стеком нельзя пользоваться итераторами и нельзя получить значение элемента из середины стека иначе, как выбрав из него все элементы, лежащие выше.

Операции отношений. Для стека, как и для всех рассмотренных ранее контейнеров, определены операции отношений ("==", "!=", ">", ">=", "<", "<="). Каждая из этих операций возвращает результат сравнения двух стеков одинакового типа. Два стека считаются равными, если они содержат одинаковое количество элементов, а также если элементы попарно совпадают и следуют в одинаковом порядке. Отношения "меньше/больше" для стеков проверяется по лексикографическому критерию.

Рассмотрим пример, иллюстрирующий работу со стеком (листинги 15.23—15.25).

Листинг 15.23. Файл stack.cpp

```
/*
    Адаптер stack. Работа со стеком, созданным на базе последовательного контейнера
    vector, с помощью методов адаптера stack.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <fstream>           // Для файловых объектов
#include <iostream>          // Поточковый ввод-вывод
#include <vector>             // Последовательный контейнер
// #include <list>           // Последовательный контейнер
// #include <deque>          // Последовательный контейнер
#include <stack>             // Последовательный контейнер

using namespace std;        // Используем стандартное
                             // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Создание трех пустых стеков с элементами целого типа
    stack< int, vector< int > >
        s1, s2, s3;
    // А можно было и так:
    // stack< int, list< int > > s1, s2, s3;
    // stack< int, deque< int > > s1, s2, s3;
    // или эквивалентно
    // stack< int > s1, s2, s3;

    // Заполнение стека s1 значениями из файла stack.dat и инициализация
    // стеков s2, s3 последовательными значениями 0, 1, 2, ... и 1, 2,
    // 3, ... соответственно
    ifstream in( "stack.dat" );
```

```

int      x, i = 0;
while( in >> x, !in.eof( ) )
{
    s1.push( x ); s2.push( i++ ); s3.push( i );
}

// Вывод на экран содержимого стека s1 - в результате стек s1 станет
// пустым
cout << "s1: ";
while( !s1.empty( ) )
{
    x = s1.top( ); cout << x << " "; s1.pop( );
}

// Сравнение стеков s2 и s3
if( s2<s3 )
    cout << "\n\ns2<s3";
// Вывод на экран содержимого стеков s2 и s3
cout << "\ns2: ";
while( !s2.empty( ) )
{
    x = s2.top( ); cout << x << " "; s2.pop( );
}
cout << "\ns3: ";
while( !s3.empty( ) )
{
    x = s3.top( ); cout << x << " "; s3.pop( );
}

cout << endl << endl;

return 0;
}

```

Листинг 15.24. Содержимое файла stack.dat

```
10 20 30 40
```

Листинг 15.25. Результаты работы программы

```
s1: 40 30 20 10
```

```
s2<s3
```

```
s2: 3 2 1 0
```

```
s3: 4 3 2 1
```

```
Press any key to continue
```

15.6. Адаптеры последовательных контейнеров. Очередь FIFO (*queue*)

Как указывалось в [3], *очередь* также представляет собой частный случай одностороннего списка, добавление элементов в который выполняется в конец, а выборка — из начала. Очередь является *адаптером*, который можно реализовать на основе двусторонней очереди или списка. Обратите внимание, что на базе вектора очередь организовать *нельзя*, поскольку для вектора не определена операция выборки из начала.

Шаблонный класс `queue<>` реализует очередь, работающую по правилу "первым занесен, первым извлечен" (FIFO). Метод `push()` заносит элементы в конец очереди, а метод `pop()` удаляет элементы в порядке их вставки. Таким образом, очередь может рассматриваться как классический буфер данных.

Объявление шаблона классов `queue` выполнено по умолчанию на базе двусторонней очереди (дека) в стандартном пространстве имен `std`, приведено в стандартном заголовочном файле `<queue>` и представлено в листинге 15.26 (дана упрощенная запись).

Листинг 15.26. Шаблон классов `queue`

```
namespace std
{
    // TEMPLATE CLASS queue: первый параметр шаблона определяет тип
    // элементов, а необязательный второй параметр определяет
    // контейнер, который будет использоваться внутренней реализацией
    // для хранения элементов. По умолчанию — это дек. Вместо дека
    // можно использовать список, но вектор использовать нельзя
    template < class T, class Container = deque< T > >
    class queue
    {
    public:

        // Определения типов
        // Тип элементов очереди
        typedef typename Container::value_type
                                value_type;

        // Тип для значений размеров очереди
        typedef typename Container::size_type
                                size_type;

        // Тип контейнера
        typedef Container        container_type;

        // Конструктор
        explicit                  queue(const Container & = Container( ) );

        // Интерфейсные методы
```

```

// *****

bool                empty( ) const
{
    return c.empty( );
}
size_type           size( ) const
{
    return c.size( );
}
// Получение значения из начала
value_type &        front( )
{
    return c.front( );
}
const value_type &   front( ) const
{
    return c.front( );
}
// Получение значения из конца
value_type &         back( )
{
    return c.back( );
}
const value_type &   back( ) const
{
    return c.back( );
}
// Занесение элемента в конец
void                push( const value_type &x )
{
    c.push_back( x );
}
// Извлечение элемента из начала
void                pop( )
{
    c.pop_front( );
}

protected:

    Container
        c;

};

```

// Операции отношений

```

template < class T, class Container >
    bool operator==( const queue < T, Container > &x,
                     const queue < T, Container > &y );

template < class T, class Container >
    bool operator!=( const queue < T, Container > &x,
                     const queue < T, Container > &y );

template < class T, class Container >
    bool operator<( const queue < T, Container > &x,
                    const queue < T, Container > &y );

template < class T, class Container >
    bool operator>( const queue < T, Container > &x,
                    const queue < T, Container > &y );

template < class T, class Container >
    bool operator>=( const queue < T, Container > &x,
                     const queue < T, Container > &y );

template < class T, class Container >
    bool operator<=( const queue < T, Container > &x,
                     const queue < T, Container > &y );
}

```

Приведем более подробные описания членов класса `queue<>`.

Конструкторы. С помощью конструктора умолчания (при создании объекта-очереди после его идентификатора список аргументов отсутствует — см. иллюстрирующий пример далее) создается пустая очередь. С помощью обычного конструктора можно создать очередь, инициализированную элементами контейнера-аргумента, которые копируются в очередь.

Проверка очереди. Для проверки очереди можно использовать метод `size()`, который возвращает текущее количество элементов в очереди, или метод `empty()`, возвращающий значение `true`, если очередь пуста. Для проверки отсутствия элементов в стеке лучше использовать именно этот метод — он работает быстрее.

Занесение/извлечение элементов. Занесение элемента в конец очереди выполняет метод `push()`, который вставляет в очередь копию своего аргумента (его реализация показана в листинге 15.26). Метод `front()` возвращает значение элемента из начала очереди (этот элемент был помещен в очередь первым). Перед вызовом данного метода необходимо убедиться в том, что очередь содержит хотя бы один элемент, иначе его вызов может привести к непредсказуемым последствиям. Неконстантная форма метода `front()` позволяет модифицировать значение элемента из начала очереди. Метод `back()` возвращает значение элемента из конца очереди (этот элемент был помещен в очередь последним). Перед вызовом данного метода необходимо убедиться в том, что очередь содержит хотя бы один элемент, иначе его вызов может привести к непредсказуемым последствиям. Неконстантная форма метода `back()` позволяет модифицировать значение элемента из конца очереди. Метод выборки из очереди `pop()` удаляет элемент из начала очереди. Метод не имеет возвращаемого значения и, чтобы обработать значение элемента из начала очереди, нужно предварительно вызвать метод `front()`. Перед вызовом метода `pop()` также необхо-

димо убедиться в том, что очередь содержит хотя бы один элемент, иначе его вызов может привести к непредсказуемым последствиям.

Операции отношений. Для очереди, как и для всех рассмотренных ранее контейнеров, определены операции отношений ("==", "!=", ">", ">=", "<", "<="). Каждая из этих операций возвращает результат сравнения двух стеков одинакового типа. Две очереди считаются равными, если они содержат одинаковое количество элементов и если элементы попарно совпадают и следуют в одинаковом порядке. Отношения "меньше/больше" для очередей проверяются по лексикографическому критерию.

Рассмотрим пример, иллюстрирующий работу с очередями (листинги 15.27—15.29).

Листинг 15.27. Файл queue.cpp

```
/*
    Адаптер queue. Работа с очередями, созданными на базе последовательного
    контейнера list, с помощью методов адаптера queue.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <fstream>           // Для файловых объектов
#include <iostream>          // Поточковый ввод-вывод
#include <list>               // Последовательный контейнер
// #include <deque>           // Последовательный контейнер
#include <queue>              // Последовательный контейнер

using namespace             // Используем стандартное
    std;                    // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Создание трех пустых очередей
    queue< int, list< int > >
        q1, q2, q3;
    // А можно было и так:
    // queue< int, deque< int > > q1, q2, q3;

    // Заполнение очереди q1 значениями из файла queue.dat и
    // инициализация очередей q2, q3 последовательными значениями 0, 1,
    // 2, ... и 1, 2, 3, ... соответственно
    ifstream in( "queue.dat" );
    int      x, i = 0;
    while( in >> x, !in.eof( ) )
    {
        q1.push( x ); q2.push( i++ ); q3.push( i );
    }

    // Вывод на экран содержимого очереди q1
    cout << "q1.front( ) = " << q1.front( ) << ", q1.back( ) = "
```

```

        << q1.back( );
    cout << "\nq1: ";
    while( !q1.empty( ) )
    {
        cout << q1.front( ) << " "; q1.pop( );
    }

    // Сравнение очередей q2, q3
    if( q2<q3 )
        cout << "\n\nq2<q3";
    // Вывод на экран содержимого очередей q2, q3
    cout << "\nq2: ";
    while( !q2.empty( ) )
    {
        cout << q2.front( ) << " "; q2.pop( );
    }
    cout << "\nq3: ";
    while( !q3.empty( ) )
    {
        cout << q3.front( ) << " "; q3.pop( );
    }

    cout << endl << endl;

    return 0;
}

```

Листинг 15.28. Содержимое файла queue.dat

```
10 20 30 40
```

Листинг 15.29. Результаты работы программы

```

q1.front( ) = 10, q1.back( ) = 40
q1: 10 20 30 40

q2<q3
q2: 0 1 2 3
q3: 1 2 3 4

Press any key to continue

```

К адаптерам стеков и очередей можно применять алгоритмы стандартной библиотеки, о чем будет сказано далее.

15.7. Адаптеры последовательных контейнеров.

Очередь с приоритетами (*priority_queue*)

В очереди с приоритетами каждому элементу соответствует приоритет, определяющий порядок выборки из очереди. По умолчанию он определяется с помощью операции `<` (`less`). Таким образом, при использовании умолчания из очереди каждый раз выбирается максимальный элемент. Если в очереди существует несколько элементов с "максимальными" приоритетами, то критерий выбора определяется реализацией.

Для реализации очереди с приоритетами подходят последовательные контейнеры, допускающие произвольный доступ к элементам. Такими контейнерами являются вектор или двусторонняя очередь.

Объявление шаблонного класса `priority_queue` выполнено в стандартном пространстве имен `std`, приведено в стандартном заголовочном файле `<queue>` и представлено в листинге 15.30 (дана упрощенная запись).

Листинг 15.30. Шаблонный класс `priority_queue`

```
namespace std
{
    // TEMPLATE CLASS priority_queue. Первый параметр шаблона определяет
    // тип элементов. Необязательный второй параметр определяет
    // контейнер, который будет использоваться внутренней реализацией
    // для хранения элементов. По умолчанию - это дек. Вместо дека
    // можно использовать список, но вектор использовать нельзя. Третий
    // параметр задает функцию или функциональный объект, с помощью
    // которых выполняется определение приоритета
    template < class T, class Container = vector< T >,
               class Compare = less< typename Container::value_type > >
    class priority_queue
    {
    public:

        // Определения типов
        // Тип элементов очереди
        typedef typename Container::value_type
                                   value_type;

        // Тип для значений размеров очереди
        typedef typename Container::size_type
                                   size_type;

        // Тип контейнера
        typedef Container           container_type;

        // Конструкторы
        explicit                    priority_queue(
            const Compare &x = Compare( ),
```

```

    const Container &y = Container( ) );
template < class InputIterator )
    priority_queue(InputIterator first,
    InputIterator last, const Compare &x = Compare( ),
    const Container &y = Container( ) );

// Интерфейсные методы
// *****

bool                empty( ) const
{
    return c.empty( );
}
size_type           size( ) const
{
    return c.size( );
}
// Получение значения элемента очереди в соответствии с его
// приоритетом
const value_type &  top( ) const
{
    return c.front( );
}
// Занесение элемента в очередь
void                push( const value_type &x );
// Извлечение элемента из очереди
void                pop( );

protected:

    Container
        c;
    Compare
        comp;

};

}

```

Из приведенного объявления видно, что для элементов с равными приоритетами очередь с приоритетами становится простой очередью. Как и для стеков, для очередей с приоритетами основными методами являются `push()`, `pop()` и `top()`. Метод `push()` вставляет элемент в приоритетную очередь, метод `top()` возвращает очередной элемент приоритетной очереди, а метод `pop()` удаляет элемент из приоритетной очереди. Как и в остальных контейнерных адаптерах, метод `pop()` удаляет следующий элемент, но не возвращает его, тогда как метод `top()` возвращает следующий

элемент без удаления. Поэтому для обработки и удаления следующего элемента очереди всегда приходится вызывать оба этих метода. Также, как и ранее, если очередь не содержит ни одного элемента, то поведение методов `pop()` и `top()` не определено. Наличие элементов в очереди проверяется методами `size()` и `empty()`.

Рассмотрим простой пример, иллюстрирующий использование этих методов (листинги 15.31, 15.32).

Листинг 15.31. Файл `p_queue.cpp`

```
/*
    Адаптер p_queue. Работа с очередями с приоритетами, созданными на базе
    последовательного контейнера vector, с помощью методов адаптера p_queue.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <vector>             // Последовательный контейнер
// #include <deque>          // Последовательный контейнер
#include <queue>              // Последовательный контейнер
#include <functional>         // Шаблон функциональных объектов

using namespace             // Используем стандартное
    std;                   // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Создание пустой очереди с приоритетами
    priority_queue< int, vector< int >, less< int > >
        pq1;

    // А можно было и так (это эквивалентно предыдущему - для
    // последовательного контейнера и операции сравнения использовано
    // умолчание): priority_queue< int > pq1;
    // Последовательный контейнер deque
    // priority_queue< int, deque< int >, less< int > > pq1;

    // Заполнение очереди с приоритетами pq1 значениями
    pq1.push( 33 ); pq1.push( 55 ); pq1.push( 100 );
    pq1.push( -12 );
    int      x;
    // Вывод на экран содержимого очереди с приоритетами pq1
    cout << "pq1: ";
    while( !pq1.empty( ) )
    {
        x = pq1.top( ); cout << x << ' '; pq1.pop( );
    }
}
```

```

    cout << endl << endl;

    return 0;
}

```

Листинг 15.32. Результаты работы программы

```
pq1: 100 55 33 -12
```

```
Press any key to continue
```

В этом примере при создании объекта `pq1` используется шаблон операции сравнения "на меньше" `less <тип>`, определенный в заголовочном файле `<functional>`. Вместо него можно с равным успехом использовать и другие шаблоны операций сравнения: "на больше" `greater <тип>`, "на больше равно" `greater_equal <тип>` и "на меньше равно" `less_equal <тип>`.

15.8. Ассоциативные контейнеры. Пары

Ассоциативные контейнеры обеспечивают быстрый доступ к данным за счет того, что они, как правило, построены на основе сбалансированных деревьев поиска. При этом уместно заметить, что стандартом языка определяется только интерфейс ассоциативных контейнеров, но не их реализация.

Существуют следующие типы ассоциативных контейнеров:

- ☐ словари (`map`);
- ☐ словари с дубликатами (`multimap`);
- ☐ множества (`set`);
- ☐ множества с дубликатами (`multiset`);
- ☐ битовые множества (`bitset`).

Словарь построен на основе *пар* "ключ/значение". У каждого элемента словаря имеется ключ, определяющий порядок сортировки элементов, и значение. Иными словами, можно сказать, что ключ *ассоциирован* с элементом. Отсюда и произошло название — *ассоциативный контейнер*. Примером ассоциативного контейнера можно считать англо-русский словарь, в котором ключом является английское слово, а элементом — русское. Массив тоже можно рассматривать как словарь, ключом в котором служит индекс элемента. В словарях, описанных в STL языка C++, в качестве ключа может использоваться значение *произвольного* типа. Каждый ключ должен присутствовать в словаре только в одном экземпляре — дубликаты не разрешаются.

Словарь с дубликатами представляет собой словарь с возможностью *дублирования* ключей.

Множество — это коллекция (набор элементов), в которой элементы *сортируются* в соответствии с их значениями. Каждый элемент присутствует в коллекции только

в одном экземпляре — *дубликаты не разрешаются*. Множество можно считать особой разновидностью словаря, в котором значение идентично ключу.

Множество с дубликатами представляет собой множество, в котором могут находиться элементы с одинаковыми значениями.

Битовые множества моделируют *массивы битов* (логических величин) *фиксированного размера*. В программах на языке С, а также в старых программах на языке С++ массив битов обычно представляется типом `long`, а работа с битами производится при помощи поразрядных операций "`&`", "`|`" и "`~`". Основным достоинством битовых множеств является произвольный размер битового поля (но фиксированный) и поддержка ряда дополнительных операций — присваивание значений отдельным битам, чтение и запись битовых полей как последовательности нулей и единиц и т. п. Если же вам потребуется контейнер с переменным количеством битов, то лучше использовать класс `vector<bool>` (см. разд. 15.2).

Ассоциативные контейнеры описаны в заголовочных файлах `<map>` и `<set>`. У всех шаблонных классов ассоциативных контейнеров имеется необязательный аргумент для передачи критерия сортировки. По умолчанию в качестве критерия сортировки используется оператор `<` (`less`).

Для работы с парами "ключ/элемент" в словарях и словарях с дубликатами используется шаблон классов `pair`, описанный в заголовочном файле `<utility>` в стандартном пространстве имен `std` (листинг 15.33).

Листинг 15.33. Шаблон классов `pair`

```
namespace std
{
    // TEMPLATE STRUCT pair
    template < class T1, class T2 >
    struct pair
    {
        // Имена типов компонентов
        typedef T1 first_type;
        typedef T2 second_type;

        // Компоненты
        T1    first;
        T2    second;

        // Конструктор умолчания - вызовы T1( ) и T2( ) обеспечивают
        // инициализацию компонентов
        pair( ) : first( T1( ) ), second( T2( ) )
        {
        }

        // Обычный конструктор
        pair( const T1 &x, const T2 &y )
```

```

        : first( x ), second( y )
    {
    }

    // Конструктор копирования с автоматическим преобразованием
    template < class U, class V >
    pair( const pair< U, V > &p )
        : first( p.first ), second( p.second )
    {
    }
};

// Операции отношений
template < class T1, class T2 >
    bool operator==( const pair< T1, T2 > &x,
                     const pair< T1, T2 > &y );
template < class T1, class T2 >
    bool operator!=( const pair< T1, T2 > &x,
                     const pair< T1, T2 > &y );
template < class T1, class T2 >
    bool operator<( const pair< T1, T2 > &x,
                    const pair< T1, T2 > &y );
template < class T1, class T2 >
    bool operator<=( const pair< T1, T2 > &x,
                     const pair< T1, T2 > &y );
template < class T1, class T2 >
    bool operator>( const pair< T1, T2 > &x,
                     const pair< T1, T2 > &y );
template < class T1, class T2 >
    bool operator>=( const pair< T1, T2 > &x,
                     const pair< T1, T2 > &y );

// Создание пары значений без явного указания типов
template < class T1, class T2 >
    pair< T1, T2 > make_pair( const T1 &x, const T2 &y );
}

```

Обратите внимание, что шаблонный класс объявляется со служебным словом **struct** и все его члены являются *открытыми*. Класс имеет два настроечных параметра — типы элементов пары — и два члена-данных шаблона. Первое из них имеет имя *first* и является ключом идентификации элемента, а второе — имя *second* и является значением элемента. Определено три конструктора: конструктор умолчания, обычный конструктор и конструктор копирования. Конструктор умолчания создает пару значений, инициализируемых конструкторами умолчания соответствующих типов. Согласно правилам языка C++, явный вызов конструктора умолчания также

инициализирует predefined типы данных. Поэтому следующее определение инициализирует компоненты `p` конструкторами `int()` и `float()`, что приводит к их обнулению:

```
std::pair< int, float > // p.first и p.second
    p;                // инициализируются нулями
```

Обычный конструктор получает два значения для инициализации пары "ключ/элемент". Шаблонная версия конструктора копирования используется в ситуациях, требующих автоматического преобразования типов.

Для пар определены операции отношений "=", "!=", ">", ">=", "<", "<=". Смысл операций отношений поясним на примере операции "<". Пара `p1` меньше пары `p2`, если `p1.first < p2.first` или `p1.first == p2.first && p1.second < p2.second`. Аналогичный смысл имеют другие операции отношений.

Шаблонный метод `make_pair()` позволяет создать пару значений без явного указания типов. В соответствии со сказанным, следующие строки эквивалентны:

```
std::make_pair( 35, 1.5 )
std::pair< int, double >( 35, 1.5 )
```

Создание пар и работу с ними иллюстрируют листинги 15.34, 15.35.

Листинг 15.34. Файл `pair.cpp`

```
/*
    Создание пар и работа с ними.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>        // Поточковый ввод-вывод
#include <utility>          // Пары

using namespace std;      // Используем стандартное
                           // пространство имен

int main( void )          // Возвращает 0 при успехе
{
    // Создание пар и печать их значений - используются все разновидности
    // конструкторов
    pair< int, double >
        p1( 21, 12.12 ), p2( p1 ), p3;
    cout << "p1: " << p1.first << " " << p1.second << endl;
    cout << "p2: " << p2.first << " " << p2.second << endl;
    cout << "p3: " << p3.first << " " << p3.second << endl << endl;

    // Присваивание значения - эквивалентно
    // p3 = pair< int, double >( 31, 21.12 );
    p3 = make_pair( 31, 21.12 );
```

```

// Печать
cout << "p3 = make_pair( 31, 21.12 );" << endl;
cout << "p3: " << p3.first << " " << p3.second << endl;

// Вычитание и сравнение пар с выводом результатов
cout << "Проверяем p1 == p2" << endl;
if( p1==p2 )
    cout << "Получили p1 == p2" << endl;
p2.first += 10;
cout << "\np2: " << p2.first << " " << p2.second << endl;
cout << "Проверяем p2 > p1" << endl;
if( p2>p1 )
    cout << "Получили p2 > p1" << endl;
p2.first -= 10; p1.second += 1;
cout << "\np1: " << p1.first << " " << p1.second << endl;
cout << "p2: " << p2.first << " " << p2.second << endl;
cout << "Проверяем p2 < p1" << endl;
if( p2<p1 )
    cout << "Получили p2 < p1" << endl;

cout << endl;

return 0;
}

```

Листинг 15.35. Результаты выполнения программы

```

p1: 21  12.12
p2: 21  12.12
p3: 0  0

p3 = make_pair( 31, 21.12 );
p3: 31  21.12
Проверяем p1 == p2
Получили p1 == p2

p2: 31  12.12
Проверяем p2 > p1
Получили p2 > p1

p1: 21  13.12
p2: 21  12.12
Проверяем p2 < p1
Получили p2 < p1

Press any key to continue

```

15.9. Ассоциативные контейнеры. Словари (*map*)

Элементами словарей и словарей с дубликатами являются пары "ключ/значение". Сортировка элементов производится автоматически на основании критерия сортировки, применяемого к ключу. По умолчанию используется критерий сортировки `< (less)`. Словари и словари с дубликатами отличаются только тем, что последние могут содержать дубликаты с одинаковыми значениями ключей, а первые — нет.

Шаблон словаря содержит три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение "меньше". Прежде чем продолжить дальнейшее изложение, кратко остановимся на функциональных классах, предназначенных для перегрузки операций вызова функций.

Перегрузка операции вызова функции и функциональный класс. Класс, в котором определена только операция вызова функции, называется *функциональным классом*. От такого класса не требуется наличия других членов:

```
class greater
{
    public:

        int operator( ) ( int a, int b ) const
        {
            return a>b;
        }
};
```

Использование функционального класса имеет специфический синтаксис. Рассмотрим пример.

```
#include <iostream>
// ...
greater      x;           // Создаем объект функции
// Будет выведен 0
cout << x( 1, 5 ) << endl;
// Будет выведена 1
cout << x( 5, 1 ) << endl;
```

Из примера следует, что объект функционального класса используется так, как если бы он был функцией. Попутно заметим, что вместо использования `x(1, 5)` можно было использовать более длинные конструкции `x.operator()(1, 5)` или `greater()(1, 5)`, но это менее удобно.

Чтобы использовать словарь в программе, необходимо добавить в нее включаемый файл `<map>`:

```
#include <map>
```

Тип словаря объявляется как шаблон классов в стандартном пространстве имен `std` (приведен с некоторыми упрощениями и сокращениями — листинг 15.36).

Листинг 15.36. Шаблон классов map

```

namespace std
{
    // TEMPLATE CLASS map. Первый параметр шаблона определяет тип ключа,
    // а второй - тип значения элемента. Необязательный третий параметр
    // задает критерий сортировки. По умолчанию - это less< Key >.
    // Необязательный четвертый параметр определяет модель памяти.
    // По умолчанию используется модель allocator, определенная
    // в стандартной библиотеке
    template < class Key, class T, class Compare = less< Key >,
              class Al = allocator< const Key, T > >

    class map
    {
    public:

        // Типы
        typedef Key          key_type;
        typedef T            mapped_type;
        typedef pair< const Key, T > value_type;
        typedef Compare      key_compare;
        typedef Al           allocator_type;
        typedef typename Al::reference      reference;
        typedef typename Al::const_reference const_reference;

        typedef implementation defined      iterator;
        typedef implementation defined      const_iterator;
        typedef implementation defined      size_type;
        typedef implementation defined      difference_type;
        typedef typename Al::pointer        pointer;
        typedef typename Al::const_pointer  const_pointer;
        typedef std::reverse_iterator<iterator>      reverse_iterator;
        typedef std::reverse_iterator< const_iterator> const_reverse_iterator;

        // Конструкторы
        explicit map( const Compare &comp = Compare( ),
                     const Al &a = Al( ) );

        template < class InputIter >
        map( InputIter first, InputIter last,
             const Compare &comp = Compare( ), const Al &a = Al( ) );
        map( const map< Key, T, Compare, Al > &x );

        // Деструктор

```

```

~map( );

// Присваивание
map< Key, T, Compare, Al > & operator=(
    const map< Key, T, Compare, Al > &x );

// Итераторы
iterator          begin( );
const_iterator    begin( ) const;
iterator          end( );
const_iterator    end( ) const;
reverse_iterator  rbegin( );
const_reverse_iterator rbegin( ) const;
reverse_iterator  rend( );
const_reverse_iterator rend( ) const;

// Параметры
size_type         size( ) const;
size_type         max_size( ) const;
bool              empty( ) const;

// Доступ к элементу по ключу
T &               operator[ ]( const Key & k );

// Вставка элементов
pair < iterator, bool > insert(const value_type &x );
iterator          insert( iterator pos,
    const value_type &x );
template < class InputIter >
void              insert( InputIter first,
    InputIter last );

// Удаление элементов
iterator          erase( iterator pos );
iterator          erase( iterator first, iterator last );
size_type         erase( const key_type &k );
void              clear( );

// Обмен всех элементов двух словарей
void              swap( map< Key, T, Compare, Al > &x );

// Поиск элементов
iterator          find( const key_type &k );
const_iterator    find( const key_type &k ) const;
size_type         count( const key_type &k ) const;

```

```

        iterator          lower_bound( const key_type &k );
        const_iterator    lower_bound( const key_type &k ) const;
        iterator          upper_bound( const key_type &k );
        const_iterator    upper_bound( const key_type &k ) const;

pair< iterator, iterator >
        equal_range( const key_type &k );
pair< const_iterator, const_iterator >
        equal_range( const key_type &k ) const;

    ...
};

// Операции отношений
template < class Key, class T, class Compare, class Al >
    bool operator==( const map< Key, T, Compare, Al > &x,
                     const map< Key, T, Compare, Al > &y );
template < class Key, class T, class Compare, class Al >
    bool operator!=( const map< Key, T, Compare, Al > &x,
                     const map< Key, T, Compare, Al > &y );
template < class Key, class T, class Compare, class Al >
    bool operator>=( const map< Key, T, Compare, Al > &x,
                     const map< Key, T, Compare, Al > &y );
template < class Key, class T, class Compare, class Al >
    bool operator<=( const map< Key, T, Compare, Al > &x,
                     const map< Key, T, Compare, Al > &y );
template < class Key, class T, class Compare, class Al >
    bool operator>( const map< Key, T, Compare, Al > &x,
                    const map< Key, T, Compare, Al > &y );
template < class Key, class T, class Compare, class Al >
    bool operator<( const map< Key, T, Compare, Al > &x,
                    const map< Key, T, Compare, Al > &y );
}

```

Кратко прокомментируем шаблонный класс. Элементы словаря могут состоять из произвольных типов `key` и `T`, удовлетворяющих двум условиям:

- пара "ключ/значение" должна поддерживать присваивание и копирование;
- ключ должен быть совместим с критерием сортировки.

Конструкторы. Шаблон словаря содержит три конструктора. Первый по порядку конструктор создает пустой словарь, используя указанный в первом параметре функциональный объект. Второй конструктор создает словарь и записывает в него элементы, определяемые диапазоном указанных итераторов. Время работы этого конструктора пропорционально количеству записываемых элементов, если они упорядочены, и квадрату количества элементов, если записываемые элементы не упорядочены. Третий конструктор является конструктором копирования и выполняет копирование словаря того же типа (с копированием всех элементов).

Как и для всех контейнеров, для словаря определены **деструктор, операция присваивания, операции отношений и итераторы**. Для доступа к элементам словаря по ключу определена **операция индексации** `[]` — см. листинг 15.36. С помощью этой операции можно не только получать значения элементов, но и добавлять в словарь новые.

ЗАМЕЧАНИЕ

Для словаря допустимы операции `++` и `--`, но не допустимы операции `+` и `-`.

Для **поиска элементов** в словаре определены методы `find()`, `count()`, `lower_bound()` и `upper_bound()` — см. листинг 15.36. Метод `find()` возвращает итератор на найденный элемент в случае успешного поиска или `end()` в противном случае. Метод `count()` возвращает количество элементов с заданным значением ключа (таких элементов может быть 0 или 1). Метод `lower_bound()` возвращает итератор на первый элемент, ключ которого больше заданного значения, или `end()`, если такого нет. Метод `upper_bound()` возвращает итератор на первый элемент, ключ которого не меньше заданного, или `end()`, если такого нет (если элемент с заданным ключом есть в словаре, будет возвращен итератор на него).

Для **вставки и удаления элементов** словаря определены методы:

```
pair< iterator, bool > insert( const value_type &x );
iterator insert( iterator pos, const value_type &x );
template < class InputIter >
    void insert( InputIter first, InputIter last );
iterator erase( iterator pos );
iterator erase( iterator first, iterator last );
size_type erase( const key_type &k );
void clear( );
```

Первая форма метода `insert()` используется для вставки в словарь пары "ключ/значение". Метод возвращает пару, состоящую из итератора, указывающего на вставленное значение, и признака результата операции `true`, если записи с таким ключом в словаре не было (в этом случае происходит добавление элемента в словарь). В противном случае возвращаемый итератор указывает на существующую запись и булевский признак результата операции равен `false` (добавление элемента в словарь не выполнено). Из сказанного следует, что с помощью данного метода нельзя скорректировать существующую запись. Это можно сделать с помощью операции *индексации*.

Вторая форма метода `insert()` применяется для ускорения процесса вставки. С этой целью ему передается первым параметром позиция словаря, начиная с которой требуется осуществить поиск места вставки. Если указанная позиция находится после места, в которое требуется вставить элемент, то вставка все равно будет выполнена верно. Вставка осуществляется только в случае отсутствия заданного ключа в словаре. Метод возвращает итератор на элемент словаря с заданным ключом. Например, если известно, что элементы будут помещаться в словарь в порядке возрастания, то можно передавать первым параметром в функцию вставки итератор предыдущего занесенного элемента. В этом случае время вставки элемента будет минимальным и постоянным.

Третья форма метода `insert()` используется для вставки группы элементов, определяемой диапазоном итераторов.

ЗАМЕЧАНИЕ

Операции вставки в словарь не приводят к порче связанных с элементами словаря итераторов и ссылок.

Методы удаления элементов и очистки словаря аналогичны одноименным функциям других контейнеров: первая форма метода `erase()` удаляет элемент словаря из позиции, заданной итератором, вторая — удаляет диапазон элементов, а третья — удаляет элемент с заданным ключом.

ЗАМЕЧАНИЕ

Операции удаления делают *недействительными* только операции и ссылки, связанные с удаляемыми элементами.

Для **обмена всех элементов** двух словарей применяется метод `swap()`.

Применение операций со словарем иллюстрируют листинги 15.37, 15.38. Советуем внимательно изучить их и провести эксперимент — это принесет большую пользу.

Листинг 15.37. Файл `map.cpp`

```
/*
    Работа со словарем (телефонной книгой) с использованием ассоциативного
    контейнера map. Ключ телефонной книги - фамилия, элемент - номер телефона.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <fstream>           // файловый поток
#include <iostream>          // потоковый ввод-вывод
#include <string>             // Класс для работы со строками
#include <map>               // Ассоциативные контейнеры

using namespace            // Используем стандартное
    std;                  // пространство имен

// Более короткий тип словаря
typedef map< string, long, less< string > > map_dic;

int main( void )           // Возвращает 0 при успехе
{
    // Создание пустого словаря - телефонной книги
    map_dic    dic1;

    // Заполнение телефонной книги данными из файла phonebook.txt
    // (в каждой строке - номер телефона, пробел, фамилия)
    ifstream   in( "phonebook.txt" );
    string     fam;        // фамилия (ключ)
    long       num;        // Номер телефона (элемент)
    cout << "Данные для телефонной книги:" << endl;
```

```
for( unsigned k=0; k<3; k++ )
{
    in >> num;          // Чтение номера
    // В реальной программе здесь следует проверить и обработать
    // достижение конца файла или ошибку чтения
    in.get( );          // Пропуск пробела
    // В реальной программе здесь следует проверить и обработать
    // достижение конца файла или ошибку чтения
    // Чтение фамилии
    getline( in, fam );
    // В реальной программе здесь следует проверить и обработать
    // достижение конца файла или ошибку чтения
    // Занесение в телефонную книгу
    dic1[ fam ] = num;
    // Вывод на экран
    cout << num << " " << fam << endl;
}

// Дополнение словаря
dic1[ "Николаев Николай Николаевич" ] = 1234567;

// Модификация элемента словаря
dic1[ "Сергеев Сергей Сергеевич" ] = 4444444;

// Вывод словаря на экран
cout << "\ndic1:" << endl;
map_dic::iterator
    i;          // Итератор словаря
for( i = dic1.begin( ); i != dic1.end( ); i++ )
{
    cout << i->first << " " << i->second << endl;
}

// Вывод элемента по ключу
cout << "\nВывод элемента по ключу" << endl;
cout << "Сергеев Сергей Сергеевич: " <<
    dic1[ "Сергеев Сергей Сергеевич" ] << endl;

// Присваивание словаря и вывод его на экран
map_dic    dic2 = dic1;
cout << "\nПосле dic2 = dic1 состояние dic2:" << endl;
for( i = dic2.begin( ); i != dic2.end( ); i++ )
{
    cout << i->first << " " << i->second << endl;
}
```

```

// Поиск элементов в словаре
getline( in, fam );
cout << "\nПоиск в dic1 элемента с ключом:\n" << fam << endl;
if( dic1.find( fam ) != dic1.end( ) )
{
    cout << "Нашли:" << fam << endl;
}
getline( in, fam );
cout << "\nПоиск в dic1 элемента с ключом:\n" << fam << endl;
if( dic1.find( fam ) == dic1.end( ) )
{
    cout << "Не нашли:" << endl;
    cout << "( *dic1.upper_bound( fam ) ).first:" <<
        ( *dic1.upper_bound( fam ) ).first << endl;
    cout << "( *dic1.lower_bound( fam ) ).first:" <<
        ( *dic1.lower_bound( fam ) ).first << endl;
}

in.close( );

// Вставка элементов в словарь и их удаление
dic2.insert( map_dic::value_type( "Петров", 5555555 ) );
fam = "Petrov";
dic2.insert( make_pair( fam, 6666666 ) );
// Попытка вставить запись с существующим ключом
dic2.insert( make_pair( fam, 7777777 ) );
fam = "Иванов Иван Иванович";
dic2.erase( fam );
// Вставка в dic2 первого элемента из dic1
i = dic1.begin( );
dic2.insert( *i );
// Быстрая вставка
fam = "Rogov"; i = dic2.begin( );
dic2.insert( i, make_pair( fam, 9999999 ) );
cout << "\nПосле вставок и удалений состояние dic2:" << endl;
for( i = dic2.begin( ); i != dic2.end( ); i++ )
{
    cout << i->first << " " << i->second << endl;
}

// Обмен местами dic1 и dic2, очистка dic1 и их печать
cout << "\nОбмен местами dic1 и dic2, очистка dic1 и их печать"
    << endl;
dic2.swap( dic1 );
dic1.clear( );
cout << "dic1:" << endl;

```

```

for( i = dic1.begin( ); i != dic1.end( ); i++ )
{
    cout << i->first << " " << i->second << endl;
}
cout << "dic2:" << endl;
for( i = dic2.begin( ); i != dic2.end( ); i++ )
{
    cout << i->first << " " << i->second << endl;
}

cout << endl;

return 0;
}

```

Листинг 15.38. Результаты выполнения программы

Данные для телефонной книги:

2222222 Петров Петр Петрович

1111111 Иванов Иван Иванович

3333333 Сергеев Сергей Сергеевич

dic1:

Иванов Иван Иванович 1111111

Николаев Николай Николаевич 1234567

Петров Петр Петрович 2222222

Сергеев Сергей Сергеевич 4444444

Вывод элемента по ключу

Сергеев Сергей Сергеевич: 4444444

После dic2 = dic1 состояние dic2:

Иванов Иван Иванович 1111111

Николаев Николай Николаевич 1234567

Петров Петр Петрович 2222222

Сергеев Сергей Сергеевич 4444444

Поиск в dic1 элемента с ключом:

Петров Петр Петрович

Нашли:Петров Петр Петрович

Поиск в dic1 элемента с ключом:

Петров Петр Васильевич

Не нашли:

(*dic1.upper_bound(fam)).first:Петров Петр Петрович

(*dic1.lower_bound(fam)).first:Петров Петр Петрович

После вставок и удалений состояние dic2:

```
Petrov 6666666
Rogov 9999999
Иванов Иван Иванович 1111111
Николаев Николай Николаевич 1234567
Петров 5555555
Петров Петр Петрович 2222222
Сергеев Сергей Сергеевич 4444444
```

Обмен местами dic1 и dic2, очистка dic1 и их печать

```
dic1:
dic2:
Иванов Иван Иванович 1111111
Николаев Николай Николаевич 1234567
Петров Петр Петрович 2222222
Сергеев Сергей Сергеевич 4444444
```

Press any key to continue

15.10. Ассоциативные контейнеры. Словари с дубликатами (*multimap*)

Как уже говорилось ранее, словари с дубликатами (*multimap*) допускают хранение элементов с одинаковыми ключами. Ввиду этого в словарях с дубликатами не определена операция индексации "[]", а добавление с помощью метода *insert()* выполняется успешно в любом случае. Метод *insert()* возвращает итератор на вставленный элемент.

Элементы с одинаковыми ключами хранятся в словаре с дубликатами в порядке их занесения. При удалении элемента по ключу метод возвращает количество удаленных элементов. Метод *equal_range()* возвращает диапазон итераторов, определяющий все вхождения элемента с заданным ключом. Метод *count()* может вернуть значение, большее единицы. В остальном словари с дубликатами *аналогичны* обычным словарям.

Далее приводится пример, иллюстрирующий указанные особенности (листинги 15.39, 15.40).

Листинг 15.39. Файл *multimap.cpp*

```
/*
    Работа со словарем (телефонной книгой) с дубликатами с использованием
    ассоциативного контейнера multimap. Ключ телефонной книги - фамилия,
    значение элемента - номер телефона.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <fstream>           // файловый поток
```

```

#include <iostream>          // ПотокОВЫЙ ВВод-Вывод
#include <string>             // Класс для работы со строками
#include <map>                // Ассоциативные контейнеры

using namespace             // Используем стандартное
    std;                    // пространство имен

// Более короткий тип словаря
typedef multimap< string, long, less< string > > multimap_dic;

int main( void )            // Возвращает 0 при успехе
{
    // Создание пустого словаря с дубликатами
    multimap_dic
        dic1;

    // Заполнение словаря
    string    fam = "Петров П.П.";
    dic1.insert( make_pair( fam, 3333333 ) );
    dic1.insert( make_pair( fam, 2222222 ) );
    dic1.insert( make_pair( fam, 4444444 ) );
    fam = "Иванов И.И.";
    dic1.insert( make_pair( fam, 1111111 ) );
    fam = "Сидоров С.С.";
    dic1.insert( make_pair( fam, 5555555 ) );

    // Вывод словаря на экран
    cout << "Исходное состояние словаря с дубликатами dic1:" << endl;
    multimap_dic::iterator
        i;                    // Итератор словаря
    for( i = dic1.begin( ); i != dic1.end( ); i++ )
    {
        cout << i->first << " " << i->second << endl;
    }

    // Получение информации об элементе с заданным ключом
    //    fam = "Петров П.П."
    fam = "Петров П.П.";
    cout << "\nВ словаре с дубликатами dic1 содержится "
        << dic1.count( fam ) <<
        " элемента \ncс ключом <Петров П.П.> " << endl;
    pair< multimap_dic::iterator, multimap_dic::iterator >
        p;
    p = dic1.equal_range( fam );
    cout << "Первый из этих элементов: " << endl;
    cout << p.first->first << " " << p.first->second << endl;
}

```

```

cout << "Последний из этих элементов: " << endl;
p.second--;
cout << p.second->first << " " << p.second->second << endl;

cout << endl;

return 0;
}

```

Листинг 15.40. Результаты выполнения программы

Исходное состояние словаря с дубликатами dic1:

```

Иванов И.И. 1111111
Петров П.П. 3333333
Петров П.П. 2222222
Петров П.П. 4444444
Сидоров С.С. 5555555

```

В словаре с дубликатами dic1 содержится 3 элемента
с ключом <Петров П.П.>

Первый из этих элементов:

```
Петров П.П. 3333333
```

Последний из этих элементов:

```
Петров П.П. 4444444
```

Press any key to continue

15.11. Ассоциативные контейнеры. Множества (set)

Множество — это ассоциативный контейнер, содержащий только значения ключей. Это означает, что тип `value_type` соответствует типу `key`. Тип множества объявляется как шаблон классов во включаемом файле `<set>` в стандартном пространстве имен `std` и имеет следующий вид (приведен с некоторыми упрощениями и сокращениями) — листинг 15.41.

Листинг 15.41. Шаблон классов `set`

```

namespace std
{
    // TEMPLATE CLASS set. Первый параметр шаблона определяет тип ключа,
    // а необязательный второй параметр - задает критерий сортировки.
    // По умолчанию - это less< Key >. Необязательный третий параметр
    // определяет модель памяти. По умолчанию используется модель
    // allocator, определенная в стандартной библиотеке
    template < class Key, class Compare = less< Key >,

```

```

        class Al = allocator< Key > >

class set
{
public:

    // Типы
    typedef Key          value_type;
    typedef Key          key_type;
    typedef Compare      value_compare;
    typedef Compare      key_compare;
    typedef Al           allocator_type;
    typedef typename Al::reference      reference;
    typedef typename Al::const_reference const_reference;

    typedef implementation defined      iterator;
    typedef implementation defined      const_iterator;
    typedef implementation defined      size_type;
    typedef implementation defined      difference_type;
    typedef typename Al::pointer        pointer;
    typedef typename Al::const_pointer  const_pointer;
    typedef std::reverse_iterator<iterator>      reverse_iterator;
    typedef std::reverse_iterator< const_iterator> const_reverse_iterator;

    // Конструкторы
    explicit          set( const Compare &comp = Compare( ),
                          const Al &All = Al( ) );

    template < class InputIter >
        set( InputIter first, InputIter last,
              const Compare &comp = Compare( ), const Al &All = Al( ) );
        set( const set< Key, Compare, Al > &x );

    // Деструктор
    ~set( );

    // Присваивание
    set< Key, Compare, Al > &
        operator=(
            const set< Key, Compare, Al > &x );

    // Методы просмотра на базе итераторов
    iterator          begin( );
    const_iterator    begin( ) const;
    iterator          end( );

```

```

const_iterator      end( ) const;
reverse_iterator    rbegin( );
const_reverse_iterator rbegin( ) const;
reverse_iterator     rend( );
const_reverse_iterator rend( ) const;

// Получение сведений
size_type           size( ) const;
size_type           max_size( ) const;
bool                empty( ) const;

// Вставка
pair< iterator, bool > insert( const value_type &x );
iterator             insert( iterator pos,
                             const value_type &x );

template < class InputIter >
void                insert( InputIter first,
                           InputIter last );

// Удаление
iterator            erase( iterator pos );
iterator            erase( iterator first, iterator last );
size_type           erase( const key_type &KeyValue );
void               clear( );

// Обмен
void               swap( set < Key, Compare, Al > &x );

// Поиск
const_iterator      find( const key_type &KeyValue ) const;
size_type           count( const key_type &KeyValue ) const;
const_iterator      lower_bound(
    const key_type &KeyValue ) const;
const_iterator      upper_bound(
    const key_type &KeyValue ) const;
pair< iterator, iterator >
    equal_range(
        const key_type &KeyValue ) const;

...
};

// Операции отношений
template < class Key, class Compare, class Al >
bool operator==( const set< Key, Compare, Al > &x,

```

```

        const set< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator!=( const set< Key, Compare, Al > &x,
                    const set< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator<=( const set< Key, Compare, Al > &x,
                    const set< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator>=( const set< Key, Compare, Al > &x,
                    const set< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator<( const set< Key, Compare, Al > &x,
                    const set< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator>( const set< Key, Compare, Al > &x,
                    const set< Key, Compare, Al > &y );
}

```

Элементы множества относятся к произвольному типу *key*, поддерживающему присваивание, копирование и сравнение по критерию сортировки. Критерий сортировки должен обеспечивать строгую упорядоченность [10].

Множество, как и другие ассоциативные контейнеры, обычно реализуется в виде сбалансированного бинарного дерева [3]. В стандарте языка такая реализация напрямую не указана, но она обусловлена косвенно требованием к сложности операций над множеством. Главное достоинство *автоматической* сортировки элементов множества заключается в том, что бинарное дерево обеспечивает хорошие показатели при поиске во множестве элемента с конкретным значением (методы поиска имеют логарифмическую сложность).

Следует обратить особое внимание на то, что автоматическая сортировка не позволяет изменять значение элемента *напрямую* — необходимо вначале удалить из множества элемент со старым значением, а затем вставить элемент с новым значением. Следствием этого являются две особенности интерфейса множества (см. листинг 15.41):

- множество не поддерживает прямое обращение к элементам;
- при косвенном обращении с помощью итераторов значение элемента является константным с точки зрения итератора.

Из приведенного объявления следует, что интерфейс множества практически аналогичен интерфейсу словаря. Тем не менее, приведем несложный иллюстрирующий пример (листинги 15.42, 15.43).

Листинг 15.42. Файл `set.cpp`

```
/*
```

```
Работа с множеством при использовании ассоциативного контейнера set.
```

```
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
```

```

*/

#include <iostream>           // Поточковый ввод-вывод
#include <string>             // Класс для работы со строками
#include <set>                // Ассоциативные контейнеры

using namespace             // Используем стандартное
    std;                   // пространство имен

// Более короткий тип множества
typedef set< string, less< string > > set_s;

// Итератор
set_s::iterator
    i;

int main( void )           // Возвращает 0 при успехе
{
    // Массив строк
    string    s[ 4 ] = { "Таня", "Лена", "Оля", "Ира" };

    set_s     s1;          // Пустое множество строк
    // Создание множества копированием массива строк
    set_s     s2( s, s+4 );
    // Копирование множества - используется конструктор копирования
    set_s     s3( s2 );

    // Вывод содержимого созданных множеств на экран
    cout << "s1: ";
    for( i = s1.begin( ); i != s1.end( ); i++ )
        cout << *i << " ";
    cout << "\ns2: ";
    for( i = s2.begin( ); i != s2.end( ); i++ )
        cout << *i << " ";
    cout << "\ns3: ";
    for( i = s3.begin( ); i != s3.end( ); i++ )
        cout << *i << " ";

    // Вставка элементов в множество s2 и вывод содержимого множества на
    // экран
    s2.insert( "Маша" );
    s2.insert( "Катя" );
    s2.insert( "Лена" );
    cout << "\n\nСостояние множества s2 после вставки:\n";
    for( i = s2.begin( ); i != s2.end( ); i++ )

```

```

    cout << *i << " ";

    // Использование метода equal_range( )
    pair< set_s::iterator, set_s::iterator >
        p; // Для хранения результата equal_range( )
    p = s2.equal_range( "Лена" );
    cout << "\n\nДиапазон элементов со значением \"Лена\": " <<
        *(p.first) << " " << *(p.second);
    p = s2.equal_range( "Коля" );
    cout << "\n\nДиапазон элементов со значением \"Коля\": " <<
        *(p.first) << " " << *(p.second);

    cout << endl << endl;

    return 0;
}

```

Листинг 15.43. Результаты выполнения программы

```

s1:
s2: Ира Лена Оля Таня
s3: Ира Лена Оля Таня

Состояние множества s2 после вставки:
Ира Катя Лена Маша Оля Таня

Диапазон элементов со значением "Лена":Лена Маша

Диапазон элементов со значением "Коля":Лена Лена

Press any key to continue

```

Как и в словаре, элементы во множестве хранятся отсортированными. Повторяющиеся элементы во множество не заносятся. Для работы с множествами в стандартной библиотеке определены алгоритмы, которые будут рассмотрены далее.

15.12. Ассоциативные контейнеры. Множества с дубликатами (*multiset*)

Во множествах с дубликатами ключи могут повторяться и поэтому операция вставки элемента всегда выполняется успешно, а метод `insert()` возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся во множестве с дубликатами в порядке их занесения. Метод `find()` возвращает итератор на первый найденный элемент или `end()`, если ни одного элемента с заданным ключом не найдено.

При работе с одинаковыми ключами во множестве с дубликатами часто используются методы `count()`, `lower_bound()`, `upper_bound()` и `equal_range()`.

Тип множества с дубликатами объявляется как шаблон классов во включаемом файле `<set>` в стандартном пространстве имен `std` и имеет следующий вид (приведен с некоторыми упрощениями и сокращениями) — листинг 15.44.

Листинг 15.44. Шаблон классов `multiset`

```
namespace std
{
    // TEMPLATE CLASS set. Первый параметр шаблона определяет тип ключа,
    // а необязательный второй параметр - задает критерий сортировки.
    // По умолчанию - это less< Key >. Необязательный третий параметр
    // определяет модель памяти. По умолчанию используется модель
    // allocator, определенная в стандартной библиотеке
    template < class Key, class Compare = less< Key >,
               class Al = allocator< Key > >
    class multiset
    {
    public:

        // Типы
        typedef Key value_type;
        typedef Key key_type;
        typedef Compare value_compare;
        typedef Compare key_compare;
        typedef Al allocator_type;
        typedef typename Al::reference reference;
        typedef typename Al::const_reference const_reference;

        typedef implementation defined iterator;
        typedef implementation defined const_iterator;
        typedef implementation defined size_type;
        typedef implementation defined difference_type;
        typedef typename Al::pointer pointer;
        typedef typename Al::const_pointer const_pointer;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator< const_iterator> const_reverse_iterator;

        // Конструкторы
        explicit multiset(
            const Compare &comp = Compare( ), const Al &All = Al( ) );
```

```

template < class InputIter >
    multiset( InputIter first,
        InputIter last, const Compare &comp = Compare( ),
        const Al &All = Al( ) );
    multiset(
        const multiset< Key, Compare, Al > &x );

// Деструктор
~multiset( );

// Присваивание
multiset< Key, Compare, Al > &
    operator=( const multiset< Key,
        Compare, Al > &x );

// Методы просмотра на базе итераторов
iterator          begin( );
const_iterator  begin( ) const;
iterator          end( );
const_iterator  end( ) const;
reverse_iterator  rbegin( );
const_reverse_iterator rbegin( ) const;
reverse_iterator  rend( );
const_reverse_iterator rend( ) const;

// Получение сведений
size_type         size( ) const;
size_type         max_size( ) const;
bool             empty( ) const;

// Вставка
iterator          insert( const value_type &x );
iterator          insert( iterator pos,
                        const value_type &x );

template < class InputIter >
void             insert( InputIter first,
                        InputIter last );

// Удаление
void             erase( iterator pos );
void             erase( iterator first, iterator last );
size_type         erase( const key_type &KeyValue );
void             clear( );

// Обмен

```

```

void swap( multiset < Key, Compare, Al > &x );

// Поиск
iterator find( const key_type &KeyValue ) const;
size_type count( const key_type &KeyValue ) const;
iterator lower_bound(
    const key_type &KeyValue ) const;
iterator upper_bound(
    const key_type &KeyValue ) const;
pair< iterator, iterator >
    equal_range(
        const key_type &KeyValue ) const;

...
};

// Операции отношений
template < class Key, class Compare, class Al >
    bool operator==( const multiset< Key, Compare, Al > &x,
        const multiset< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator!=( const multiset< Key, Compare, Al > &x,
        const multiset< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator<= ( const multiset< Key, Compare, Al > &x,
        const multiset< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator>= ( const multiset< Key, Compare, Al > &x,
        const multiset< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator< ( const multiset< Key, Compare, Al > &x,
        const multiset< Key, Compare, Al > &y );
template < class Key, class Compare, class Al >
    bool operator> ( const multiset< Key, Compare, Al > &x,
        const multiset< Key, Compare, Al > &y );
}

```

Приведем несложный иллюстрирующий пример, аналогичный предыдущему (листинги 15.45, 15.46).

Листинг 15.45. Файл multiset.cpp

```

/*
Работа с множеством с дубликатами при использовании ассоциативного контейнера multiset.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

```

*/

#include <iostream>           // Поточковый ввод-вывод
#include <string>              // Класс для работы со строками
#include <set>                 // Ассоциативные контейнеры

using namespace              // Используем стандартное
    std;                     // пространство имен

// Более короткий тип множества с дубликатами
typedef multiset< string, less< string > > mset_s;

// Итератор
mset_s::iterator
    i;

int main( void )             // Возвращает 0 при успехе
{
    // Массив строк
    string    s[ 4 ] = { "Таня", "Лена", "Оля", "Ира" };

    mset_s     ms1;           // Пустое множество строк
    // Создание множества строк с дубликатами копированием массива строк
    mset_s     ms2( s, s+4 );
    // Копирование множества - используется конструктор копирования
    mset_s     ms3( ms2 );

    // Вывод содержимого созданных множеств на экран
    cout << "ms1: ";
    for( i = ms1.begin( ); i != ms1.end( ); i++ )
        cout << *i << " ";
    cout << "\nms2: ";
    for( i = ms2.begin( ); i != ms2.end( ); i++ )
        cout << *i << " ";
    cout << "\nms3: ";
    for( i = ms3.begin( ); i != ms3.end( ); i++ )
        cout << *i << " ";

    // Вставка элементов в множество ms2 и вывод содержимого множества на
    // экран
    ms2.insert( "Маша" );
    ms2.insert( "Катя" );
    // Вставка дубликата
    ms2.insert( "Лена" );
    cout << "\n\nСостояние множества ms2 после вставки:\n";
    for( i = ms2.begin( ); i != ms2.end( ); i++ )

```

```

    cout << *i << " ";

    // Использование метода equal_range( )
    pair< mset_s::iterator, mset_s::iterator >
        p;          // Для хранения результата equal_range( )
    p = ms2.equal_range( "Лена" );
    cout << "\n\nДиапазон элементов со значением \"Лена\":" << *(p.first)
        << " " << *(p.second);
    p = ms2.equal_range( "Миша" );
    cout << "\n\nДиапазон элементов со значением \"Миша\":" << *(p.first)
        << " " << *(p.second);

    // Использование метода count( )
    cout << "\n\nМножество ms2 содержит " << ms2.count( "Лена" )
        << " элемента с ключом \"Лена\"";

    cout << endl << endl;

    return 0;
}

```

Листинг 15.46. Результаты выполнения программы

```

ms1:
ms2: Ира Лена Оля Таня
ms3: Ира Лена Оля Таня

Состояние множества ms2 после вставки:
Ира Катя Лена Лена Маша Оля Таня

Диапазон элементов со значением "Лена":Лена Маша

Диапазон элементов со значением "Миша":Оля Оля

Множество ms2 содержит 2 элемента с ключом "Лена"

Press any key to continue

```

15.13. Специальные контейнеры. Битовые множества (*bitset*)

Битовое множество представляет собой шаблон для представления и обработки *длинных* последовательностей битов *фиксированного* размера. Напомним, что длинные последовательности битов произвольного размера следует обрабатывать с помощью контейнера `vector<bool>`. Напомним также, что недлинные последовательности битов можно обрабатывать с помощью битовых операций над целыми операндами или с помощью операций над полями битов. Битовое множество по своей сути

представляет собой битовый массив, для которого определены операции произвольного доступа, изменение отдельных битов и массива в целом. Биты множества индексируются *справа налево*, начиная с 0.

Шаблон битового множества содержится во включаемом файле `<bitset>`, где он объявлен в стандартном пространстве имен `std`. Параметром шаблона является длина битовой последовательности, которая должна быть константой. Шаблон битового множества приведен в листинге 15.47 с некоторыми упрощениями и сокращениями.

Листинг 15.47. Шаблон классов `bitset`

```
namespace std
{
    // TEMPLATE CLASS bitset
    template < size_t N >
    class bitset
    {
    public:

        // Класс для адресации отдельного бита битового множества и
        // работы с ним
        class reference
        {
            friend class bitset;

            reference( );

        public:

            ~reference( );

            // Присваивание элементу битового множества значения x
            // ( b[ i ] = x )
            reference & operator=( bool x );

            // Присваивание b[ i ] = b[ j ]
            reference & operator=( const reference & );

            // Инверсия b[ i ]
            reference & flip( );

            bool operator~( ) const;

            // Присваивание x = b[ i ]
            operator bool( ) const;

        };

        // Конструкторы
        // Создает битовое множество из нулей
        bitset( );

        // Создает битовое множество и инициализирует его биты
```

```

// соответствующими битами x
        bitset( unsigned long x );

// Создает битовое множество: строка str должна состоять из нулей
// и единиц (иначе порождается исключение invalid_argument) и
// инициализирует соответствующие биты множества; pos и n
// задают позицию начала строки и количество символов, которые
// используются для инициализации. По умолчанию для
// инициализации элементов множества используется вся строка
// str
explicit bitset( const string &str,
        string::size_type pos = 0,
        string::size_type n = string::npos );

// Операции с битовыми множествами
// *****
// Перемножение элементов множеств
bitset< N > & operator&=( const bitset< N > &rop );
// Сложение элементов множеств
bitset< N > & operator|=( const bitset< N > &rop );
// "Исключающее ИЛИ" над элементами множеств
bitset< N > & operator^=( const bitset< N > &rop );
// Сдвиг элементов множества на pos элементов влево
// (освобождающиеся позиции заполняются нулями)
bitset< N > & operator<<=( size_t pos );
// Сдвиг элементов множества на pos элементов вправо
// (освобождающиеся позиции заполняются нулями)
bitset< N > & operator>>=( size_t pos );
// Задание единичных значений элементов множества
bitset< N > & set( );
// Задание значения x (по умолчанию true) элементу множества
// с индексом pos
bitset< N > & set( size_t pos, bool x = true );
// Задание нулевых значений элементов множества
bitset< N > & reset( );
// Задание нулевых значений элементу множества с индексом pos
bitset< N > & reset( size_t pos );
// Возвращает значение множества с инвертированными значениями
// элементов
bitset< N > operator~( ) const;
// Инвертирует значения элементов множества
bitset< N > & flip( );
// Инвертирует значения элемента множества с индексом pos
bitset< N > & flip( size_t pos );

// Индексация битов множества

```

```

reference                operator[]( size_t pos );

// Преобразование множества в длинное целое
unsigned long            to_ulong( ) const;
// Преобразование множества в строку
string                   to_string( ) const;
// Возвращает количество единичных элементов
size_t                   count( ) const;
// Возвращает количество элементов множества
size_t                   size( ) const;
// Сравнение множеств на равенство
bool                     operator==(
    const bitset< N > &rop ) const;
// Сравнение множеств на неравенство
bool                     operator!=(
    const bitset< N > &rop ) const;
// Возвращает true, если b[ pos ] == 1
bool                     test( size_t pos ) const;
// Возвращает true, если множество содержит хотя бы один
// единичный элемент
bool                     any( ) const;
// Возвращает true, если множество не содержит ни одного
// единичного элемента
bool                     none( ) const;
// Возвращает значение множества, элементы которого сдвинуты на
// rop элементов влево
bitset< N >               operator<<( size_t rop ) const;
// Возвращает значение множества, элементы которого сдвинуты на
// rop элементов вправо
bitset< N >               operator>>( size_t rop ) const;
};
}

```

Для адресации отдельного бита битового множества введен класс `reference` (см. листинг 15.47).

В битовом множестве не определены итераторы, и по этой причине битовое множество не является контейнером в чистом виде.

Рассмотрим иллюстрирующий пример (листинги 15.48, 15.49).

Листинг 15.48. Файл `bitset.cpp`

```

/*
    Работа с битовыми множествами с использованием контейнера bitset.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

```

```

#include <iostream>           // Потокный ввод-вывод
#include <string>              // Класс для работы со строками
#include <bitset>              // Шаблон битового множества

using namespace              // Используем стандартное
    std;                     // пространство имен

int main( void )             // Возвращает 0 при успехе
{
    // Создание битовых множеств
    // *****
    cout << "Создание битовых множеств" << endl;
    bitset< 32 >
        bs1;                 // 32 нуля
    // 000000000000000001111000011100001
    bitset< 32 >
        bs2( 0xF0E1 );
    // 000000000000000000000000000011100110
    bitset< 32 >
        bs3( "11100110" );
    // 00110
    bitset< 5 >
        bs4( "11100110", 3 );
    // 0110
    bitset< 4 >
        bs5( "11100110", 1, 3 );

    // Печать значений элементов созданных множеств
    cout << "bs1: " << bs1 << endl;
    cout << "bs2: " << bs2 << endl;
    cout << "bs3: " << bs3 << endl;
    cout << "bs4: " << bs4 << endl;
    cout << "bs5: " << bs5 << endl;

    // Индексация элементов
    cout << "Индексация элементов bs4: bs4[ 0 ] = bs4[ 1 ]; " << endl;
    bs4[ 0 ] = bs4[ 1 ];
    cout << "bs4: " << bs4 << endl;

    // Операции с битовыми множествами
    // *****
    cout << endl << "Операции с битовыми множествами: bs2 &= bs3;"
        << endl;
    bs2 &= bs3;
    cout << "bs2: " << bs2 << endl;

```

```

cout << "Операции с битовыми множествами: bs2 <= 3;" << endl;
bs2 <= 3;
cout << "bs2: " << bs2 << endl;
cout << "Операции с битовыми множествами: bs1.set( );" << endl;
bs1.set( );
cout << "bs1: " << bs1 << endl;
cout << "Операции с битовыми множествами: bs1.set( 4, false );"
    << endl;
bs1.set( 4, false );
cout << "bs1: " << bs1 << endl;
cout << "Операции с битовыми множествами: bs1.reset( 3 );" << endl;
bs1.reset( 3 );
cout << "bs1: " << bs1 << endl;
cout << "Операции с битовыми множествами: ~bs1" << endl;
cout << "~bs1: " << ~bs1 << endl;
cout << "Операции с битовыми множествами: bs1.flip( 1 );" << endl;
bs1.flip( 1 );
cout << "bs1: " << bs1 << endl;
cout << "Операции с битовыми множествами: bs1.reset( );" << endl;
bs1.reset( );
cout << "bs1: " << bs1 << endl;
cout << "Операции с битовыми множествами: cout << bs5.to_string( );"
    << endl;
cout << "bs5.to_string( ): " << bs5.to_string( ) << endl;

cout << endl;

return 0;
}

```

Листинг 15.49. Результаты выполнения программы

```

Создание битовых множеств
bs1: 00000000000000000000000000000000
bs2: 00000000000000000000000000000000
bs3: 00000000000000000000000000000000
bs4: 00110
bs5: 0110
Индексация элементов bs4: bs4[ 0 ] = bs4[ 1 ];
bs4: 00111

Операции с битовыми множествами: bs2 &= bs3;
bs2: 00000000000000000000000000000000
Операции с битовыми множествами: bs2 <= 3;
bs2: 00000000000000000000000000000000
Операции с битовыми множествами: bs1.set( );

```

```

bs1: 11111111111111111111111111111111
Операции с битовыми множествами: bs1.set( 4, false );
bs1: 1111111111111111111111111111101111
Операции с битовыми множествами: bs1.reset( 3 );
bs1: 1111111111111111111111111111001111
Операции с битовыми множествами: ~bs1
~bs1: 0000000000000000000000000000000011000
Операции с битовыми множествами: bs1.flip( 1 );
bs1: 1111111111111111111111111111001011
Операции с битовыми множествами: bs1.reset( );
bs1: 0000000000000000000000000000000000
Операции с битовыми множествами: cout << bs5.to_string( );
bs5.to_string( ): 0110

```

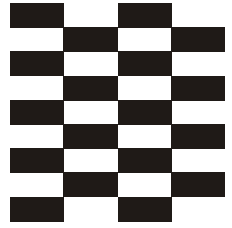
Press any key to continue

15.14. Вопросы и упражнения для самопроверки

1. Что собой представляют контейнеры, зачем они нужны?
2. Что представляют собой последовательные и ассоциативные контейнеры?
3. Назовите для контейнеров общие возможности, унифицированные типы и общие операции и методы.
4. Что собой представляет последовательный контейнер — вектор?
5. Что собой представляет последовательный контейнер — вектор логических значений?
6. Что собой представляет последовательный контейнер — двусторонняя очередь?
7. Что собой представляет последовательный контейнер — список?
8. Что собой представляет адаптер последовательного контейнера — стек?
9. Что собой представляет адаптер последовательного контейнера — очередь FIFO?
10. Что собой представляет адаптер последовательного контейнера — очередь с приоритетами?
11. Что представляет собой пара, используемая в ассоциативном контейнере?
12. Ассоциативные контейнеры: что представляют собой словари, словари с дубликатами, функциональные классы и функциональные объекты?
13. Ассоциативные контейнеры: назовите особенности множеств и множеств с дубликатами.
14. Специальный контейнер: что представляет собой битовое множество?
15. Напишите законченную программу, в которой создайте матрицу с заданными размерами, инициализируйте ее элементы и напечатайте, используя операцию "[]". Матрица должна быть представлена в программе как вектор векторов. Используйте последовательный контейнер-вектор.

Ответы на эти вопросы и упражнения приведены в разд. П1.12 приложения 1.

Глава 16



Итераторы и функциональные объекты

Итераторы и функциональные объекты довольно широко применяются в стандартной библиотеке языка C++. Начальные сведения о них были приведены в предыдущем разделе. Рассмотрим итераторы и функциональные объекты подробнее, чтобы их можно было уверенно применять в алгоритмах библиотеки, которые описываются в *гл. 17*.

16.1. Итераторы

Данные, по существу, представляют собой некоторую обобщенную последовательность. Вне зависимости от способа организации и типа данных требуются средства просмотра последовательности и доступа к каждому ее элементу. Такими средствами и являются *итераторы*. Таким образом, итератор является обобщенным указателем и семантика итератора и указателя одинаковы. По этой причине все функции, принимающие в качестве параметра итераторы, могут использовать и обычные указатели.

В стандартной библиотеке языка C++ итераторы используются для работы с контейнерными классами, потоками и буферами потоков. При работе с итераторами используются понятия *текущий указываемый элемент* и *указатель на следующий элемент*. Доступ к текущему элементу последовательности выполняется, как и при использовании указателей, с помощью операций "*" и "->". Переход к следующему элементу выполняется с помощью операции инкремента "++". Для всех итераторов определены также операции присваивания, проверки на равенство и неравенство. Тем не менее, разные итераторы обладают и разными свойствами. Иногда это очень существенно, поскольку для работы некоторых алгоритмов (*см. гл. 17*) необходимы особые свойства итераторов. Например, для алгоритмов сортировки нужны итераторы произвольного доступа — без них эффективность алгоритмов была бы слишком низкой.

Данные могут быть организованы различным образом (в виде массива, списка, дерева и т. д.) и для каждого вида последовательности требуется свой тип итератора, поддерживающий определенный набор операций. Пусть *i* и *j* — итераторы одного типа, *x* — переменная того же типа, что и элемент последовательности, а *n* — целая величина. Тогда для *любого типа итератора* допустимы выражения:

`i++ ++i i = j i == j i != j`

В соответствии с набором остальных, варьируемых операций, итераторы делятся на пять групп (табл. 16.1). Перечисленные в таблице операции выполняются за постоянное время.

Как следует из таблицы, *прямой итератор* поддерживает все операции *входных* и *выходных* итераторов и может использоваться везде, где требуются входные или выходные итераторы.

Таблица 16.1. Категории итераторов и их свойства

Категория итератора	Операция	Контейнеры
Входной (input)	x = *i;	Все
Выходной (output)	*i = x;	Все
Прямой (forward)	x = *i; *i = x;	Все
Двунаправленный (bidirectional)	x = *i; *i = x; --i; i--;	Все
Произвольного доступа (random access)	x = *i; *i = x; --i; i--; i+n; i-n; i += n; i -= n; i<j i>j i<=j i>=j	Все, кроме list

Двунаправленный итератор поддерживает все операции прямого итератора, а также декремент, и может использоваться везде, где требуется прямой итератор.

Итератор *произвольного* доступа поддерживает все операции двунаправленного, а кроме того, переход к произвольному элементу последовательности и сравнение итераторов.

Итераторные классы и методы определены в заголовочных файлах <iterator> и <utility>. Итератор может быть *действительным*, когда он указывает на какой-либо элемент, или *недействительным*. Итератор недействителен в следующих случаях:

- ❑ если он не был инициализирован;
- ❑ если контейнер, с которым он связан, изменил размеры или уничтожен;
- ❑ если итератор указывает на конец последовательности.

Конец последовательности представляется как указатель на элемент, следующий за последним элементом последовательности. Такой указатель всегда существует. Указанный прием позволяет не рассматривать пустую последовательность как особый случай (понятие "нулевой итератор" не существует).

Итераторы могут быть константными. Они используются тогда, когда изменять значения соответствующих элементов контейнера не нужно. Связанные с итератором типы описываются небольшим набором объявлений в шаблоне классов `iterator_traits`:

```
// TEMPLATE CLASS iterator_traits (from < iterator >)
template < class Iterator >
struct iterator_traits
{
    // Для разности между двумя итераторами
    typedef typename Iterator::distance_type
        distance_type;
```

```
// Тип элемента
typedef typename Iterator::value_type    value_type;
// Возвращаемый тип оператора ->
typedef typename Iterator::pointer      pointer;
// Возвращаемый тип оператора *()
typedef typename Iterator::reference    reference;
// Категории итераторов (см. табл. 16.1)
typedef typename Iterator::iterator_category iterator_category;
};
```

Стандартная библиотека обеспечивает пять классов, представляющих пять категорий итераторов (см. также табл. 16.1):

```
namespace std
{
    // ITERATOR TAGS (from < iterator >)
    struct input_iterator_tag {};
    struct output_iterator_tag {};
    struct forward_iterator_tag : public input_iterator_tag {};
    struct bidirectional_iterator_tag : public forward_iterator_tag {};
    struct random_access_iterator_tag
        : public bidirectional_iterator_tag {};
}
```

16.2. Обратные итераторы

По аналогии с последовательными контейнерами для двунаправленных итераторов и итераторов произвольного доступа существуют *итераторные адаптеры* — специальные итераторы, позволяющие выполнять алгоритмы, которые поддерживают:

- ☐ перебор элементов в обратном порядке (обратные итераторы);
- ☐ режим вставки (итераторы вставки);
- ☐ работу с потоками данных (потокковые итераторы).

Подобный адаптер, просматривающий последовательность в обратном направлении, обозначается `reverse_iterator`. *Обратные итераторы* представляют собой адаптеры, переопределяющие операции "++" и "--" так, что перебор элементов производится в обратном направлении. Для обратных итераторов поддерживаются обычные операторы отношения "==", "!=", "<", "<=", ">" и ">=". Обратные итераторы описаны в контейнерных классах для просмотра их элементов в обратном порядке. Кроме того, описаны методы `rbegin()` и `rend()`, возвращающие `reverse_iterator`:

```
vector< int > v;
...
for( vector< int >::reverse_iterator j = v.rbegin( ); j != rend( ); ++j )
    cout << *j << " ";
...

```

Если контейнер объявлен как `const`, то следует использовать итератор `const_reverse_iterator`.

16.3. Итераторы вставки

Итераторы вставки, так же, как и обратные итераторы, являются адаптерами итераторов. Они предназначены для добавления новых элементов в начало, конец или произвольное место контейнера. В стандартной библиотеке определено три шаблона классов итераторов вставки, построенных на основе выходных итераторов: `back_insert_iterator`, `front_insert_iterator` и `insert_iterator`.

Для вставки определены три метода.

```
template < class C >
    back_insert_iterator< C >      back_inserter( C &x );
template < class C >
    front_insert_iterator< C >    front_inserter( C &x );
template < class C, class Iterator >
    insert_iterator< C >          inserter( C &x, Iterator i );
```

Здесь `c` — контейнер, в который требуется вставить элементы. Метод `back_inserter()` вставляет элементы в конец контейнера, `front_inserter()` — в начало, а `inserter()` — перед элементом, на который ссылается его аргумент-итератор.

16.4. Поточковые итераторы

Поточковые итераторы обеспечивают при использовании стандартных алгоритмов возможность работы с потоками ввода-вывода. Определено два шаблона классов потоковых итераторов:

- итератор входного потока `istream_iterator`;
- итератор выходного потока `ostream_iterator`.

Итератор входного потока обеспечивает чтение элементов из потока, для которого он был создан. После этого к прочитанному элементу можно обращаться через операцию разадресации:

```
// Чтение целого числа из файла inp
istream      in( "inp" );
istream_iterator< int >
                i( in );
int          tmp = *i;
// Чтение очередного значения из входного потока
++i;
int          tmp1 = *i;
```

При достижении конца входного потока его итератор принимает значение конца ввода. Это же значение имеет умалчиваемый конструктор итератора. Поэтому цикл чтения из файла можно организовать следующим образом:

```
while( i != istream_iterator< int >( ) )
    cout << *i++ << " ";
```

Итераторы входного потока можно сравнивать на равенство и неравенство, причем все итераторы, равные концу ввода, равны между собой. Итераторы же, не равные концу ввода, можно сравнивать лишь при условии, что они сконструированы для

одного и того же потока. Особенностью итераторов входного потока является то, что они *не сохраняют* равенство после инкрементации (из $i == j$ не следует $++i == ++j$). Поэтому их рекомендуется использовать только в однопроходных алгоритмах.

Итератор выходного потока с помощью операции "<<" записывает элементы в выходной поток, для которого он был сконструирован. Если вторым аргументом конструктора итератора была строка символов, то она выводится после каждого выводимого значения.

```
ostream_iterator< int >
    out( cout, " сек.\n" );

*out = 10;                // Будет выведено: 10 сек.
++out; *out = 2;          // Будет выведено: 2 сек.
```

Примеры применения потоковых итераторов приведены в гл. 17.

16.5. Функциональные объекты

Функциональным объектом называют объект с типом класс, в котором определена операция вызова функции (). Ранее, при рассмотрении очередей с приоритетами, мы уже сталкивались с функциональными объектами. Чаще всего они применяются в качестве параметров стандартных алгоритмов для задания пользовательских критериев сравнения объектов или способов их обработки. В тех алгоритмах, где в качестве параметра можно использовать *функциональный объект*, можно применить и *указатель на функцию*.

Стандартная библиотека содержит множество функциональных объектов, предназначенных для ее эффективного использования и расширения. Их описание имеется во включаемом файле <functional>. Имеются и функциональные объекты, возвращающие значение `bool` и называемые *предикатами*. Предикатом называют и обычную функцию, возвращающую значение `bool`.

Базовые классы для функциональных объектов. Стандартная библиотека содержит множество полезных функциональных объектов. Для помощи в их написании и в написании пользовательских функциональных объектов библиотека содержит шаблоны унарной и бинарной функции:

```
namespace std
{
    template < class Arg, class Res >
    struct unary_function
    {
        typedef Arg argument_type;
        typedef Res result_type;
    };

    template < class Arg1, class Arg2, class Res >
    struct binary_function
    {
        typedef Arg1 first_argument_type;
        typedef Arg2 second_argument_type;
```

```
typedef Res result_type;
};
}
```

Назначение этих классов — дать стандартные имена типам аргументов и возвращаемых значений для использования в классах, производных от `unary_function` и `binary_function`.

Адаптером функции называют функцию, которая получает в качестве аргумента функцию, чтобы сконструировать из нее другую функцию. На месте аргумента-функции может находиться также функциональный объект. Стандартная библиотека содержит описания следующих типов адаптеров: связывателей, отрицателей, адаптеров указателей на функции и адаптеров методов.

Арифметические функциональные объекты. В стандартной библиотеке определены шаблоны функциональных объектов для всех арифметических операций языка C++:

```
plus (бинарная: x + y);
minus (бинарная: x - y);
multiplies (бинарная: x * y);
divides (бинарная: x / y);
modulus (бинарная: x % y);
negative (унарная: - x)
```

В качестве примера приведем шаблон объекта `plus`, определенный в стандартном пространстве имен `std` (остальные объекты описываются аналогичным образом):

```
template < class T >
struct plus : binary_function < T, T, T >
{
    T operator()( const T &x, const T &y ) const
    {
        return x + y;
    }
};
```

Примеры применения арифметических функциональных объектов приведены в гл. 17.

Предикаты. В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения и логических операций языка C++:

```
equal_to (бинарная: x == y);
not_equal_to (бинарная: x != y);
greater (бинарная: x > y);
less (бинарная: x < y);
greater_equal (бинарная: x >= y);
less_equal (бинарная: x <= y);
logical_and (бинарная: x && y);
logical_or (бинарная: x || y);
logical_not (унарная: !x)
```

В качестве примера приведем шаблон объекта `equal_to`, определенный в стандартном пространстве имен `std` (остальные объекты описываются аналогичным образом):

```
template < class T >
struct equal_to : binary_function < T, T, bool >
```

```
{
    bool operator()( const T &x, const T &y ) const
    {
        return x == y;
    }
};
```

Программист может описать собственные предикаты для определения критериев сравнения объектов.

Примеры применения предикатов будут рассмотрены далее.

Отрицатели. Отрицатели `not1` и `not2` применяются для получения результата, противоположного унарному и бинарному предикатам соответственно. Например, чтобы получить инверсию предиката `less<int>()`, достаточно записать `not2(less<int>())`. Полученный результат эквивалентен `greater_equal<int>()`. Отрицатели применяются обычно для инвертирования предикатов, заданных пользователем. Объясняется это тем, что для стандартных предикатов библиотека содержит соответствующие им противоположные объекты.

Связыватели. Бинарные предикаты стандартной библиотеки, такие как `less`, полезны и достаточно гибки [6]. Однако оказывается, что самый полезный вид предиката — тот, который сравнивает фиксированный аргумент (часто константу) с элементами контейнера. Чтобы в этом случае использовать тот же самый предикат, требуется связать один из двух его аргументов с фиксированным аргументом (константой). Для этого в стандартной библиотеке используются связыватели `bind1st()` и `bind2nd()`, позволяющие связать с конкретным значением соответственно первый и второй аргумент бинарного предиката стандартной библиотеки.

Связыватели реализованы в стандартной библиотеке как шаблоны функций, принимающих первым параметром функциональный объект `f` с двумя аргументами, а вторым — привязываемое значение `value` [9]. В результате вызова связыватель возвращает функциональный объект, созданный из входного объекта `f` путем подстановки `value` в его первый или второй аргумент.

Для описания типов функциональных объектов, возвращаемых связывателями, в стандартной библиотеке объявлены шаблоны классов `binder1st` и `binder2nd`. Шаблоны связывателей при этом имеют вид:

```
template < class Operation, class Type >
binder1st< Operation > bind1st( const Operation &f, const Type &value );

template < class Operation, class Type >
binder2nd< Operation > bind2nd( const Operation &f, const Type &value );
```

Здесь `Operation` — тип функционального объекта, `Type` — тип привязываемого значения.

Рассмотрим несложный иллюстрирующий пример (листинги 16.1, 16.2).

Листинг 16.1. Файл bind.cpp

```
/*
    Использование связывателей.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <functional>         // Функциональные объекты
#include <algorithm>          // Алгоритмы

using namespace std;         // Используем стандартное
                               // пространство имен

int main( void )             // Возвращает 0 при успехе
{
    int      arr[ 6 ] = { -10, 45, 0, 12, 10, 30 };

    cout << "Целочисленный массив: -10, 45, 0, 12, 10, 30 " << endl
         << "1) содержит " <<
         count_if( arr, arr+6, bind1st( less< int >( ), 10 ) ) <<
         " элемента, значения которых > 10;" << endl << "2) содержит "
         << count_if( arr, arr+6, bind2nd( less< int >( ), 10 ) ) <<
         " элемента, значения которых < 10." << endl;

    cout << endl;

    return 0;
}
```

Листинг 16.2. Результат выполнения программы

```
Целочисленный массив: -10, 45, 0, 12, 10, 30
1) содержит 3 элемента, значения которых > 10;
2) содержит 2 элемента, значения которых < 10.
```

Press any key to continue

В этой простой программе для подсчета количества элементов массива применяется алгоритм `count_if` стандартной библиотеки (см. о нем подробнее в гл. 17). Первыми двумя аргументами в вызове алгоритма `count_if()` должны быть итераторы, определяющие начало и конец обрабатываемой последовательности. В примере при обработке массива вместо итераторов использованы указатели на его элементы. Третий аргумент в вызове алгоритма должен быть бинарной функцией или функциональным объектом (в примере использован функциональный объект).

Адаптеры указателей на функции. Адаптер указателя на функцию позволяет использовать указатель на функцию как аргумент алгоритма [6]. В частности, с его помощью можно применить связыватели к обычному указателю на функцию, которая не является предикатом стандартной библиотеки.

Стандартная библиотека определяет два функциональных объекта: указатель на унарную функцию `pointer_to_unary_function` и указатель на бинарную функцию `pointer_to_binary_function`, а также два метода-адаптера `ptr_fun()`, которые преобразуют переданный им в качестве аргумента указатель на функцию в функциональный объект [9]:

```
template < class Arg, class Result >
class pointer_to_unary_function
    : public unary_function< Arg, Result >
{
public:

    explicit pointer_to_unary_function(
        Result ( *pf )( Arg ) );
    Result operator( )( const Arg x ) const;
};

template < class Arg, class Result >
pointer_to_unary_function< Arg, Result >
    ptr_fun( Result ( *pf )( Arg ) );

template < class Arg1, class Arg2, class Result >
class pointer_to_binary_function
    : public binary_function< Arg1, Arg2, Result >
{
public:

    explicit pointer_to_binary_function(
        Result ( *pf )( Arg1, Arg2 ) );
    Result operator( )( const Arg1 x, const Arg2 y ) const;
};

template < class Arg1, class Arg2, class Result >
pointer_to_binary_function< Arg1, Arg2, Result >
    ptr_fun( Result ( *pf )( Arg1, Arg2 ) );
```

Использование адаптера указателя на функцию иллюстрирует следующий пример (листинги 16.3, 16.4).

Листинг 16.3. Файл adap1.cpp

```

/*
    Использование адаптеров указателей на функции.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <functional>         // Функциональные объекты
#include <algorithm>          // Алгоритмы

using namespace std;         // Используем стандартное
                               // пространство имен

// Структурный тип
struct S
{
    int      x, y;
};

// Сравнение структурных объектов по значению поля "x"
bool lss(
    S      s1,
    S      s2 )
{
    return s1.x < s2.x;
}

int main( void )              // Возвращает 0 при успехе
{
    S      arr[ 4 ] = { {2, 4}, {2, 2}, {3, 1}, {0, 0} };
    S      el = {3, 0};

    cout << "Массив структур: {2, 4}, {2, 2}, {3, 1}, {0, 0} " << endl
         << "содержит " <<
         count_if( arr, arr+4, bind2nd( ptr_fun( lss ), el ) ) <<
         " элемента, значения поля <x> которых меньше 3" << endl;

    cout << endl;

    return 0;
}

```

Листинг 16.4. Результат выполнения программы

Массив структур: {2, 4}, {2, 2}, {3, 1}, {0, 0}
 содержит 3 элемента, значения поля <x> которых меньше 3

Press any key to continue

В приведенном примере функция `lss()` определяет правило сравнения элементов массива структур `arr`. Алгоритм стандартной библиотеки `count_if()` вычисляет количество элементов массива структур `arr`, удовлетворяющих условию, заданному третьим аргументом. Этим аргументом является функциональный объект, созданный связывателем `bind2nd()` из функционального объекта, полученного из функции `lss()` с помощью метода-адаптера `ptr_fun()` и переменной, подставляемой на место второго параметра функции.

Адаптеры методов. Адаптер метода класса позволяет использовать метод класса в качестве аргумента алгоритма [6]. При хранении в контейнерах объектов пользовательских классов часто требуется применить ко всем элементам контейнера один и тот же метод пользовательского класса. Для этого метода следует использовать соответствующий адаптер, который получает метод класса и конструирует из него функциональный объект. В стандартной библиотеке определено несколько адаптеров для методов с различным числом аргументов:

- ❑ `mem_fun()` — вызывает через указатель безаргументный метод, константный безаргументный метод или унарный метод;
- ❑ `mem_fun_ref()` — вызывает по ссылке безаргументный метод, константный безаргументный метод, унарный метод, константный унарный метод или вызывает через указатель унарный константный метод.

Рассмотрим иллюстрирующий пример (листинги 16.5, 16.6).

Листинг 16.5. Файл `adap2.cpp`

```
/*
    Использование адаптеров методов класса.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <vector>             // Векторные объекты
#include <functional>         // Функциональные объекты
#include <algorithm>          // Алгоритмы

using namespace std;        // Используем стандартное
                             // пространство имен

// Пользовательский класс для элементов вектора
class Elem
{
    bool    Value;

public:
    // Конструктор
    Elem( bool V = true )
    {
```

```

        Value = V;
    }

    // Метод, вызываемый для элементов контейнера
    bool GetValue( void )
    {
        return Value;
    }
};

int main( void )           // Возвращает 0 при успехе
{
    // Вектор с элементами пользовательского типа
    vector< Elem >
        arr( 10 );

    cout << "В векторе arr( 10 ) элементов со значением " << endl
         << "поля элемента Value=true: " <<
        count_if( arr.begin( ), arr.end( ),
        mem_fun_ref( &Elem :: GetValue ) ) << endl;

    cout << endl;

    return 0;
}

```

Листинг 16.6. Результат работы программы

```

В векторе arr( 10 ) элементов со значением
поля элемента Value=true: 10

Press any key to continue

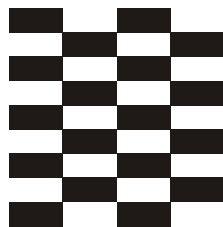
```

16.6. Вопросы для самопроверки

1. Для чего предназначены итераторы?
2. Одинакова ли семантика итератора и указателя?
3. Какие операции допустимы для любого типа итератора?
4. Перечислите категории итераторов, разрешенные для каждой категории операции и области их применения.
5. В каких случаях итератор может быть недействительным?
6. В каких случаях следует использовать константные итераторы?
7. Перечислите итераторные адаптеры и укажите их назначение.
8. Перечислите разновидности функциональных объектов.

Ответы на эти вопросы приведены в *разд. П1.13 приложения 1*.

Глава 17



Алгоритмы

Сами по себе контейнеры большого значения не имеют [6]. Чтобы стать поистине полезным, контейнер должен быть снабжен основными операциями. Стандартная библиотека предоставляет алгоритмы для выполнения большинства распространенных операций, которые требуются пользователю контейнера. Общее число таких алгоритмов около шестидесяти.

Каждый алгоритм реализован в виде шаблона функции или в виде набора шаблонов функций. Благодаря этому алгоритм может работать с различными видами последовательностей и данными разнообразных типов. Для настройки алгоритма на конкретные требования пользователя применяются функциональные объекты, рассмотренные в *разд. 16.5*.

Использование стандартных алгоритмов, как и других средств стандартной библиотеки, избавляет программиста от написания, отладки и документирования циклов обработки последовательностей. Это способствует уменьшению ошибок в программе, сокращает время ее разработки и делает ее более читаемой и компактной.

Объявления стандартных алгоритмов находятся в заголовочном файле `<algorithm>`, а стандартных функциональных объектов — в файле `<functional>`.

Все алгоритмы STL можно разделить на пять категорий.

- ☐ Немодифицирующие операции с последовательностями (не меняют значения элементов последовательности или последовательность в целом, извлекают информацию из последовательности или определяют положение элемента в последовательности).
- ☐ Модифицирующие операции с последовательностями (могут изменять последовательность или значения ее элементов).
- ☐ Алгоритмы сортировки последовательностей.
- ☐ Алгоритмы работы с множествами и пирамидами.
- ☐ Обобщенные численные алгоритмы, объявления которых помещены в заголовочный файл `<numeric>`.

Первые четыре категории стандартных алгоритмов рассматриваются далее.

В качестве параметров алгоритму передаются итераторы, определяющие начало и конец обрабатываемой последовательности. Вид итераторов определяет типы контейнеров, для которых может использоваться данный алгоритм. Например, алгоритм

сортировки `sort()` требует использования итераторов произвольного доступа. Поэтому он не может работать с контейнером `list`. *Алгоритмы не проверяют выход за пределы последовательности!*

Далее при описании параметров шаблонов алгоритмов будут использованы следующие сокращения:

- `In` — итератор для чтения;
- `Out` — итератор для записи;
- `For` — прямой итератор;
- `Bi` — двунаправленный итератор;
- `Ran` — итератор произвольного доступа;
- `Pred` — унарный предикат;
- `BinPred` — бинарный предикат;
- `Comp` — функция сравнения;
- `Op` — унарная операция;
- `BinOp` — бинарная операция.

17.1. Немодифицирующие операции с последовательностями

Алгоритмы этой категории просматривают последовательность, не изменяя ее. Они используются для получения информации о последовательности или для определения положения элемента.

Алгоритм `for_each()`. Алгоритм вызывает для каждого элемента последовательности заданную функцию `f`:

```
template < class In, class Function >
    Function for_each( In first, In last, Function f );
```

Одно из обычных применений этого алгоритма — извлечение информации из элементов последовательности. Пример использования данного алгоритма и других алгоритмов этой группы приводится далее.

Алгоритмы `find()`, `find_if()`, `find_first_of()`, `find_end()` и `adjacent_find()`. Алгоритмы семейства `find` просматривают последовательность или две последовательности в поисках значения или соответствия заданному предикату.

Алгоритм `find()` выполняет поиск в последовательности заданного значения `value`:

```
template < class In, class T >
    In find( In first, In last, const T &value );
```

Алгоритм возвращает итератор на самое левое найденное значение в случае успешного поиска и на конец последовательности — в ином случае.

Алгоритм `find_if()` выполняет поиск в последовательности значения, соответствующего заданному предикату `pred`:

```
template < class In, class Pred >
    In find_if( In first, In last, Pred pred );
```

Алгоритм возвращает итератор на самое левое найденное значение в случае успешного поиска и на конец последовательности — в ином случае.

Алгоритм `find_first_of()` находит первое вхождение в первую последовательность элемента из второй последовательности:

```
template < class For1, class For2 >
    For1 find_first_of( For1 first1, For1 last1, For2 first2,
                       For2 last2 );
template < class For1, class For2, class BinPred >
    For1 find_first_of( For1 first1, For1 last1, For2 first2, For2 last2,
                       BinPred pred );
```

Границы последовательностей задаются с помощью итераторов. Первая форма алгоритма ищет вхождение любого элемента, а вторая — элемента, для которого выполняется бинарный предикат, анализирующий элементы первой и второй последовательностей. В случае неудачного поиска возвращается `last1`.

Алгоритм `find_end()` находит последнее вхождение в первую последовательность второй последовательности (с анализом или без анализа предиката) и возвращает итератор на первый совпадающий элемент:

```
template < class For1, class For2 >
    For1 find_end( For1 first1, For1 last1, For2 first2, For2 last2 );
template < class For1, class For2, class BinPred >
    For1 find_end( For1 first1, For1 last1, For2 first2, For2 last2,
                  BinPred pred );
```

В случае неудачного поиска возвращается `last1`.

Алгоритм `adjacent_find()` выполняет поиск пары соседних значений:

```
template < class For >
    For adjacent_find( For first, For last );
template < class For, class BinPred >
    For adjacent_find( For first, For last, BinPred pred );
```

Первая форма алгоритма находит в последовательном контейнере пару одинаковых соседних значений и возвращает итератор на первое из них. Вторая форма находит соседние элементы, удовлетворяющие условию, заданному предикатом `pred` в виде функции или функционального объекта. В случае неудачного поиска возвращается `last`.

Алгоритмы `count()` и `count_if()`. Алгоритм `count()` выполняет подсчет количества вхождений в последовательность заданного значения `value`:

```
template < class In, class T >
    typename iterator_traits < In >::difference_type
    count( In first, In last, const T &value );
```

Результат имеет тип разности между двумя итераторами `difference_type`.

Алгоритм `count_if()` выполняет подсчет количества элементов последовательности, удовлетворяющих условию, заданному предикатом `pred` в виде функции или функционального объекта:

```
template < class In, class Pred >
    typename iterator_traits < In >::difference_type
    count_if( In first, In last, Pred pred );
```

Примеры использования алгоритма `count_if()` приведены в разд. 16.5.

Алгоритм *mismatch()*. Алгоритм ищет первую пару несовпадающих значений двух последовательностей и возвращает итераторы на эту пару:

```
template < class In1, class In2 >
    pair< In1, In2 > mismatch( In1 first1, In1 last1, In2 first2 );
template < class In1, class In2, class BinPred >
    pair< In1, In2 > mismatch( In1 first1, In1 last1, In2 first2,
                             BinPred pred );
```

Длина второй последовательности считается большей или равной длине первой. С помощью второй формы алгоритма можно задать предикат, определяющий, что считать несовпадением.

Алгоритм *equal()*. Алгоритм возвращает `true`, если элементы двух последовательностей попарно равны:

```
template < class In1, class In2 >
    bool equal( In1 first1, In1 last1, In2 first2 );
template < class In1, class In2, class BinPred >
    bool equal( In1 first1, In1 last1, In2 first2, BinPred pred );
```

Длина второй последовательности считается большей или равной длине первой. С помощью второй формы алгоритма можно задать предикат, определяющий, что считать равенством.

Алгоритмы *search()* и *search_n()*. Алгоритм `search()` находит первое вхождение в первую последовательность второй последовательности (с анализом предиката или без), и возвращает итератор на первый совпадающий элемент:

```
template < class For1, class For2 >
    For1 search( For1 first1, For1 last1, For2 first2, For2 last2 );
template < class For1, class For2, class BinPred >
    For1 search( For1 first1, For1 last1, For2 first2, For2 last2,
                BinPred pred );
```

В случае неудачного поиска возвращается `last1`.

Алгоритм `search_n()` находит в последовательности подпоследовательность, содержащую, по крайней мере, `count` значений `value` (с анализом предиката или без), и возвращает итератор на первый совпадающий элемент:

```
template < class For, class Size, class T >
    For search_n( For first, For last, Size count, const T &value );
template < class For, class Size, class T, class BinPred >
    For search_n( For first, For last, Size count, const T &value,
                BinPred pred );
```

Рассмотрим иллюстрирующий пример (листинги 17.1, 17.2).

Листинг 17.1. Файл `alg_nomodif.cpp`

/*

Использование немодифицирующих алгоритмов для последовательностей.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```
*/

#include <iostream>          // Поточковый ввод-вывод
#include <vector>             // Векторы
#include <functional>        // Функциональные объекты
#include <algorithm>         // Алгоритмы

using namespace             // Используем стандартное
    std;                   // пространство имен

// Размер вектора v
const int    VECTOR_SIZE = 4 ;
// Размер вектора v1
const int    VECTOR_SIZE1 = 2 ;

// Функция, вызываемая для каждого элемента v
void show(
    int      a )
{
    cout << a << " ";

    return;
}

// Функция для поиска первого нечетного элемента v
bool odd(          // Возвращает true, если нечетное
    int          a )
{
    if( a%2 )
        return true;
    return false;
}

int main( void )      // Возвращает 0 при успехе
{
    // Векторы с элементами целого типа
    vector< int >
        v( VECTOR_SIZE ), v1( VECTOR_SIZE1 );

    // Инициализация элементов v и v1
    for( unsigned i = 0; i < VECTOR_SIZE; i++ )
        v[ i ] = i + 1;
    for( i = 0; i < VECTOR_SIZE1; i++ )
        v1[ i ] = i + 3;

    // Печать значения каждого элемента v и v1 с использованием алгоритма
    //  for_each
```

```

cout << "v: ";
for_each( v.begin( ), v.end( ), show );
cout << endl << "v1: ";
for_each( v1.begin( ), v1.end( ), show );

int          value = 3;
// Поиск в векторе v элемента со значением value
vector< int >::iterator
    it = find( v.begin( ), v.end( ), value );
if( it == v.end( ) )
{
    cout << endl << endl << "В v элемент со значением " << value
        << " не обнаружен";
}
else
{
    cout << endl << endl << "В v обнаружен элемент со значением "
        << *it;
}

// Поиск в векторе v первого элемента с нечетным значением
it = find_if( v.begin( ), v.end( ), odd );
if( it == v.end( ) )
{
    cout << endl << "В v нечетный элемент не обнаружен";
}
else
{
    cout << endl << "В v обнаружен первым нечетный элемент со "
        "значением " << *it;
}

// Поиск в v первой пары соседних элементов, у которой левый элемент
// меньше правого
it = adjacent_find( v.begin( ), v.end( ), less< int >( ) );
if( it == v.end( ) )
{
    cout << endl << "В v пара соседних элементов меньше-больше не"
        " обнаружена";
}
else
{
    cout << endl << "В v обнаружена первая пара соседних элементов"
        " \n меньше-больше (" << *it << ", " << *( it+1 ) << ")";
}

```

```
// Поиск последнего вхождения v1 в v
it = find_end( v.begin( ), v.end( ), v1.begin( ), v1.end( ) );
if( it == v.end( ) )
{
    cout << endl << "В v v1 не содержится";
}
else
{
    cout << endl << "В v содержится v1, первый совпадающий элемент"
        " v равен " << *it;
}

// Поиск первого вхождения в v элемента v1
it = find_first_of( v.begin( ), v.end( ), v1.begin( ), v1.end( ) );
if( it == v.end( ) )
{
    cout << endl << "В v элементы из v1 не содержатся";
}
else
{
    cout << endl << "В v обнаружено первое вхождение элемента из"
        " v1: " << *it;
}

// Элементы v просматриваются по порядку, и фиксируется очередной
// элемент. Для него производится сравнение в соответствии с
// предикатом со всеми элементами v1, просматриваемыми по порядку.
// При выполнении условия возвращается итератор зафиксированного
// элемента v. Иначе - аналогично обрабатываются последующие
// элементы v
it = find_first_of( v.begin( ), v.end( ), v1.begin( ), v1.end( ),
    greater< int >( ) );
if( it == v.end( ) )
{
    cout << endl << "В v элементы, большие к.-л. элементов v1, "
        "\nне содержатся";
}
else
{
    cout << endl << "В v обнаружено первое вхождение элемента,"
        " большего \n к.-л. элемента v1: " << *it;
}

// Подсчет в векторе v количества элементов со значением value
cout << endl << "В v имеется " << count( v.begin( ), v.end( ),
```

```

    value ) << " элементов со значением " << value;

// Проверка v и v1 на несовпадение
// Тип итератора
typedef vector< int >::iterator IntVecIt;
// Определение пары
pair < IntVecIt, IntVecIt >
    p = mismatch( v1.begin( ), v1.end( ), v.begin( ) );
cout << endl << "Первая пара несовпадающих элементов " << endl
    << "v1: " << *p.first << ", v: " << *p.second;

// Проверка v и v1 на совпадение
if( equal( v.begin( ), v.end( ), v1.begin( ) ) )
{
    cout << endl << "Последовательности совпадают";
}
else
{
    cout << endl << "Последовательности не совпадают";
}

// Проверка вхождения v1 в v
it = search( v.begin( ), v.end( ), v1.begin( ), v1.end( ) );
if( it == v.end( ) )
{
    cout << endl << "v1 не входит в v";
}
else
{
    cout << endl << "v1 входит в v - первый совпадающий элемент: "
        << *it;
}

cout << endl << endl;

return 0;
}

```

Листинг 17.2. Результат выполнения программы

v: 1 2 3 4

v1: 3 4

В v обнаружен элемент со значением 3

В v обнаружен первым нечетный элемент со значением 1

В v обнаружена первая пара соседних элементов

```

меньше-больше (1, 2)
В v содержится v1, первый совпадающий элемент v равен 3
В v обнаружено первое вхождение элемента из v1: 3
В v обнаружено первое вхождение элемента, большего
к.-л. элемента v1: 4
В v имеется 1 элементов со значением 3
Первая пара несовпадающих элементов
v1: 3, v: 1
Последовательности не совпадают
v1 входит в v - первый совпадающий элемент: 3

Press any key to continue

```

17.2. Модифицирующие операции с последовательностями

Алгоритмы этой категории изменяют последовательность, с которой они работают. Они выполняют копирование, удаление, замену и изменение порядка следования элементов последовательности.

Алгоритмы `copy()` и `copy_backward()`. Алгоритм `copy()` выполняет копирование, начиная с первого элемента последовательности, границы которой задаются итераторами `first` и `last`, в выходную последовательность, для которой задается итератор начала `result`:

```

template < class In, class Out >
    Out copy( In first, In last, Out result );

```

Алгоритм `copy_backward()` выполняет копирование, начиная с последнего элемента заданной последовательности. Его третий параметр должен указывать на элемент, следующий за последним элементом приемника — его значение уменьшается на шаг перед операцией копирования каждого элемента:

```

template < class Bi1, class Bi2 >
    Bi2 copy_backward( Bi1 first, Bi1 last, Bi2 result);

```

Последовательности могут перекрываться. При копировании нужно следить за тем, чтобы не выйти за границы выходной последовательности.

Поясним сказанное содержательным примером [9] (листинги 17.3, 17.4).

Листинг 17.3. Файл `alg_modif_copy.cpp`

```

/*
    Копирование и вывод последовательности.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <algorithm>         // Алгоритмы

using namespace std        // Используем стандартное

```

```
std;          // пространство имен

int main( void )          // Возвращает 0 при успехе
{
    // Копирование последовательности
    int      b[ 4 ], a[ 5 ] = {1, 2, 3, 4, 5};
    // Копируем с начала часть массива <a> в массив <b>
    copy( a+1, a+5, b );
    cout << "b: ";
    for( int i=0; i<4; i++ )
    {
        cout << b[ i ] << " ";
    }
    cout << endl;

    // Копируем слева-направо часть массива <a> в его начало
    copy( a+1, a+5, a );
    cout << "a: ";
    for( i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;

    // Копируем справа-налево массив <b> в его начало - обратите
    // внимание, что последовательности перекрываются
    copy_backward( b, b+3, b+4 );
    cout << "b: ";
    for( i=0; i<4; i++ )
    {
        cout << b[ i ] << " ";
    }
    cout << endl;

    // Вывод последовательности - третьим параметром м. б. потоковый
    // итератор для вывода
    cout << "a: ";
    copy( a, a+5, ostream_iterator< int >( cout, " " ) );
    cout << endl;

    cout << endl;
    return 0;
}
```

Листинг 17.4. Результат работы программы

```

b: 2 3 4 5
a: 2 3 4 5 5
b: 2 2 3 4
a: 2 3 4 5 5

```

Press any key to continue

Алгоритмы *fill()* и *fill_n()*. Алгоритм *fill()* выполняет замену всех элементов последовательности, определенной с помощью итераторов *first* и *last*, заданным значением *value*. Алгоритм *fill_n()* выполняет замену *n* элементов последовательности заданным значением *value*:

```

template < class For, class T >
    void fill( For first, For last, const T &value );

template < class Out, class Size, class T >
    void fill_n( Out first, Size n, const T &value );

```

Поясним сказанное содержательным примером [9] (листинги 17.5, 17.6).

Листинг 17.5. Файл `alg_modif_fill.cpp`

```

/*
    Заполнение последовательности заданным значением.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <algorithm>          // Алгоритмы

using namespace std;         // Используем стандартное
                               // пространство имен

int main( void )             // Возвращает 0 при успехе
{
    int a[ 5 ];
    // Заполняем массив <a> значением 1
    fill( a, a+5, 1 );
    cout << "a: ";
    for( int i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;
    // !!! Для списков мы не можем пользоваться выражением
    // fill( a, a+5, 1 );, так как операция сложения для итераторов

```

```

// списка не определена. В подобном случае можно воспользоваться
// алгоритмом fill_n( )

// Заполняем третий и четвертый элементы массива <a> значением 0
fill_n( a+2, 2, 0 );
cout << "a: ";
for( i=0; i<5; i++ )
{
    cout << a[ i ] << " ";
}
cout << endl;

cout << endl;
return 0;
}

```

Листинг 17.6. Результат работы программы

```

a: 1 1 1 1 1
a: 1 1 0 0 1

```

Press any key to continue

Алгоритмы *generate()* и *generate_n()*. Алгоритм *generate()* выполняет замену всех элементов последовательности значениями, вычисленными с помощью функции или функционального объекта, заданного третьим параметром. Алгоритм *generate_n()* выполняет замену *n* элементов последовательности:

```

template < class For, class Generator >
    void generate( For first, For last, Generator gen );
template < class Out, class Size, class Generator >
    void generate_n( Out first, Size n, Generator gen );

```

Поясним сказанное содержательным примером [9] (листинги 17.7, 17.8).

Листинг 17.7. Файл `alg_modif_gen.cpp`

```

/*
    Заполнение последовательности значениями, заданными с помощью функции.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <algorithm>         // Алгоритмы

using namespace std;        // Используем стандартное
                             // пространство имен

```

```
// Функция, вычисляющая значения для заполнения последовательности
int f( void )
{
    static int i = 1;
    return ( ++i ) * 3;
}

int main( void )          // Возвращает 0 при успехе
{
    int          a[ 5 ];
    // Заполняем массив <a> значениями, возвращаемыми функцией f( )
    generate( a, a+5, f );
    // !!! Обратите внимание, что в качестве третьего аргумента в вызове
    // функции следует указывать только имя функции без последующих
    // круглых скобок
    cout << "a: ";
    for( int i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;

    cout << endl;
    return 0;
}
```

Листинг 17.8. Результат работы программы

```
a: 6 9 12 15 18
```

```
Press any key to continue
```

Алгоритмы *iter_swap()*, *swap()* и *swap_ranges()*. Алгоритм *iter_swap()* выполняет обмен местами двух элементов, заданных итераторами:

```
template < class For1, class For2 >
void iter_swap( For1 a, For2 b );
```

Алгоритм *swap()* выполняет обмен местами двух заданных элементов:

```
template < class T >
void swap( T &a, T &b );
```

Алгоритм *swap_ranges()* выполняет обмен местами элементов в двух указанных диапазонах (для второго диапазона задано только его начало):

```
template < class For1, class For2 >
void swap_ranges( For1 first1, For1 last1, For2 first2 );
```

Поясним использование перечисленных алгоритмов примером (листинги 17.9, 17.10).

Листинг 17.9. Файл alg_modif_swap.cpp

```
/*
    Обмен местами элементов.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <algorithm>         // Алгоритмы

using namespace            // Используем стандартное
    std;                  // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    int        a[ 5 ] = { 1, 2, 3, 4, 5 },
               b[ 5 ] = { 11, 12, 13, 14, 15 };
    // Меняем местами два элемента, заданных итераторами
    iter_swap( a, b+4 );
    cout << "a: ";
    for( int i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl << "b: ";
    for( i=0; i<5; i++ )
    {
        cout << b[ i ] << " ";
    }
    cout << endl;
    // !!! А можно и так:
    iter_swap( a, a+4 );
    cout << "a: ";
    for( i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;

    // Обмен местами двух элементов
    float      f1 = 1.5, f2 = -1.5;
    swap( f1, f2 );
    cout << "f1 = " << f1 << ", f2 = " << f2 << endl;

    // Обмен местами элементов двух последовательностей, в двух указанных
    // диапазонах
```

```

swap_ranges( a, a+2, b+3 );
cout << endl << "a: ";
for( i=0; i<5; i++ )
{
    cout << a[ i ] << " ";
}
cout << endl << "b: ";
for( i=0; i<5; i++ )
{
    cout << b[ i ] << " ";
}
cout << endl;
// !!! А можно и так:
swap_ranges( a, a+2, a+3 );
cout << endl << "a: ";
for( i=0; i<5; i++ )
{
    cout << a[ i ] << " ";
}
cout << endl;

cout << endl;
return 0;
}

```

Листинг 17.10. Результат работы программы

```

a: 15 2 3 4 5
b: 11 12 13 14 1
a: 5 2 3 4 15
f1 = -1.5, f2 = 1.5

```

```

a: 14 1 3 4 15
b: 11 12 13 5 2

```

```

a: 4 15 3 14 1

```

```

Press any key to continue

```

Алгоритм *random_shuffle()* выполняет перетасовку элементов последовательности в соответствии со случайным равномерным распределением. Третьим параметром алгоритма можно задать генератор случайных чисел, который может быть функцией или функциональным объектом, получающим аргумент *n* типа *int* и возвращающим целое число в диапазоне *last-first-1*:

```

template < class Ran >
void random_shuffle( Ran first, Ran last );

```

```
template < class Ran, class RanNumGen >
    void random_shuffle( Ran first, Ran last, RanNumGen &r );
```

Поясним использование алгоритма примером (листинги 17.11, 17.12).

Листинг 17.11. Файл `alg_modif_rand.cpp`

```
/*
    Перетасовка элементов последовательности в соответствии со случайным
    равномерным распределением.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <algorithm>         // Алгоритмы

using namespace            // Используем стандартное
    std;                  // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    int        a[ 5 ] = { 1, 2, 3, 4, 5 };

    // Перетасовываем элементы массива <a>
    random_shuffle( a, a+5 );
    cout << "a: ";
    for( int i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;

    // Перетасуем часть элементов массива
    random_shuffle( a, a+4 );
    cout << "a: ";
    for( i=0; i<5; i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;

    cout << endl;
    return 0;
}
```

Листинг 17.12. Результат работы программы

```
a: 5 4 1 3 2
a: 3 5 4 1 2
```

Press any key to continue

Алгоритмы *remove()*, *remove_if()*, *remove_copy()* и *remove_copy_if()* выполняют удаление из последовательности элементов с заданным значением *value* или по предикату *pred*. При этом оставшиеся элементы перемещаются в начало последовательности с сохранением их относительного порядка. Алгоритмы возвращают границу их размещения. Элементы, расположенные начиная с границы размещения, не удаляются, сохраняют свое положение в последовательности и значения, а размер последовательности не изменяется. Формы алгоритмов, содержащие слово *copy*, перед обработкой копируют последовательность на место, заданное итератором *Out*, и обрабатывают копию последовательности:

```
template < class For, class T >
    For remove( For first, For last, const T &value );
template < class For, class Pred >
    For remove_if( For first, For last, Pred pred );
template < class In, class Out, class T >
    Out remove_copy( In first, In last, Out result, const T &value );
template < class In, class Out, class Pred >
    Out remove_copy_if( In first, In last, Out result, Pred pred );
```

Поясним использование указанных алгоритмов содержательным примером (листинги 17.13, 17.14).

Листинг 17.13. Файл `alg_modif_remove.cpp`

```
/*
    Удаление из последовательности элементов с заданным значением.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <algorithm>          // Алгоритмы
#include <vector>              // Векторы

using namespace std;         // Используем стандартное
                               // пространство имен

// Функция-предикат для выбора элементов вектора
bool great_1(                 // Возвращает true при x>1
    int x )                   // Проверяемое значение
{
    return x>1;
}
```

```

}

int main( void )           // Возвращает 0 при успехе
{
    // Создаем векторы и инициализируем их (вектор <b> нулями)
    vector< int >
        a, b( 10 );
    for( int i=0; i<5; i++ )
        a.push_back( i );
    for( i=0; i<5; i++ )
        a.push_back( i );

    cout << "a: ";
    for( i=0; i<a.size( ); i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;

    cout << "b: ";
    for( i=0; i<b.size( ); i++ )
    {
        cout << b[ i ] << " ";
    }
    cout << endl << endl;

    // Удаляем элементы вектора со значением 2: на первый элемент,
    // который сохранил свое положение в векторе, указывает итератор
    // <p> (граница размещения)
    vector< int >::iterator
        p = remove( a.begin( ), a.end( ), 2 );
    cout << " После удаления элементов со значением 2 состояние <a>:"
        << endl << "( remove )" << endl;
    for( i=0; i<a.size( ); i++ )
    {
        cout << a[ i ] << " ";
    }
    cout << endl;
    cout << "Граница размещения *p= " << *p << endl;
    cout << " *(--p)= " << *(--p) << endl;
    cout << " *(--p)= " << *(--p) << endl;
    cout << " *(--p)= " << *(--p) << endl;

    // Копируем вектор <a> в вектор <b> и в копии удаляем элементы со
    // значением 3: на первый элемент, который сохранил свое положение
    // в векторе, указывает итератор <p> (граница размещения)

```

```

p = remove_copy( a.begin( ), a.end( ), b.begin( ), 3 );
cout << endl << " После удаления элементов со значением 3 состояние"
    " <a>:" << endl << "( remove_copy )" << endl;
for( i=0; i<a.size( ); i++ )
{
    cout << a[ i ] << " ";
}
cout << endl;
cout << " После удаления элементов со значением 3 состояние <b>:"
    << endl;
for( i=0; i<b.size( ); i++ )
{
    cout << b[ i ] << " ";
}
cout << endl;
cout << "Граница размещения *p= " << *p << endl;
cout << " *(--p)= " << *(--p) << endl;
cout << " *(--p)= " << *(--p) << endl;
cout << " *(--p)= " << *(--p) << endl;

// Из вектора <a> удаляем элементы, большие 1
p = remove_if( a.begin( ), a.end( ), great_1 );
a.erase( p, a.end( ) );
cout << endl << " После удаления элементов со значениями > 1"
    " состояние <a>:" << endl << "( remove_if + erase )" << endl;
for( i=0; i<a.size( ); i++ )
{
    cout << a[ i ] << " ";
}
cout << endl;

cout << endl;
return 0;
}

```

Листинг 17.14. Результат работы программы

```

a: 0 1 2 3 4 0 1 2 3 4
b: 0 0 0 0 0 0 0 0 0 0

```

После удаления элементов со значением 2 состояние <a>:

(remove)

```
0 1 3 4 0 1 3 4 3 4
```

Граница размещения *p= 3

*(--p)= 4

*(--p)= 3

```
*(--p)= 1
```

После удаления элементов со значением 3 состояние <a>:

```
( remove_copy )
0 1 3 4 0 1 3 4 3 4
```

После удаления элементов со значением 3 состояние :

```
0 1 4 0 1 4 4 0 0 0
```

Граница размещения *p= 0

```
*(--p)= 4
```

```
*(--p)= 4
```

```
*(--p)= 1
```

После удаления элементов со значениями > 1 состояние <a>:

```
( remove_if + erase )
0 1 0 1
```

Press any key to continue

Алгоритмы *replace()*, *replace_if()*, *replace_copy()* и *replace_copy_if()* выполняют замену элементов с заданным значением *old_value* или в соответствии с предикатом *pred* на новое значение *new_value*. Формы алгоритмов, содержащие слово *copy*, перед обработкой копируют последовательность на место, заданное итератором *out*, и обрабатывают копию последовательности:

```
template < class For, class T >
    void replace( For first, For last, const T &old_value,
                  const T &new_value );

template < class For, class Pred, class T >
    void replace_if( For first, For last, Pred pred,
                    const T &new_value );

template < class In, class Out, class T >
    Out replace_copy( In first, In last, Out result,
                     const T &old_value, const T &new_value );

template < class In, class Out, class Pred, class T >
    Out replace_copy_if( In first, In last, Out result, Pred pred,
                        const T &new_value );
```

Поясним использование указанных алгоритмов содержательным примером, в котором выполняется копирование вектора *v1* в новый вектор *v2* с заменой всех элементов со значениями больше 30 и меньше 70 на значение 5 (листинги 17.15, 17.16).

Листинг 17.15. Файл `alg_modif_replace.cpp`

```
/*
```

Замена элементов последовательности.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```
*/
```

```

#include <iostream>           // Поточковый ввод-вывод
#include <algorithm>          // Алгоритмы
#include <vector>              // Векторы

using namespace              // Используем стандартное
    std;                     // пространство имен

// Функция-предикат для выбора элементов вектора
bool In_30_70(               // Возвращает true при 30 < x < 50
    int x )                  // Проверяемое значение
{
    return x>30 && x<70;
}

int main( void )             // Возвращает 0 при успехе
{
    // Создаем векторы и инициализируем их (вектор v2 нулями)
    vector< int >
        v1, v2( 9 );
    for( int i=1; i<10; i++ )
        v1.push_back( i*10 );
    cout << "v1: ";
    for( i=0; i<v1.size( ); i++ )
    {
        cout << v1[ i ] << " ";
    }
    cout << endl;
    cout << "v2: ";
    for( i=0; i<v2.size( ); i++ )
    {
        cout << v2[ i ] << " ";
    }
    cout << endl << endl;

    // Копируем вектор v1 в вектор v2 с заменой значений больше 30 и
    // меньше 50 на значение 5
    replace_copy_if( v1.begin( ), v1.end( ), v2.begin( ), In_30_70, 5 );
    cout << " После копирования вектора v1 в вектор v2\n"
        "с заменой значений больше 30 и меньше 70 на \n"
        "значение 5 ( replace_copy_if )" << endl << "v1:";
    for( i=0; i<v1.size( ); i++ )
    {
        cout << v1[ i ] << " ";
    }
    cout << endl;
}

```

```

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

cout << endl;
return 0;
}

```

Листинг 17.16. Результат работы программы

```

v1: 10 20 30 40 50 60 70 80 90
v2: 0 0 0 0 0 0 0 0 0
После копирования вектора v1 в вектор v2
с заменой значений больше 30 и меньше 70 на
значение 5 ( replace_copy_if )
v1:10 20 30 40 50 60 70 80 90
v2:10 20 30 5 5 5 70 80 90

Press any key to continue

```

Алгоритмы `reverse()` и `reverse_copy()`. Алгоритм `reverse()` изменяет порядок следования элементов последовательности на противоположный, а алгоритм `reverse_copy()` выполняет копирование исходной последовательности в результирующую в обратном порядке:

```

template < class Bi >
    void reverse( Bi first, Bi last );
template < class Bi, class Out >
    Out reverse_copy( Bi first, For last, Out result );

```

Данные алгоритмы просты и не требуют дополнительных пояснений.

Алгоритмы `rotate()` и `rotate_copy()`. Алгоритм `rotate()` выполняет циклическое перемещение элементов последовательности, а алгоритм `rotate_copy()` — копии последовательности:

```

template < class For >
    void rotate( For first, For middle, For last );
template < class For, class Out >
    Out rotate_copy( For first, For middle, For last, Out result );

```

Перемещение выполняется до тех пор, пока элемент, на который указывает второй параметр `middle`, не станет первым в последовательности. Первый и третий параметры задают начало и конец обрабатываемой последовательности. Например, для целочисленной последовательности

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

ВЫЗОВ

```
rotate( a, a+2, a+5 );
```

приведет к тому, что элементы будут следовать в порядке 3 4 5 1 2. Если после этого вызвать

```
rotate( a, a+3, a+5 );
```

то массив примет первоначальный вид.

Алгоритм *transform()* выполняет заданную операцию над каждым элементом последовательности. Первая форма алгоритма выполняет унарную операцию, заданную функцией или функциональным объектом *op*, и помещает результат в место, заданное итератором *result*:

```
template < class In, class Out, class Op >
    Out transform( In first, In last, Out result, Op op );
```

Вторая форма алгоритма выполняет бинарную операцию над парой соответствующих элементов двух последовательностей и помещает результат в место, заданное итератором *result*:

```
template < class In1, class In2, class Out, class BinOper >
    Out transform( In1 first1, In1 last1, In2 first2, Out result,
                  BinOper bin_op );
```

Поясним использование указанного алгоритма содержательным примером [9], в котором первый вызов *transform()* выполняет преобразование массива *a* по формуле $a_i = a_i^2 - b_i^2$, а второй вызов меняет знак у элементов массива *b* с помощью стандартного функционального объекта *negate* (листинги 17.17, 17.18).

Листинг 17.17. Файл `alg_modif_transform.cpp`

```
/*
    Операция над элементами последовательности или над парами элементов двух последовательностей.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <algorithm>          // Алгоритмы
#include <functional>         // Функциональные объекты

using namespace std;        // Используем стандартное
                             // пространство имен

// Функциональный объект пользователя, выполняющий обработку пар
// элементов двух последовательностей
struct conv : binary_function < double, double, double >
{
    double operator( ) ( double x, double y ) const
    {
        return x*x - y*y;
    }
}
```

```

};

int main( void )           // Возвращает 0 при успехе
{
    const int  sz = 5;      // Размер последовательностей
    double     a[ sz ] = { 5, 4, 3, 2, 1 },
               b[ sz ] = { -1, 2, -3, 4, -5 };

    // Печать состояния исходных последовательностей
    cout << "a: ";
    for( int i=0; i<sz; i++ )
        cout << a[ i ] << " ";
    cout << endl << "b: ";
    for( i=0; i<sz; i++ )
        cout << b[ i ] << " ";
    cout << endl << endl;

    transform( a, a+sz, b, a, conv( ) );
    cout << "После  transform( a, a+sz, b, a, conv( ) );" << endl <<
        "a: ";
    for( i=0; i<sz; i++ )
        cout << a[ i ] << " ";

    transform( b, b+sz, b, negate < double >( ) );
    cout << "\n\nПосле transform( b, b+sz, b, negate < double >( ) );"
        << endl << "b: ";
    for( i=0; i<sz; i++ )
        cout << b[ i ] << " ";
    cout << endl << endl;

    return 0;
}

```

Листинг 17.18. Результат работы программы

```

a: 5 4 3 2 1
b: -1 2 -3 4 -5

```

```

После  transform( a, a+sz, b, a, conv( ) );
a: 24 12 0 -12 -24

```

```

После transform( b, b+sz, b, negate < double >( ) );
b: 1 -2 3 -4 5

```

Press any key to continue

Алгоритмы *unique()* и *unique_copy()*. Алгоритм `unique()` выполняет удаление из последовательности соседних элементов, равных друг другу или удовлетворяющих критерию, заданному с помощью бинарного предиката `pred`. Из нескольких подряд идущих повторяющихся элементов последовательности остается только один элемент. Алгоритм возвращает итератор на новый логический конец данных:

```
template < class For >
    For unique( For first, For last );
template < class For, class BinPred >
    For unique( For first, For last, BinPred pred );
```

Алгоритм `unique_copy()` выполняет те же действия с копией последовательности:

```
template < class In, class Out >
    Out unique_copy( In first, In last, Out result );
template < class In, class Out, class BinPred >
    Out unique_copy( In first, In last, Out result, BinPred pred );
```

Проиллюстрируем использование этих алгоритмов несложным примером (листинги 17.19, 17.20).

Листинг 17.19. Файл `alg_modif_unique.cpp`

```
/*
    Удаление из последовательности равных друг другу соседних элементов.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <algorithm>         // Алгоритмы

using namespace            // Используем стандартное
    std;                  // пространство имен

int main( void )           // Возвращает 0 при успехе
{
    const int  sz = 9;     // Размер последовательностей
    double     a[ sz ] = { 5, 5, 5, 4, 3, 3, 2, 1, 5 },
               b[ sz ],
               c[ sz ] = { 5, 5, 5, 4, 3, 3, 2, 1, 5 };

    // Печать состояния исходных последовательностей
    cout << "Последовательности <a> и <c>: ";
    for( int i=0; i<sz; i++ )
        cout << a[ i ] << " ";
    cout << endl << "Их размер sz = " << sz << endl;

    // Удаление из последовательности повторяющихся элементов
    double     *ple = unique( a, a+sz );
    cout << endl << "После выполнения unique( a, a+sz );" << endl <<
```

```

    "a: ";
    for( double *p=a; p<ple; p++ )
        cout << *p << " ";
    cout << endl << endl;

    // Удаление из копии последовательности повторяющихся элементов
    ple = unique_copy( c, c+sz, b );
    cout << "После unique_copy( c, c+sz, b );" << endl << "b: ";
    for( p=b; p<ple; p++ )
        cout << *p << " ";
    cout << endl;

    cout << endl;
    return 0;
}

```

Листинг 17.20. Результат работы программы

Последовательности <a> и <c>: 5 5 5 4 3 3 2 1 5

Их размер sz = 9

После выполнения unique(a, a+sz);

a: 5 4 3 2 1 5

После unique_copy(c, c+sz, b);

b: 5 4 3 2 1 5

Press any key to continue

17.3. Алгоритмы, связанные с сортировкой

В состав этой группы входят следующие алгоритмы.

Алгоритмы `sort()` и `stable_sort()`. Алгоритм `sort()` выполняет эффективную сортировку последовательности за время, пропорциональное $N \cdot \log_2 N$, где N — размер последовательности. Для сохранения порядка следования одинаковых элементов следует применять алгоритм `stable_sort()`, время работы которого пропорционально $N \cdot (\log_2 N)^2$.

```

template < class Ran >
    void sort( Ran first, Ran last );
template < class Ran, class Compare >
    void sort( Ran first, Ran last, Compare comp );
template < class Ran >
    void stable_sort( Ran first, Ran last );
template < class Ran, class Compare >
    void stable_sort( Ran first, Ran last, Compare comp );

```

Этим алгоритмам требуются итераторы произвольного доступа.

Алгоритм *binary_search()* выполняет двоичный поиск значения `val` в *отсортированной* последовательности, заданной итераторами `first` и `last`. Возвращает `true`, если заданное значение в последовательности найдено, или `false` в ином случае. Двоичным поиск называется потому, что выполняется путем последовательного деления интервала пополам.

```
template < class For, class T >
    bool binary_search( For first, For last, const T &val );
template < class For, class T, class Compare >
    bool binary_search( For first, For last, const T &val,
                       Compare comp );
```

Алгоритм *equal_range()* находит границы последовательности элементов, в любое место которой можно вставить заданное значение `value` без нарушения порядка. Последовательность должна быть *отсортирована*. При задании функционального объекта алгоритм находит границы, в пределах которых для каждого значения итератора `k` из этих границ выполняется условие `comp(*k, value) == false` && `comp(value, *k) == false`.

```
template < class For, class T >
    pair< For, For > equal_range( For first, For last, const T &value );
template < class For, class T, class Compare >
    pair< For, For > equal_range( For first, For last, const T &value,
                               Compare comp );
```

Например, для последовательности -17, 5, 5, 5, 20 вызов `equal_range()` с `value = 10` даст в результате пару итераторов, указывающих на элементы 20 и 20, а вызов с `value = 5` — на первый из элементов, равных 5 и 20 (см. листинги 17.21, 17.22).

Алгоритм *inplace_merge()* выполняет слияние двух *смежных* отсортированных частей одной последовательности: `[first, middle]` и `[middle, last]`. В результате интервал `[first, last]` будет содержать упорядоченную совокупность элементов обоих интервалов.

```
template < class Bi >
    void inplace_merge( Bi first, Bi middle, For last );
template < class Bi, class Compare >
    void inplace_merge( Bi first, Bi middle, For last, Compare comp );
```

Алгоритм *merge()* выполняет слияние *отсортированных* последовательностей в новую последовательность. При этом элементы из исходных последовательностей не удаляются. При равенстве элементов элементы первой последовательности предшествуют элементам второй.

```
template < class In1, class In2, class Out >
    Out merge( In1 first1, In1 last1, In2 first2, In2 last2, Out result );
template < class In1, class In2, class Out, class Compare >
    Out merge( In1 first1, In1 last1, In2 first2, In2 last2, Out result,
               Compare comp );
```

Алгоритм *lexicographical_compare()* выполняет поэлементное сравнение двух последовательностей либо с использованием операции "<", либо с помощью заданной функции `comp`. Алгоритм возвращает `true`, если первая последовательность лексикографически меньше второй (очередной элемент первой последовательности оказался

меньше соответствующего элемента второй последовательности). Если длины последовательностей не совпадают, то недостающие элементы более короткой последовательности считаются меньшими соответствующих элементов более длинной последовательности.

```
template < class In1, class In2 >
    bool lexicographical_compare( In1 first1, In1 last1, In2 first2,
                                  In2 last2 );

template < class In1, class In2, class Compare >
    bool lexicographical_compare( In1 first1, In1 last1, In2 first2,
                                  In2 last2, Compare comp );
```

Алгоритмы *lower_bound()* и *upper_bound()*. Алгоритм *lower_bound()* находит итератор на первый, а *upper_bound()* — на последний элемент *отсортированной* последовательности, перед которым можно вставить заданное значение, не нарушая упорядоченности:

```
template < class For, class T >
    For lower_bound( For first, For last, const T &value );

template < class For, class T, class Compare >
    For lower_bound( For first, For last, const T &value, Compare comp );

template < class For, class T >
    For upper_bound( For first, For last, const T &value );

template < class For, class T, class Compare >
    For upper_bound( For first, For last, const T &value, Compare comp );
```

Алгоритмы *max()* и *min()*. Алгоритм *max()* ищет наибольшее из двух значений, а *min()* — наименьшее, используя или операцию "<", или собственный критерий сравнения:

```
template < class T >
    const T &max( const T &a, const T &b );

template < class T, class Compare >
    const T &max( const T &a, const T &b, Compare comp );

template < class T >
    const T &min( const T &a, const T &b );

template < class T, class Compare >
    const T &min( const T &a, const T &b, Compare comp );
```

Алгоритмы *max_element()* и *min_element()*. Алгоритм *max_element()* возвращает итератор на наибольшее значение в последовательности, а *min_element()* — на наименьшее значение:

```
template < class For >
    For max_element( For first, For last );

template < class For, class Compare >
    For max_element( For first, For last, Compare comp );

template < class For >
    For min_element( For first, For last );

template < class For, class Compare >
    For min_element( For first, For last, Compare comp );
```

Алгоритмы *next_permutation()* и *prev_permutation()*. Алгоритм `next_permutation()` производит в последовательности очередную перестановку в лексикографическом порядке, а алгоритм `prev_permutation()` — предыдущую. Алгоритмы возвращают значение `true`, если следующая перестановка существует.

```
template < class Bi >
    bool next_permutation( Bi first, Bi last );
template < class Bi, class Compare >
    bool next_permutation( Bi first, Bi last, Compare comp );
template < class Bi >
    bool prev_permutation( Bi first, Bi last );
template < class Bi, class Compare >
    bool prev_permutation( Bi first, Bi last, Compare comp );
```

Алгоритм *nth_element()* выполняет частичную сортировку последовательности. После выполнения алгоритма, значение элемента, заданного итератором `nth`, будет таким же, как после полной сортировки. Это означает, что все элементы левее этой позиции будут меньше него, а все, что правее, — больше.

```
template < class Ran >
    void nth_element( Ran first, Ran nth, Ran last );
template < class Ran, class Compare >
    void nth_element( Ran first, Ran nth, Ran last, Compare comp );
```

Этот алгоритм можно использовать в быстрой сортировке Хоора для разбиения исходного сегмента (массива) на подсегменты.

Алгоритмы *patial_sort()* и *patial_sort_copy()*. Алгоритм `patial_sort()`, так же как и предыдущий алгоритм, производит частичную сортировку последовательности. После выполнения алгоритма элементы от `first` до `middle` будут располагаться в таком же порядке, как после полной сортировки. Алгоритм `partial_sort_copy()` выполняет те же действия с копией последовательности.

```
template < class Ran >
    void partial_sort( Ran first, Ran middle, Ran last );
template < class Ran, class Compare >
    void partial_sort( Ran first, Ran middle, Ran last, Compare comp );
template < class In, class Ran >
    void partial_sort_copy( In first, In last, Ran result_first,
                           Ran result_last );
template < class In, class Ran, class Compare >
    void partial_sort_copy( In first, In last,
                           Ran result_first, Ran result_last, Compare comp );
```

Частичная сортировка экономит время в тех случаях, когда интересуются только несколькими самыми большими или самыми маленькими значениями.

Алгоритмы *partition()* и *stable_partition()*. Алгоритм `partition()` размещает элементы, удовлетворяющие заданному условию, перед остальными элементами последовательности. Алгоритм `stable_partition()` выполняет то же самое, но с сохранением относительного порядка элементов. Условие задается с помощью функции или функционального объекта.

```
template < class Bi, class Pred >
```

```

    Bi partition( Bi first, Bi last, Pred pred );
template < class Bi, class Pred >
    Bi stable_partition( Bi first, Bi last, Pred pred );

```

ПРИМЕЧАНИЕ

Как вы заметили, для каждого из перечисленных алгоритмов сортировки существуют две формы: одна использует операцию "меньше (<)", а другая функцию сравнения, заданную пользователем. Многим из алгоритмов требуются итераторы произвольного доступа.

Проиллюстрируем использование алгоритмов сортировки и алгоритмов, связанных с сортировкой, примером, который является очень важным для практического освоения материала — изучите его внимательно (листинги 17.21, 17.22).

Листинг 17.21. Файл alg_sort.cpp

```

/*
    Алгоритмы сортировки последовательности и алгоритмы, связанные с сортировкой.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
#include <vector>             // Векторы
#include <algorithm>          // Алгоритмы
#include <functional>         // Функциональные объекты

using namespace             // Используем стандартное
    std;                   // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    const int  sz = 5;      // Размер векторов
    // Создаем, инициализируем и печатаем значения элементов трех
    // векторов
    vector< int >
        v( sz, 5 );
    v[ 0 ] = 20; v[ 4 ] = -17;
    vector< int >
        v1( v ), v2( 2*sz );
    cout << "Векторы <v> и <v1>: ";
    for( int i=0; i<sz; i++ )
        cout << v[ i ] << " ";
    cout << endl << "Их размер sz = " << sz << endl;

    // Обычная сортировка вектора <v>
    sort( v.begin( ), v.end( ) );
    cout << endl << "После sort( v.begin( ), v.end( ) );"
        << endl;

```

```

for( i=0; i<sz; i++ )
    cout << v[ i ] << " ";
cout << endl;

// Устойчивая сортировка вектора <v1>
stable_sort( v1.begin( ), v1.end( ) );
cout << endl << "После stable_sort( v1.begin( ), "
    "v1.end( ) );" << endl;
for( i=0; i<sz; i++ )
    cout << v1[ i ] << " ";
cout << endl << endl;

// Двоичный поиск в векторах
if( binary_search( v.begin( ), v.end( ), 10 ) )
    cout << "В <v> значение 10 обнаружено";
else
    cout << "В <v> значение 10 не обнаружено";
cout << endl;
if( binary_search( v1.begin( ), v1.end( ), 5 ) )
    cout << "В <v1> значение 5 обнаружено";
else
    cout << "В <v1> значение 5 не обнаружено";
cout << endl;

// Нахождение границ в векторе <v> для вставки значения 10 без
// нарушения порядка элементов
pair< vector< int >::iterator, vector< int >::iterator >
    p = equal_range( v.begin( ), v.end( ), 10 );
cout << "\nДиапазон элементов в <v> для вставки 10: " << *(p.first)
    << " " << *(p.second);

// Нахождение границ в векторе <v1> для вставки значения 5 без
// нарушения порядка элементов
p = equal_range( v1.begin( ), v1.end( ), 5 );
cout << "\nДиапазон элементов в <v1> для вставки 5: " << *(p.first)
    << " " << *(p.second);
cout << endl;

// Слияние двух отсортированных частей одной последовательности
int      arr[ 5 ] = { -7, 2, 17, -100, -80 };
cout << endl << "arr: ";
for( i=0; i<5; i++ )
    cout << arr[ i ] << " ";
cout << endl;
inplace_merge( arr, arr+3, arr+5 );
cout << "После inplace_merge( arr, arr+3, arr+5 );" << endl <<

```

```

    "arr: ";
for( i=0; i<5; i++ )
    cout << arr[ i ] << " ";
cout << endl << endl;

// Лексикографическое сравнение <v> и <v1>
cout << "После lexicographical_compare( v.begin( ), v.end( ),"
    " \nv1.begin( ), v1.end( ) ); получаем: " <<
    lexicographical_compare(v.begin( ), v.end( ), v1.begin( ),
        v1.end( ) ) << endl;
vector< int >::iterator
    it = v.end( );
cout << "vector<int>::iterator it = v.end( );" << endl;
cout << "После lexicographical_compare( v.begin( ), --it,"
    " \nv1.begin( ), v1.end( ) ); получаем: " <<
    lexicographical_compare(v.begin( ), --it, v1.begin( ),
        v1.end( ) ) << endl;
// Лексикографическое сравнение последовательностей с использованием
// предиката
int        a[ 5 ] = {5, 3, 2, 3, 1},
            b[ 5 ] = {5, 3, 2, 3, 2};
cout << "\na[ 5 ] = {5, 3, 2, 3, 1}" << endl <<
    "b[ 5 ] = {5, 3, 2, 3, 2}" << endl;
cout << "lexicographical_compare( a, a+5, b, b+5,"
    "\ngreater<int>( ) ); дает " << lexicographical_compare( a, a+5,
    b, b+5, greater<int>( ) ) << endl << endl;

// Определение итераторов для вставки заданного элемента
cout << "В векторе v элемент 5 можно вставить после " << endl <<
    "*" (lower_bound(v.begin(),v.end(),5)) = " <<
    * (lower_bound(v.begin( ),v.end( ),5)) << endl;
cout << "В векторе v элемент 5 можно вставить перед " << endl <<
    "*" (upper_bound(v.begin(),v.end(),5)) = " <<
    * (upper_bound(v.begin( ),v.end( ),5)) << endl;

// Нахождение наименьшего и наибольшего элементов в v
cout << endl << "В векторе v наибольший элемент " << endl <<
    "*" (max_element(v.begin(),v.end())) = "
    << * ( max_element( v.begin(), v.end() ) ) << endl;
cout << "В векторе v наименьший элемент " << endl <<
    "*" (min_element(v.begin(),v.end())) = "
    << * ( min_element( v.begin(), v.end() ) ) << endl;

// Слияние отсортированных векторов v и v1 в вектор v2
cout << "\nСлияние отсортированных векторов v и v1 в вектор v2"
    << endl << "v2: ";

```

```

merge( v.begin( ), v.end( ), v1.begin( ), v1.end( ), v2.begin( ) );
for( i=0; i<2*sz; i++ )
    cout << v2[ i ] << " ";
cout << endl << endl;

// Генерация перестановок элементов последовательности
//   в лексикографическом порядке
vector< int >
    v3( 3 ), v4( 3 );
v3[ 0 ] = v4[ 2 ] = 1; v3[ 1 ] = v4[ 1 ] = 2;
v3[ 2 ] = v4[ 0 ] = 3;
cout << "v3: ";
for( i=0; i<3; i++ )
    cout << v3[ i ] << " ";
cout << endl << "next_permutation( v3.begin( ),"
    " v3.end( ) );" << endl;
while( next_permutation( v3.begin( ), v3.end( ) ) )
{
    for( i=0; i<3; i++ )
        cout << v3[ i ] << " ";
    cout << endl;
}
cout << endl;
cout << "v4: ";
for( i=0; i<3; i++ )
    cout << v4[ i ] << " ";
cout << endl << "prev_permutation( v4.begin( ), v4.end( ) );"
    << endl;
while( prev_permutation( v4.begin( ), v4.end( ) ) )
{
    for( i=0; i<3; i++ )
        cout << v4[ i ] << " ";
    cout << endl;
}

// Частичная сортировка последовательности
cout << "\na[ 5 ] = {5, 3, 2, 3, 1}" << endl;
nth_element( a, a+2, a+5 );
cout << "После nth_element( a, a+2, a+5 );" << endl << "a: ";
for( i=0; i<5; i++ )
    cout << a[ i ] << " ";
cout << endl;

// Частичная сортировка начала последовательности
partial_sort( a, a+2, a+5, greater<int>( ) );
cout << "После partial_sort( a, a+2, a+5, greater<int>( ) );" << endl

```

```

    << "a: ";
    for( i=0; i<5; i++ )
        cout << a[ i ] << " ";
    cout << endl;

    // Перестановка элементов по условию в начало последовательности
    vector< int >
        v5( v2 );
    partition( v2.begin( ), v2.end( ),
        bind2nd( greater< int > ( ), 7 ) );
    cout << "\nПосле partition( v2.begin( ), v2.end( ),\n"
        " bind2nd( greater< int > ( ), 7 ) ); " << endl << "v2: ";
    for( i=0; i<2*sz; i++ )
        cout << v2[ i ] << " ";
    cout << endl;
    cout << "\nv5: ";
    for( i=0; i<2*sz; i++ )
        cout << v5[ i ] << " ";
    cout << endl;
    stable_partition( v5.begin( ), v5.end( ),
        bind2nd( greater< int > ( ), 7 ) );
    cout << "После stable_partition( v5.begin( ), v5.end( ),"
        "\nbind2nd( greater< int > ( ), 7 ) ); " << endl << "v5: ";
    for( i=0; i<2*sz; i++ )
        cout << v5[ i ] << " ";
    cout << endl;

    cout << endl;
    return 0;
}

```

Листинг 17.22. Результат работы программы

Векторы <v> и <v1>: 20 5 5 5 -17

Их размер sz = 5

После sort(v.begin(), v.end());

-17 5 5 5 20

После stable_sort(v1.begin(), v1.end());

-17 5 5 5 20

В <v> значение 10 не обнаружено

В <v1> значение 5 обнаружено

Диапазон элементов в <v> для вставки 10: 20 20

Диапазон элементов в <v1> для вставки 5: 5 20

```
arr: -7 2 17 -100 -80
```

```
После inplace_merge( arr, arr+3, arr+5 );
```

```
arr: -100 -80 -7 2 17
```

```
После lexicographical_compare( v.begin( ), v.end( ),
```

```
v1.begin( ), v1.end( ) ); получаем: 0
```

```
vector<int>::iterator it = v.end( );
```

```
После lexicographical_compare( v.begin( ), --it,
```

```
v1.begin( ), v1.end( ) ); получаем: 1
```

```
a[ 5 ] = {5, 3, 2, 3, 1}
```

```
b[ 5 ] = {5, 3, 2, 3, 2}
```

```
lexicographical_compare( a, a+5, b, b+5,
```

```
greater<int>( ) ); дает 0
```

В векторе v элемент 5 можно вставить после

```
*(lower_bound(v.begin(),v.end(),5)) = 5
```

В векторе v элемент 5 можно вставить перед

```
*(upper_bound(v.begin(),v.end(),5)) = 20
```

В векторе v наибольший элемент

```
*(max_element(v.begin(),v.end())) = 20
```

В векторе v наименьший элемент

```
*(min_element(v.begin(),v.end())) = -17
```

Слияние отсортированных векторов v и v1 в вектор v2

```
v2: -17 -17 5 5 5 5 5 20 20
```

```
v3: 1 2 3
```

```
next_permutation( v3.begin( ), v3.end( ) );
```

```
1 3 2
```

```
2 1 3
```

```
2 3 1
```

```
3 1 2
```

```
3 2 1
```

```
v4: 3 2 1
```

```
prev_permutation( v4.begin( ), v4.end( ) );
```

```
3 1 2
```

```
2 3 1
```

```
2 1 3
```

```
1 3 2
```

```
1 2 3
```

```
a[ 5 ] = {5, 3, 2, 3, 1}
```

```
После nth_element( a, a+2, a+5 );
```

```
a: 1 2 3 3 5
```

```
После partial_sort( a, a+2, a+5, greater<int>( ) );
```

```
a: 5 3 1 2 3
```

```
После partition( v2.begin( ), v2.end( ),
```

```
bind2nd( greater < int > ( ), 7 ) );
```

```
v2: 20 20 5 5 5 5 5 -17 -17
```

```
v5: -17 -17 5 5 5 5 5 20 20
```

```
После stable_partition( v5.begin( ), v5.end( ),
```

```
bind2nd( greater < int > ( ), 7 ) );
```

```
v5: 20 20 -17 -17 5 5 5 5 5
```

```
Press any key to continue
```

17.4. Алгоритмы работы с множествами и пирамидами

Алгоритмы данной группы выполняют сортировку множеств и операции с пирамидами. В результате сортировки множества отсортированная последовательность рассматривается как *множество*, а операции объединения и пересечения имеют тот же смысл, что и в теории множеств. Сортировка множеств не изменяет входные последовательности, выходные последовательности упорядочены.

Пирамидой (кучей, двоичным деревом) является последовательность, для всех элементов которой выполняются условия:

$$a[i] \geq a[2 \cdot i + 1]$$

$$a[i] \geq a[2 \cdot i + 2]$$

Пример пирамиды из 7 целых чисел: 100 50 -50 40 20 -100 -70.

Как следует из определения, максимальный элемент пирамиды расположен *первым* (в вершине пирамиды, двоичного дерева). Это же свойство относится и к любой подпирамиде (двоичному поддереву). Особенностью пирамиды является также то, что добавление и удаление элементов производится с *логарифмической сложностью*. По этой причине пирамиды удобно использовать для реализации *очередей с приоритетами*, так как существуют эффективные алгоритмы извлечения первого элемента и добавления нового с сохранением условия пирамидальности. Для работы с пирамидами требуются *итераторы произвольного доступа*.

Алгоритмы работы с пирамидами и множествами существуют в двух формах: первая использует операцию "<", а вторая — функцию сравнения, заданную пользователем в виде бинарного предиката.

Представим алгоритмы, входящие в состав этой группы.

Алгоритм *includes()* выполняет проверку включения одного множества в другое. Результат равен **true** в том случае, когда каждый элемент последовательности [first2, last2) содержится в последовательности [first1, last1).

```
template < class In1, class In2 >
```

```
bool includes( In1 first1, In1 last1, In2 first2, In2 last2 );
```

```
template < class In1, class In2, class Compare >
    bool includes( In1 first1, In1 last1, In2 first2, In2 last2,
                  Compare comp );
```

Алгоритм *set_intersection()* создает отсортированное пересечение множеств. При этом исходные множества сохраняются неизменными.

```
template < class In1, class In2, class Out >
    Out set_intersection( In1 first1, In1 last1, In2 first2, In2 last2,
                        Out res );

template < class In1, class In2, class Out, class Compare >
    Out set_intersection( In1 first1, In1 last1, In2 first2, In2 last2,
                        Out res, Compare comp );
```

ЗАМЕЧАНИЕ

Элементы множеств должны быть отсортированными.

Алгоритмы *set_difference()* и *set_symmetric_difference()*. Алгоритм *set_difference()* выполняет копирование в *Out* элементов, входящих только в первую из двух последовательностей:

```
template < class In1, class In2, class Out >
    Out set_difference( In1 first1, In1 last1, In2 first2, In2 last2,
                      Out result );

template < class In1, class In2, class Out, class Compare >
    Out set_difference( In1 first1, In1 last1, In2 first2, In2 last2,
                      Out result, Compare comp );
```

Алгоритм *set_symmetric_difference()* выполняет копирование в *Out* элементов, входящих только в одну из двух последовательностей:

```
template < class In1, class In2, class Out >
    Out set_symmetric_difference( In1 first1, In1 last1, In2 first2,
                                In2 last2, Out result );

template < class In1, class In2, class Out, class Compare >
    Out set_symmetric_difference( In1 first1, In1 last1, In2 first2,
                                In2 last2, Out result, Compare comp );
```

Алгоритм *set_union()* создает отсортированное объединение множеств без повторяющихся элементов:

```
template < class In1, class In2, class Out >
    Out set_union( In1 first1, In1 last1, In2 first2, In2 last2,
                  Out result );

template < class In1, class In2, class Out, class Compare >
    Out set_union( In1 first1, In1 last1, In2 first2, In2 last2,
                  Out result, Compare comp );
```

Проиллюстрируем использование перечисленных алгоритмов работы с множествами примером [9] (листинги 17.23, 17.24). Советуем внимательно изучить его — это важно для практического освоения алгоритмов.

Листинг 17.23. Файл alg_set.cpp

```

/*
    Алгоритмы работы с множествами.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>          // Поточковый ввод-вывод
#include <algorithm>         // Алгоритмы

using namespace            // Используем стандартное
    std;                   // пространство имен

// Отображение элементов последовательности на экране
void display(
    const char *pHeader, // Указатель на заголовок
    const int *pBeg,     // Указатель на начало
    const int *pEnd )    // Указатель на конец
{
    cout << pHeader;      // Печатаем заголовок
    // Печатаем последовательность
    copy( pBeg, pEnd, ostream_iterator< int >( cout, " " ) );
    cout << endl;

    return;
}

int main( void )           // Возвращает 0 при успехе
{
    // Создаем, инициализируем и печатаем значения элементов
    // последовательностей
    int      a[ ] = {2, 5, 7, 9},
            b[ ] = {1, 5, 9};

    display( "a: ", a, a+4 );
    display( "b: ", b, b+3 );
    cout << endl;

    // Создаем отсортированное пересечение множеств <a> и <b>, помещаем
    // его в intersection[ 3 ] и печатаем
    int      intersection[ 3 ], *p_isect;
    p_isect = set_intersection( a, a+4, b, b+3, intersection );
    display( "intersection: ", intersection, p_isect );

    // Создаем отсортированное объединение множеств <a> и <b>, помещаем
    // его в Union[ 7 ] и печатаем
    int      Union[ 7 ], *p_union;

```

```

p_union = set_union( a, a+4, b, b+3, Union );
display( "Union: ", Union, p_union );

// Копируем в difference[ 4 ] элементы, входящие только в первую из
// последовательностей <a> и <b>, и печатаем difference[ 4 ]
int      difference[ 4 ], *p_difference;
p_difference = set_difference( a, a+4, b, b+3, difference );
display( "difference: ", difference, p_difference );

// Копируем в sym_difference[ 7 ] элементы, входящие только в любую
// из последовательностей <a> и <b>, и печатаем sym_difference[ 7 ]
int      sym_difference[ 7 ], *p_sym_dif;
p_sym_dif = set_symmetric_difference( a, a+4, b, b+3,
                                      sym_difference );
display( "sym_difference: ", sym_difference, p_sym_dif );

// Проверяем включение <b> в <a> и печатаем результат
if( includes( a, a+4, b, b+3 ) )
    cout << "<a> включает <b>";
else
    cout << "<a> не включает <b>";
cout << endl;
// Проверяем включение <b> в <Union> и печатаем результат
if( includes( Union, p_union, b, b+3 ) )
    cout << "<Union> включает <b>";
else
    cout << "<Union> не включает <b>";
cout << endl;

cout << endl;
return 0;
}

```

Листинг 17.24. Результат работы программы

```

a: 2 5 7 9
b: 1 5 9

intersection: 5 9
Union: 1 2 5 7 9
difference: 2 7
sym_difference: 1 2 7
<a> не включает <b>
<Union> включает <b>

```

Press any key to continue

А теперь рассмотрим алгоритмы, предназначенные для работы с пирамидами.

Алгоритм *make_heap()* выполняет преобразование последовательности с произвольным доступом в пирамиду:

```
template < class Ran >
    void make_heap( Ran first, Ran last );
template < class Ran, class Compare >
    void make_heap( Ran first, Ran last, Compare comp );
```

Алгоритм *pop_heap()* удаляет первый элемент последовательности, а затем восстанавливает условие пирамидальности:

```
template < class Ran >
    void pop_heap( Ran first, Ran last );
template < class Ran, class Compare >
    void pop_heap( Ran first, Ran last, Compare comp );
```

Получить значение элемента, извлекаемого из пирамиды, можно так:

```
x = *a; pop_heap( a, a+m );
```

Алгоритм *push_heap()* выполняет преобразование последовательности в пирамиду после добавления в нее последнего элемента:

```
template < class Ran >
    void push_heap( Ran first, Ran last );
template < class Ran, class Compare >
    void push_heap( Ran first, Ran last, Compare comp );
```

До вызова *push_heap()* в последовательность следует добавить элемент способом, соответствующим типу контейнера, например:

```
v.push_back( x ); push_heap( v.begin( ), v.end( ) );
```

Алгоритм *sort_heap()* преобразует пирамиду в отсортированную по возрастанию последовательность:

```
template < class Ran >
    void sort_heap( Ran first, Ran last );
template < class Ran, class Compare >
    void sort_heap( Ran first, Ran last, Compare comp );
```

Этот алгоритм использует свойства пирамиды и, поэтому, работает быстрее обычной сортировки. Сортировка не сохраняет относительный порядок следования элементов с одинаковыми ключами.

Поясним использование алгоритмов работы с пирамидами с помощью примера (листинги 17.25, 17.26).

Листинг 17.25. Файл `alg_heap.cpp`

```
/*
    Алгоритмы работы с пирамидами.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/

#include <iostream>           // Поточковый ввод-вывод
```

```
#include <vector>           // Векторы
#include <algorithm>         // Алгоритмы

using namespace std;        // Используем стандартное
                             // пространство имен

int main( void )            // Возвращает 0 при успехе
{
    // Размер вектора
    const int VECTOR_SIZE = 8;

    // Создаем, инициализируем и печатаем вектор
    vector< int >
        Numbers( VECTOR_SIZE );
    Numbers[ 0 ] = 4; Numbers[ 1 ] = 10; Numbers[ 2 ] = 70;
    Numbers[ 3 ] = 10; Numbers[ 4 ] = 30; Numbers[ 5 ] = 69;
    Numbers[ 6 ] = 96; Numbers[ 7 ] = 100;
    // Итератор
    vector< int >::iterator
        it, itl;
    cout << "Numbers{ ";
    for( it=Numbers.begin( ); it!=Numbers.end( ); it++ )
        cout << *it << " ";
    cout << "}\n" << endl;

    // Преобразуем Numbers в пирамиду и печатаем ее
    make_heap( Numbers.begin( ), Numbers.end( ) );
    cout << "После вызова make_heap( )" << endl;
    cout << "Numbers{ ";
    for( it=Numbers.begin( ); it!=Numbers.end( ); it++ )
        cout << *it << " ";
    cout << "}\n" << endl;

    // Преобразуем пирамиду в отсортированную последовательность и
    // печатаем
    sort_heap( Numbers.begin( ), Numbers.end( ) );
    cout << "После вызова sort_heap( )" << endl;
    cout << "Numbers{ ";
    for( it=Numbers.begin( ); it!=Numbers.end( ); it++ )
        cout << *it << " ";
    cout << "}\n" << endl;

    // Добавляем в последовательность элемент со значением 7 и печатаем
    // ее
    Numbers.push_back( 7 );
    push_heap( Numbers.begin( ), Numbers.end( ) );
    cout << "После вызова push_heap( )" << endl;
    cout << "Numbers{ ";
```

```

for( it=Numbers.begin( ); it!=Numbers.end( ); it++ )
    cout << *it << " ";
cout << "}\n" << endl;

// Превращаем последовательность в пирамиду и печатаем ее
make_heap( Numbers.begin( ), Numbers.end( ) );
cout << "После вызова make_heap( )" << endl;
cout << "Numbers{ ";

for( it=Numbers.begin( ); it!=Numbers.end( ); it++ )
    cout << *it << " ";
cout << "}\n" << endl;

// Из пирамиды Numbers извлекаем корневой элемент, печатаем его и
// восстановленную пирамиду
int    x = *( Numbers.begin( ) );
pop_heap( Numbers.begin( ), Numbers.end( ) );
cout << "Корневой элемент пирамиды: " << x << endl;
cout << "После вызова pop_heap( )" << endl;
cout << "Numbers{ ";
it1 = Numbers.end( ); --it1;
for( it=Numbers.begin( ); it!=it1; it++ )
    cout << *it << " ";
cout << "}" << endl ;

cout << endl;
return 0;
}

```

Листинг 17.26. Результат работы программы

```

Numbers{ 4 10 70 10 30 69 96 100 }

После вызова make_heap( )
Numbers{ 100 30 96 10 4 69 70 10 }

После вызова sort_heap( )
Numbers{ 4 10 10 30 69 70 96 100 }

После вызова push_heap( )
Numbers{ 4 10 10 30 69 70 96 100 7 }

После вызова make_heap( )
Numbers{ 100 69 96 30 4 70 10 10 7 }

Корневой элемент пирамиды: 100
После вызова pop_heap( )
Numbers{ 96 69 70 30 4 7 10 10 }

Press any key to continue

```

17.5. Другие средства стандартной библиотеки

Кроме рассмотренных, стандартная библиотека языка C++ содержит следующие менее употребительные средства:

- ☐ для распределения памяти;
- ☐ для численных расчетов;
- ☐ средства поддержки языка;
- ☐ средства диагностики;
- ☐ средства локализации;
- ☐ средства работы с комплексными числами;
- ☐ большое количество функций, унаследованных из стандартной библиотеки языка C, а также типов для их поддержки и макросов.

17.5.1. Распределители памяти [10]

В стандартной библиотеке языка C++ имеются *распределители* (allocators), занимающиеся выделением и освобождением динамической памяти. Среди объектов-распределителей, в стандартной библиотеке определен *распределитель по умолчанию*:

```
namespace std
{
    template < class T >
    class allocator;
```

Этот распределитель используется по умолчанию во всех ситуациях, когда распределитель может передаваться в качестве аргумента. Он использует стандартные средства выделения и освобождения памяти: операции `new` и `delete`. Большинство программ использует распределитель по умолчанию. Иногда библиотеки представляют специализированные распределители, которые передаются в виде аргументов. Таким образом, необходимость в самостоятельном программировании распределителей возникает очень редко. По этой причине мы ограничимся только приведенными ранее сведениями. Более подробную информацию о распределителях памяти можно найти в стандарте языка и в [10, стр. 701—715].

17.5.2. Средства для численных расчетов

В стандартной библиотеке языка C++ для выполнения численных расчетов предусмотрены обобщенные численные алгоритмы, шаблонный класс для работы с массивами чисел `valarray` и четыре вспомогательных шаблонных класса, позволяющих получать различные подмножества `valarray`.

Обобщенные численные алгоритмы. Для их использования требуется подключить заголовочный файл `<numeric>`. Эта группа алгоритмов включает четыре стандартных алгоритма: для накопления суммы элементов последовательности (`accumulate`), для вычисления скалярного произведения двух последовательностей (`inner_product`), для формирования последовательности из начальных частичных сумм элементов

(`partial_sum`) и для вычисления разности между смежными элементами последовательности (`adjacent_difference`).

Более подробные сведения об обобщенных численных алгоритмах можно найти в стандарте языка и в [9, стр. 369—371].

Шаблонный класс для работы с массивами чисел и вспомогательные шаблонные классы. Для эффективной работы с массивами чисел в стандартной библиотеке определен шаблонный класс `valarray`. Для использования этого класса и связанных с ним средств (четыре вспомогательных класса: `slice_array`, `gslice_array`, `mask_array` и `indirect_array`) следует подключить к программе заголовочный файл `<valarray>`. В этом заголовочном файле описаны также обычные классы `slice` и `gslice`, задающие подмножества индексов элементов массива.

Шаблонный класс `indirect_array` и операции с этим классом реализованы в расчете на их поддержку в архитектурах высокопроизводительных систем.

Вспомогательный шаблонный класс `slice_array` позволяет выделить и представить столбец, строку или другое регулярное подмножество элементов `valarray`. С помощью этого шаблона можно, например, представить `valarray` как матрицу произвольной размерности. Шаблон `gslice_array` позволяет задавать более сложные законы формирования подмножества массива, а `mask_array` — произвольные подмножества массива с помощью битовой маски. Шаблон `indirect_array` содержит не сами элементы массива, а их индексы.

Обычный класс `slice` (срез) позволяет задать подмножество индексов элементов массива, а класс `gslice` (обобщенный срез) используют при работе с подмножеством массива, которое нельзя задать одним срезом.

Более полные сведения о перечисленных средствах стандартной библиотеки можно найти в стандарте языка C++ и в [10, стр. 525—556].

17.5.3. Средства поддержки языка

Средства поддержки языка включают описания функций и типов, которые используются при выполнении программы. Они включают в себя поддержку запуска и завершения программы, операций `new` и `delete`, описание свойств встроенных типов данных, динамическую идентификацию типов, обработку исключений и другие средства времени выполнения. Эти средства определены в заголовочных файлах `<cstddef>`, `<limits>`, `<climits>`, `<cfloat>`, `<cstdlib>`, `<new>`, `<typeinfo>`, `<exception>`, `<cstdarg>`, `<setjmp>`, `<ctime>` и `<csignal>`. Константы, макросы и типы данных языка C++ приведены в *приложении 6* на компакт-диске.

17.5.4. Средства диагностики

Средства диагностики включают стандартные классы и функции, которые используются для диагностики ошибок, возникающих в процессе работы программы. Все ошибки можно разделить на *логические* (`logic_error`), которые можно обнаружить до запуска программы, и *ошибки времени выполнения* (`runtime_error`). Для диагностики ошибок в стандартной библиотеке языка C++ определена простая иерархия классов исключений (рис. 17.1), которая должна служить основой для обработки исключений, создаваемой программистом [9].

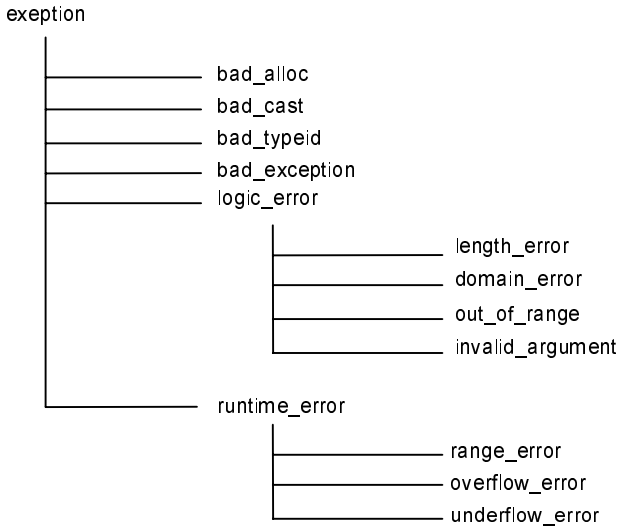


Рис. 17.1. Иерархия классов исключений

Все классы для обработки стандартных исключений являются потомками описанного в заголовочном файле `<exception>` класса `exception`. Эти классы имеют следующее назначение:

- ❑ `bad_alloc` — обработка ошибки динамического выделения памяти с помощью операции `new`;
- ❑ `bad_cast` — обработка ошибки неправильного использования оператора `dynamic_cast`;
- ❑ `bad_typeid` — обработка ошибки, если операция `typeid` не может определить тип операнда;
- ❑ `bad_exception` — обработка ошибки, порождаемой, если при вызове функции произошло неожиданное исключение;
- ❑ `length_error` — обработка ошибки при попытке создания объекта, размер которого превышает максимальный размер для типа `size_t`;
- ❑ `domain_error` — обработка ошибки при нарушении внутренних условий перед выполнением действия;
- ❑ `out_of_range` — обработка ошибки при попытке вызова функции с аргументом, не попадающим в область допустимых значений;
- ❑ `invalid_argument` — обработка ошибки при попытке вызова функции с неверным аргументом;
- ❑ `range_error` — обработка ошибки при получении неверного результата;
- ❑ `overflow_error` — обработка ошибки при арифметическом переполнении (получен слишком большой результат для указанного типа данных);
- ❑ `underflow_error` — обработка ошибки при исчезновении порядка (получен слишком маленький результат для указанного типа данных).

17.5.5. Средства локализации и работы с комплексными числами

Средства локализации включают упорядоченную по категориям информацию, специфичную для разных стран (формат даты и времени, формат национальной валюты, представление чисел, символов и т. п.). Описание средств локализации находится в заголовочных файлах `<locale>` и `<clocale>`. Подробную информацию о средствах локализации можно найти в стандарте языка и в [10, стр. 657—700].

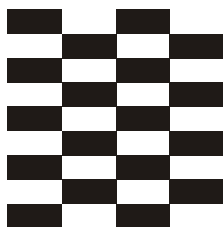
Для работы с комплексными числами стандартная библиотека представляет шаблон классов `complex`, его специализации для типов `float`, `double` и `long double`, а также многочисленные функции. Работа с комплексными числами возможна после подключения заголовочного файла `<complex>`. Подробную информацию об этих средствах можно найти в стандарте языка и в [10, стр. 510—525].

На компакт-диске для стандартной библиотеки языка C++ описаны заголовочные файлы в *приложении 5*, константы, макросы и типы данных — в *приложении 6* и функции библиотеки — в *приложении 7*.

17.6. Вопросы для самопроверки

1. В каких файлах содержатся объявления стандартных функциональных объектов и стандартных алгоритмов?
2. Перечислите основные категории алгоритмов STL.

Ответы на эти вопросы приведены в *разд. П1.14 приложения 1*.



Приложения

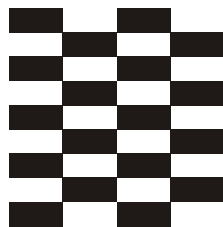
**Приложение 1. Ответы и решения к вопросам
и упражнениям для самопроверки**

**Приложение 2. Тесты и программные проекты.
Варианты заданий**

**Приложение 3. Создание и отладка программного
проекта консольного приложения
в Microsoft Visual Studio C++ .NET**

Приложение 4. Прилагаемый компакт-диск

Приложение 1



Ответы и решения к вопросам и упражнениям для самопроверки

П1.1. Глава 1

1. Чем определяется размер объекта класса?

Размер объекта класса в памяти определяется суммой размеров членов-данных класса. Методы класса не занимают место в области памяти, выделенной для объекта класса.

2. Почему не следует делать члены-данные класса открытыми?

Открытость члена-данного класса делает программу менее надежной, так как позволяет изменить его значение из произвольной программной среды (нарушает принцип инкапсуляции).

Кроме того, объявление членов-данных класса закрытыми позволяет клиенту класса использовать данные, не заботясь о том, как они хранятся или вычисляются.

3. Имеет ли смысл использовать структуры в программах на языке C++?

Многие программисты используют служебное слово `struct` для классов, которые не имеют методов. Можно расценивать это как склонность к использованию устаревших структур языка C, которые не могли иметь функций. Следовательно, структуры в программах на языке C++ лучше не использовать. Зачем же они тогда введены в язык C++? Ответ — их наличие обеспечивает совместимость языков C и C++ снизу вверх.

4. Что представляет собой оператор прямого доступа и для чего он используется с объектом класса?

Оператор прямого доступа представляет собой символ точки `.`. Он используется для обращения к членам класса.

5. Что резервирует память — определение класса или определение объекта с типом класса?

Память резервируется определением объекта с типом класса. Определение класса не резервирует память.

6. Объявление класса является его интерфейсом или реализацией класса?

Объявление класса является его интерфейсом, который сообщает пользователям класса, как с ним взаимодействовать. Объявление класса обычно помещается

во включаемый (заголовочный) файл с расширением `h` или `hpp`. Реализация класса — это набор определений методов класса, сохраняемых обычно в файле с расширением `cpp`.

ОСОБЕННОСТИ РАЗМЕЩЕНИЯ ОПРЕДЕЛЕНИЯ ШАБЛОНА КЛАССА

И объявление шаблона класса и определения его методов должны располагаться в заголовочном файле с расширением `h` или `hpp`. Объясняется это тем, что конкретный экземпляр шаблона класса генерируется при выполнении программы.

7. Какова разница между открытыми (`public:`) и закрытыми (`private:`) членами-данными класса?

К открытым членам-данным класса могут обращаться пользователи класса (они доступны из произвольной программной среды), а к закрытым могут получить доступ только методы этого же класса и производных от него классов и функции-друзья класса. О наследовании и производных классах см. *гл. 2*.

8. Могут ли методы класса быть закрытыми?

Да. Как методы, так и члены-данные класса могут быть закрытыми.

9. Могут ли данные-члены класса быть открытыми?

Хотя члены-данные класса, в принципе, могут быть открытыми, но считается хорошей практикой программирования, когда члены-данные определяются все же закрытыми, а доступ к ним обеспечивается за счет открытых методов класса. Использование такого подхода обеспечивает большую надежность программно-го кода.

10. Если определить два объекта некоторого класса, то могут ли они иметь различные значения своих членов-данных?

Да. Каждый объект класса имеет свои собственные члены-данные.

11. Нужно ли объявление класса завершать точкой с запятой? А определение метода класса?

Объявление класса всегда заканчивается точкой с запятой после закрывающей фигурной скобки, а определение метода класса — нет.

12. Как бы выглядел заголовок определения метода `GetX()` класса `ANYCLASS`, который не принимает никаких параметров и возвращает целое значение? Считайте, что определение метода помещается вне блока класса.

```
int ANYCLASS :: GetX( void )
```

13. Какой метод вызывается для инициализации объекта с типом класса?

Для инициализации объекта с типом класса вызывается конструктор, причем это происходит автоматически при создании объекта.

14. Напишите объявление класса с именем `EMPLOYEE` (служащие) с открытыми членами-данными целого типа `Age` (возраст) и `Salary` (зарплата).

```
class EMPLOYEE
{
    // Данные
public:
```

```
    unsigned int
        Age;        // Возраст
    unsigned int
        Salary;     // Зарплата
};
```

15. Перепишите класс `EMPLOYEE` из п. 14 таким образом, чтобы сделать члены-данные класса закрытыми и обеспечить открытые методы доступа для чтения и установки значений всех членов-данных.

```
class EMPLOYEE
{
    // Данные
private:
    unsigned int
        Age;        // Возраст
    unsigned int
        Salary;     // Зарплата
    // Методы
public:
    // Получение значения возраста: подставляемый метод
    unsigned int GetAge( void ) const
    {
        return Age;
    }
    // Получение значения зарплаты: подставляемый метод
    unsigned int GetSalary( void ) const
    {
        return Salary;
    }
    // Задание значения возраста: подставляемый метод
    void SetAge( unsigned int age )
    {
        Age = age;
        return;
    }
    // Задание значения зарплаты: подставляемый метод
    void SetSalary( unsigned int salary )
    {
        Salary = salary;
        return;
    }
};
```

16. Напишите программу с использованием класса `EMPLOYEE` из п. 15, которая создает объекты `Ivanov` и `Petrov` этого класса, задает значения членов-данных этих объектов, а затем выводит их на печать.

```
#include <iostream.h>    // Поточковый ввод-вывод
int main( void )        // Возвращает 0 при успехе
```

```

{
    // Создаем два объекта
    EMPLOYEE   Ivanov, Petrov;
    // Инициализируем объекты
    Ivanov.SetAge( 30 ); Petrov.SetAge( 32 );
    Ivanov.SetSalary( 40000 ); Petrov.SetSalary( 30000 );
    // Печатаем информацию об объектах
    cout << "Иванову " << Ivanov.GetAge( ) << " лет и его зарплата "
           << Ivanov.GetSalary( ) << " рублей" << endl;
    cout << "Петрову " << Petrov.GetAge( ) << " лет и его зарплата "
           << Petrov.GetSalary( ) << " рублей" << endl;
    return 0;
}

```

- 17. В класс из п. 15 добавьте метод, который сообщает, сколько тысяч рублей зарабатывает служащий, округляя ответ до тысячи рублей. Напишите определение этого метода, поместив его вне блока класса.**

```

// Прототип метода - его следует поместить в блок класса со
// спецификатором доступа public:
unsigned int GetRoundedThousands( void ) const;
// Определение метода вне блока класса
unsigned int EMPLOYEE :: GetRoundedThousands( void ) const
{
    return ( Salary+500 ) / 1000;
}

```

- 18. Измените класс EMPLOYEE из п. 15 так, чтобы можно было инициализировать члены-данные класса в процессе "создания" служащего.**

```

class EMPLOYEE
{
    // Данные
private:
    unsigned int
        Age;           // Возраст
    unsigned int
        Salary;        // Зарплата
    // Методы
public:
    // Обычный конструктор - инициализация объекта: подставляемый метод
    EMPLOYEE( unsigned int age, unsigned int salary )
    {
        Age = age; Salary = salary;
    }
    // Получение значения возраста: подставляемый метод
    unsigned int GetAge( void ) const
    {

```

```

        return Age;
    }
    // Получение значения зарплаты: подставляемый метод
    unsigned int GetSalary( void ) const
    {
        return Salary;
    }
};

```

19. Что неправильно в следующем объявлении класса?

```

class Square
{
public:
    int        Side;
}

```

Блок класса должен завершаться точкой с запятой.

20. Что весьма полезное отсутствует в следующем объявлении класса?

```

class Square
{
    int        Side;
    void SetSide( int side );
};

```

Отсутствуют спецификаторы доступа. Поэтому метод `SetSide()` и член-данное `side` являются закрытыми. Помните, что члены класса являются закрытыми по умолчанию.

21. Какие три ошибки обнаружит компилятор в этом программном коде?

```

class STATION
{
private:
    int        Station;
public:
    void SetStation( int station );
    int GetStation( void ) const;
};
// ...
int main( void )
{
    STATION    mySTATION;
    mySTATION.Station = 9;
    STATION.SetStation( 10 );
    STATION    myOtherSTATION( 2 );
    return 0;
}

```

Нельзя обращаться к `Station` непосредственно — это закрытый член-данное.

Нельзя вызывать метод `setStation()` прямо в классе, этот метод можно вызывать только для объекта класса.

Нельзя инициализировать член-данные `station` при создании объекта, так как в классе отсутствует нужный для этого конструктор.

22. Какие члены-данные класса следует инициализировать одновременно с инициализацией конструктора, а какие инициализировать в теле конструктора?

Бытует правило, что одновременно с инициализацией конструктора следует инициализировать как можно больше членов-данных класса. Однако, по нашему мнению, это дело вкуса. Поэтому используйте способ инициализации членов-данных класса, который вам больше импонирует.

23. Почему одни методы класса определяются в блоке класса, а другие — за его пределами?

Если метод определяется в блоке класса, то он используется как подставляемый метод. Вместе с тем отметим, что и в подобном случае метод становится действительно подставляемым, если он достаточно прост. Если же определение метода размещается вне блока класса, то метод не является подставляемым. Чтобы в такой ситуации метод сделать подставляемым, достаточно его определение начать со служебного слова `inline`.

24. Какая разница между определением класса и определением объекта класса?

Определение объекта класса резервирует память, а определение класса — нет.

25. Когда вызывается конструктор копирования?

Всегда, когда создается временная копия объекта. Это случается каждый раз, когда объект класса передается по значению:

- ☐ при определении нового объекта с типом `класс` с инициализацией его другим существующим объектом того же типа;
- ☐ при передаче в метод класса параметра-объекта с типом `класс` по значению;
- ☐ при возврате из метода класса значения объекта с типом `класс` с помощью оператора `return`.

26. Когда вызывается деструктор?

Деструктор автоматически вызывается, когда:

- ☐ объект уходит из области действия;
- ☐ завершается программа;
- ☐ используется операция `delete` для объектов, размещенных операцией `new`.

Можно использовать явный вызов с полным именем деструктора.

27. Чем отличается конструктор копирования от оператора присваивания?

Оператор присваивания работает с существующим объектом, а конструктор копирования создает новый временный объект класса.

28. Что представляет собой указатель `this`?

Это скрытый указатель, который указывает на сам объект класса. Его можно использовать для любых членов класса. В частности, указатель `this` является скрытым параметром каждого метода класса.

29. Как различить перегрузку префиксных и постфиксных операторов приращения?

Префиксный оператор не принимает никаких параметров. Постфиксный оператор принимает один параметр типа `int`, который используется в качестве флага, сообщающего компилятору о том, что это постфиксный оператор.

30. Можно ли перегрузить операцию суммирования для операндов типа `short int`?

Нет, для встроенных типов нельзя перегружать никаких операторов.

31. Допускается ли в C++ перегрузка оператора "++" таким образом, чтобы он выполнял операцию декремента?

В принципе допускается, но этого делать не стоит. Операторы нужно перегружать таким образом, чтобы это было понятно любому читателю вашей программы.

32. Напишите объявление класса `SimpleCircle` (простая окружность) с единственным членом-данным `Radius` (радиус). В классе должны использоваться конструктор умолчания и деструктор, а также методы для чтения и записи значения `Radius`.

```
typedef unsigned int uint;
class SimpleCircle
{
    // Данные
private:
    uint      Radius;
    // Методы
public:
    SimpleCircle( void );
    ~SimpleCircle( void );
    void SetRadius( uint radius );
    uint GetRadius( void ) const;
};
```

33. Для класса `SimpleCircle`, созданного в п. 32, напишите определение конструктора умолчания, инициализирующего `Radius` значением 5. Инициализацию выполните во время инициализации конструктора, а определение конструктора умолчания поместите вне блока класса.

```
SimpleCircle :: SimpleCircle( void ) :
    Radius( 5 )
{ }
```

34. Для класса, созданного в п. 32, напишите определение обычного конструктора, который присваивает значение своего параметра члену-данному `Radius`. Инициализацию выполните в блоке конструктора.

```
SimpleCircle :: SimpleCircle( uint radius )
{
    Radius = radius;
}
```

35. Для класса `SimpleCircle`, созданного в пп. 32—34, перегрузите операции преинкремента и постинкремента для члена-данного `Radius`.

```
// Перегрузка оператора преинкремента
SimpleCircle SimpleCircle :: operator++( void )
{
    ++Radius;
    return *this;
}
```

Обратите внимание на то, что в данном случае можно было бы возвращать и ссылку на объект, так как после выполнения перегружающего метода объект продолжает существовать.

```
// Перегрузка оператора постинкремента
SimpleCircle SimpleCircle :: operator++( int )
{
    // Создает локальный объект и инициализирует его текущим объектом
    SimpleCircle
        temp( *this );
    ++Radius;
    return temp;
}
```

Обратите внимание на то, что в данном случае нельзя возвращать ссылку на объект, так как возвращается локальный объект, который после выполнения перегружающего метода перестает существовать.

В обоих случаях при выполнении перегруженных операций будет вызван умалчиваемый конструктор копирования. Это в данном случае допустимо, так как класс не использует динамической памяти.

36. Измените класс `SimpleCircle`, созданный в пп. 32—35, таким образом, чтобы член-данное `Radius` размещался в динамической памяти. Зафиксируйте все созданные методы класса.

```
typedef unsigned int uint;
#include <stdlib.h>          // Для exit( )
class SimpleCircle
{
    // Данные
private:
    uint      *pRadius; // Указатель на радиус
    // Методы
public:
    SimpleCircle( void );
    SimpleCircle( uint radius );
    ~SimpleCircle( void );
    void SetRadius( uint radius );
    uint GetRadius( void ) const;
    SimpleCircle operator++( void );
```

```

        SimpleCircle operator++( int );
};
// *****
// Конструктор умолчания
SimpleCircle :: SimpleCircle( void )
{
    pRadius = new uint( 5 );
    if( !pRadius )
        exit( 1 );
}
// Обычный конструктор
SimpleCircle :: SimpleCircle( uint radius )
{
    pRadius = new uint( radius );
    if( !pRadius )
        exit( 1 );
}
// Деструктор
SimpleCircle :: ~SimpleCircle( void )
{
    if( pRadius )
    {
        delete pRadius; pRadius = NULL;
    }
}
// Задание значения *pRadius
void SimpleCircle :: SetRadius( uint radius )
{
    *pRadius = radius;
    return;
}
// Получение значения *pRadius
uint SimpleCircle :: GetRadius( void ) const
{
    return *pRadius;
}
// Перегрузка оператора преинкремента
SimpleCircle SimpleCircle :: operator++( void )
{
    ++(*pRadius);
    return *this;
}
// Перегрузка оператора постинкремента
SimpleCircle SimpleCircle :: operator++( int )
{

```

```

        // Создает локальный объект и инициализирует его текущим объектом
        SimpleCircle
            temp( *this );
        ++(*pRadius);
        return temp;
    }

```

Обратите внимание, что теперь в классе нужен свой конструктор копирования — класс использует динамическую память.

37. В класс `SimpleCircle`, созданный в п. 36, добавьте конструктор копирования.

```

// Конструктор копирования
SimpleCircle :: SimpleCircle( const SimpleCircle &copy )
{
    uint        value = copy.GetRadius( );
    pRadius = new uint( value );
    if( !pRadius )
        exit( 1 );
}

```

38. В класс `SimpleCircle`, созданный в п. 37, добавьте метод, перегружающий операцию присваивания.

```

// Перегрузка операции присваивания
SimpleCircle & SimpleCircle :: operator=( const SimpleCircle &right )
{
    if( this != &right )
        *pRadius = copy.GetRadius( );
    return *this;
}

```

39. На базе класса `SimpleCircle`, созданного в пп. 37—38, напишите программу, которая создает два объекта этого класса. Для создания одного объекта используйте конструктор умолчания, а второму объекту при его определении присвойте значение 9. С каждым из объектов используйте оператор инкремента и выведите полученные значения на печать. И, наконец, присвойте значение одного объекта другому объекту и выведите результат на печать.

```

typedef unsigned int uint;
#include <stdlib.h>        // Для exit( )
#include <iostream.h>     // Для потокового вывода
class SimpleCircle
{
    // Данные
private:
    uint        *pRadius; // Указатель на радиус
    // Методы
public:
    // Конструкторы и деструктор
    SimpleCircle( void );
    SimpleCircle( uint radius );
}

```

```

SimpleCircle( const SimpleCircle &copy );
~SimpleCircle( void );
// Методы доступа к данным
void SetRadius( uint radius );
uint GetRadius( void ) const;
// Операторы
SimpleCircle operator++( void );
SimpleCircle operator++( int );
SimpleCircle & operator=( const SimpleCircle &right );
};
// *****
// Конструктор умолчания
SimpleCircle :: SimpleCircle( void )
{
    pRadius = new uint( 5 );
    if( !pRadius )
        exit( 1 );
}
// Обычный конструктор
SimpleCircle :: SimpleCircle( uint radius )
{
    pRadius = new uint( radius );
    if( !pRadius )
        exit( 1 );
}
// Конструктор копирования
SimpleCircle :: SimpleCircle( const SimpleCircle &copy )
{
    uint value = copy.GetRadius( );
    pRadius = new uint( value );
    if( !pRadius )
        exit( 1 );
}
// Деструктор
SimpleCircle :: ~SimpleCircle( void )
{
    if( pRadius )
    {
        delete pRadius; pRadius = NULL;
    }
}
// Задание значения *pRadius
void SimpleCircle :: SetRadius( uint radius )
{
    *pRadius = radius;
}

```

```

    return;
}
// Получение значения *pRadius
uint SimpleCircle :: GetRadius( void ) const
{
    return *pRadius;
}
// Перегрузка оператора преинкремента
SimpleCircle SimpleCircle :: operator++( void )
{
    ++(*pRadius);
    return *this;
}
// Перегрузка оператора постинкремента
SimpleCircle SimpleCircle :: operator++( int )
{
    // Создаем локальный объект и инициализируем его текущим объектом
    SimpleCircle
        temp( *this );
    ++(*pRadius);
    return temp;
}
// Перегрузка операции присваивания
SimpleCircle & SimpleCircle :: operator=( const SimpleCircle &right )
{
    if( this != &right )
        *pRadius = right.GetRadius( );
    return *this;
}
// *****
// Тестирование класса
int main( void )          // Возвращает 0 при успехе
{
    // Создаем два объекта
    SimpleCircle
        One,          // Конструктор умолчания
        Two( 9 );    // Обычный конструктор с одним
                     // параметром

    // Инкрементируем объекты
    One++; ++Two;
    // Печатаем значения объектов
    cout << "One: " << One.GetRadius( ) << endl;
    cout << "Two: " << Two.GetRadius( ) << endl;
    // Присваиваем значение одного объекта другому
    One = Two;
}

```

```
// Печатаем значения объектов
cout << "One: " << One.GetRadius( ) << endl;
cout << "Two: " << Two.GetRadius( ) << endl;
return 0;
}
```

П1.2. Глава 2

1. Методы базового и производных классов могут иметь одинаковые имена? Как при этом можно будет различать разные варианты этих методов?

Для объекта некоторого класса, входящего в иерархию классов, будет вызываться метод этого класса. Если в классе такой метод отсутствует, то будет выполнен поиск в нижележащих базовых классах, которые будут просматриваться сверху вниз. При обнаружении первого же метода с заданным именем он и будет вызван.

Чтобы для объекта некоторого класса, в котором есть метод с заданным именем, вызвать метод с тем же именем из нижележащего базового класса, достаточно после операции точка ("."), разделяющей имя объекта и имя функции, вставить имя соответствующего класса и операцию разрешения области видимости (":"). Иллюстрирующий пример см. в *разд. 2.2* в программе из файла `access.cpp`.

2. Наследуются ли данные и методы базового класса в последующие поколения производных классов?

Да. Если последовательно производить ряд классов, последний класс в этом ряду унаследует всю сумму членов-данных и методов предыдущих базовых классов.

3. Пусть класс `Three` произведен от класса `Two`, а класс `Two` произведен от класса `One`. В классе `Two` замещен метод, описанный в классе `One`. Какой вариант этого метода получит класс `Three`?

Так как класс `Three` непосредственно наследуется от класса `Two`, то он получит метод в том виде, в каком он существует в классе `Two`.

4. Можно ли в производном классе описать как `private:` метод, который перед этим был описан в базовом классе как `public:`?

Да, конечно. Такой метод останется закрытым для всех последующих классов, которые будут произведены от него.

5. С какой целью используется служебное слово `protected`?

С использованием служебного слова `protected` объявляются защищенные члены класса. Тем самым они становятся доступными методам производных объектов.

6. Запишите объявление класса `Square` (квадрат), произведенного от класса `Rectangle` (прямоугольник), который, в свою очередь, произведен от класса `Form` (форма).

```
class Square : public Rectangle
{ ... };
```

7. Класс `Square` (квадрат), произведен от класса `Rectangle` (прямоугольник), который, в свою очередь, произведен от класса `Form` (форма). Объект класса `Form` не

использует параметры, объект класса `Rectangle` принимает два параметра целого типа (`length` и `width`), а объект класса `Square` — один параметр (`length`). Напишите конструктор класса `Square`, поместив его определение вне блока класса.

```
Square :: Square( int length ) :  
    Rectangle( length, width )  
{ ... }
```

П1.3. Глава 3

1. Зачем в методе класса используют значения параметров, заданные по умолчанию, если метод можно перегрузить?

Проще иметь дело с одним методом с умалчиваемыми значениями параметров, чем с двумя перегруженными методами. Кроме того, обновление одной версии перегруженного метода без обновления другой часто бывает причиной ошибок в программе.

2. Почему бы тогда всегда не использовать только методы с умалчиваемыми значениями параметров?

Перегрузка методов предоставляет возможности, которые нельзя реализовать, используя только методы с умалчиваемыми значениями параметров. В подобных случаях лучше применять перегрузку методов.

3. Может ли перегруженный метод содержать параметры, заданные по умолчанию?

Да, конечно. Нет никакой причины, по которой не следовало бы использовать это мощное средство.

4. Если вы перегрузили метод класса, то как потом можно будет различать разные варианты методов?

Перегруженными называются методы класса, которые имеют одно и то же имя, но разное количество параметров или их тип.

5. В каких случаях не следует делать методы класса виртуальными?

Появление в классе первого виртуального метода приведет к созданию для него *v*-таблицы, что потребует времени и дополнительной памяти. Кроме того, на обращение к виртуальному методу потребуется также больше времени. Поэтому виртуальные методы следует использовать *только тогда*, когда без них обойтись нельзя. Таким случаем является использование для вызова метода класса указателя на класс, у которого есть базовые и производные классы, имеющие методы с одинаковой сигнатурой.

6. Предположим, что некоторый метод без параметров был описан в базовом классе как виртуальный, а затем перегружен таким образом, чтобы принимать один или два целочисленных параметра. Затем в производном классе был замещен вариант метода с одним целочисленным параметром. Что произойдет, если с помощью указателя, связанного с объектом производного класса, вызвать вариант метода с двумя параметрами?

Замещение в производном классе варианта метода с одним параметром скроет от объектов этого класса все остальные варианты метода. Поэтому в данном случае компилятор покажет сообщение об ошибке.

7. Для чего возиться с абстрактным классом? Не проще ли создать обычный базовый класс, для которого в программе не создавать объекты?

При написании программы всегда следует использовать такие подходы, которые гарантировали бы обнаружение ошибок в программе не во время выполнения программы, а при ее компиляции. Если класс явно будет описан как абстрактный, то любая попытка создать объект этого класса приведет к показу компилятором сообщения об ошибке.

8. Что такое `vptr`?

Указатель на таблицу виртуальных методов `vptr` является элементом выполнения виртуальных методов. Каждый объект класса с виртуальными методами имеет неявный указатель `vptr`, который ссылается на таблицу виртуальных методов для этого класса.

П1.4. Глава 4

1. Обязательно ли использовать пространства имен?

Нет. Простые программы можно писать и без помощи пространства имен. Пространства имен следует использовать при проектировании больших программ коллективом программистов. Стандартное пространство имен `std` используется при работе со средствами стандартной библиотеки языка C++.

2. Каковы отличия между использованием `using namespace` и `using`?

Оператор `using namespace` открывает доступ ко всем идентификаторам соответствующего пространства имен, а спецификатор `using` — доступ только к одному идентификатору. Спецификатор `using` имеет более высокий приоритет по отношению к `using namespace`.

3. Что такое неименованные пространства имен и зачем они нужны?

Неименованными называются пространства имен, для которых не задано собственное имя. Они используются для защиты наборов идентификаторов в разных файлах одной программы от возможных конфликтов имен.

Определение или объявление объекта в пространстве имен без указания имени области равнозначно его описанию как глобального объекта с описателем класса хранения `static` (такой объект имеет описатель класса хранения "внешний статический").

4. Можно ли использовать идентификаторы, объявленные в пространстве имен без применения служебного слова `using`?

Да, имена, объявленные в пространстве имен, можно свободно использовать в программе, если указывать перед ними идентификатор пространства имен и операцию `::`.

5. Что такое стандартное пространство имен `std`?

Данное пространство определено в стандартной библиотеке языка C++ и содержит объявления всех классов и функций стандартной библиотеки.

П1.5. Глава 5

1. Как определить, когда использовать перегруженные операции ввода ("**>>**") и вывода ("**<<**"), а когда другие методы классов потоков?

В целом операторы ввода и вывода проще в использовании, поэтому в большинстве случаев лучше применять их. В некоторых, немногочисленных случаях, когда эти операторы не в состоянии выполнить требуемые действия или эти действия выполняются неэффективно, можно прибегнуть к использованию других методов классов потоков. Примерами таких случаев является ввод пробельных символов или ввод строк из слов, разделенных пробелами.

2. Какие отличия между `cerr` и `clog`?

Объект `cerr` не буферизируется. Это означает, что все данные, поступающие в `cerr`, немедленно выводятся на экран. Это отлично подходит для вывода сообщений об ошибках. Объект `clog` буферизирует свой вывод.

3. Зачем создавать потоки ввода-вывода, если отлично работают семейства функций `scanf()` - `fscanf()`, `printf()` - `fprintf()` - `sprintf()`?

Семейства стандартных функций языка C не в состоянии строго контролировать типы выводимых данных, чего требует стандарт языка C++. Кроме того, эти семейства функций не поддерживают работу с классами.

4. Когда следует применять метод `ignore()`?

Наиболее часто он используется после метода `get()`, выполняющего считывание одного любого символа. Поскольку метод `get()` может оставлять в буфере управляющий символ `'\n'`, то при необходимости его пропустить за методом `get()` вызывается метод `ignore(1, '\n')`.

5. Что такое перегруженный оператор ввода и как он работает?

Перегруженный оператор ввода ("**>>**") является членом объекта `istream` и используется для ввода значений переменных.

6. Что такое перегруженный оператор вывода и как он работает?

Перегруженный оператор вывода ("**<<**") является членом объекта `ostream` и используется для вывода значений переменных в устройства вывода.

7. Какая ширина поля вывода принимается по умолчанию при выводе целых чисел?

Автоматически выбирается минимальная ширина поля вывода, достаточная для изображения числа.

8. Какое значение возвращает перегруженный оператор вывода?

Ссылку на объект `ostream` или `ofstream`.

9. Какой обязательный параметр принимает обычный конструктор объекта `ofstream`?

Имя файла вывода.

10. Что обеспечивает элементарный режим открытия `ios::ate`?

Однократно после открытия файла устанавливает текущую позицию для вывода в конце файла. В дальнейшем вы можете выполнять вывод в любое место файла.

П1.6. Главы 6 и 7

1. Зачем тратить время на программирование исключений? Не лучше ли устранять ошибки по мере их возникновения?

Часто одна и та же ошибка может возникать в разных местах программы, при выполнении разных функций. Использование обработки исключений позволяет собрать коды обработки ошибок в одном месте программы. Кроме того, далеко не всегда можно записать код устранения ошибки в том месте программы, где эта ошибка возникает.

2. Что такое исключение?

Это объект, который создается при использовании служебного слова `throw`. Данный объект является признаком возникновения исключительной ситуации и используется для выбора подходящего обработчика исключения.

3. Для чего нужен блок `try`?

Он определяет фрагмент программы, в котором могут создаваться исключительные ситуации.

4. Для чего используется оператор `catch`?

Оператор `catch` содержит сигнатуру типа исключения, которое он способен обработать. Операторов `catch` может быть несколько и располагаются они друг за другом сразу же за блоком `try`. Они выполняют роль приемников исключения, сгенерированного внутри блока `try`.

5. Какую информацию может содержать исключение?

В общем случае исключение — это объект, способный содержать информацию для передачи ее в обработчик исключения. В качестве исключения можно использовать и объект-класс.

6. Когда создается объект исключения?

Объект исключения создается при вызове оператора `throw`.

7. Что означает оператор `catch(...)`?

Оператор `catch(...)` будет перехватывать исключения любого типа. По этой причине в последовательности обработчиков исключений его следует располагать последним.

8. Для чего нужна операция преобразования типа `const_cast`?

Операция преобразования типа `const_cast` применяется для того, чтобы аннулировать действие модификатора `const`. Необходимость включения в язык операции `const_cast` обусловлена тем, что программист, проектирующий функцию, не обязан описывать не изменяемые в ней параметры как `const`, хотя это и рекомендуется. Правила языка C++ запрещают передачу константного указателя на место обычного. Поэтому операция `const_cast` введена для того, чтобы обойти это ограничение. Естественно, что в подобном случае функция не должна изменять значение, на которое ссылается передаваемый указатель. Если есть возможность добавить к описанию параметра функции модификатор `const`, то это предпочтительнее использования операции `const_cast` при вызове функции.

9. Для чего нужна операция преобразования типа `static_cast`?

Операция преобразования типа `static_cast` выполняется компилятором и должна применяться только в том случае, когда без нее нельзя обойтись. Если же нужное преобразование типа достигается использованием стандартного (неявного) преобразования языка C++, то не следует использовать операцию `static_cast` только для того, чтобы избавиться от предупреждений транслятора. Предупреждения позволяют избежать возможных ошибок, а явное преобразование скрывает возможные ошибки. В качестве примеров целесообразного применения операции `static_cast` можно назвать преобразование указателя `void*` к указателю на необходимый тип или преобразования родственных типов. При выполнении этой операции кодовое представление может быть модифицировано.

10. Для чего нужна операция преобразования типа `dynamic_cast`?

Преобразование `dynamic_cast` позволяет производить безопасные, надежные преобразования типа, причем если преобразуются типы *полиморфных* объектов (полиморфный объект содержит хотя бы один виртуальный метод), то допустимость данного преобразования проверяется при выполнении программы. Преобразование `dynamic_cast` необходимо применять, когда правильность преобразования *невозможно* установить на этапе компиляции. Операцию `dynamic_cast` обычно применяют для полиморфных объектов и для понижающего преобразования виртуальных базовых классов, хотя операция применима и для повышающего или перекрестного преобразования.

11. Для чего нужна операция преобразования типа `reinterpret_cast`?

Преобразование с помощью операции `reinterpret_cast` применяется достаточно редко, когда нужно осуществить "экзотические" преобразования несвязанных между собой типов. В результате такого преобразования получается значение нового типа, состоящее из той же цепочки двоичных разрядов, что и аргумент преобразования. Иными словами, операция `reinterpret_cast` используется для того, чтобы изменить точку зрения компилятора на тип объекта, при этом операция не модифицирует сам объект.

П1.7. Глава 8

1. Программу из *разд. 8.2* в учебных целях модифицируйте таким образом, чтобы в ней использовались не шаблоны классов, а обычные классы. Считайте, что ключ сортировки имеет тип `double`. Для сокращения объема программы в производном классе оставьте только лучшую из простых сортировок (вставками) и лучшую из сложных сортировок (Хоора).

Файл `iofile.h` полностью совпадает с аналогичным файлом, приведенным в *разд. 8.2*. Остальные файлы программы приводятся в листингах П1.1—П1.3.

Листинг П1.1. Файл `SAr_B.H`

```
/*
```

```
Сортировка динамически размещенного массива.
```

```
Используется базовый класс со следующим набором операций:
```

```
- динамическое размещение массива с инициализацией конструктор);
```

- освобождение занятой динамической памяти (деструктор);
- заполнение массива;
- печать содержимого массива.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
// Предотвращение многократного включения данного файла
#ifndef __SAr_B_H
#define __SAr_B_H

// Определение класса для открытия-закрытия файлов на базе
// стандартного класса fstream
#include "iofile.h"
// *****
// Базовый класс для сортировки массивов
class SAr_B // SortArray_Base
{
    // Локальные типы
    struct ELEMENT // Тип элемента массива
    {
        double
            key; // Ключ сортировки
        // Описание других компонентов элемента
    };
    // Данные
protected:
    ELEMENT
        *arr; // Адрес первого элемента массива
    int
        size; // Размер массива
    // Методы
public:
    SAr_B( int s ); // Конструктор
    // Деструктор - подставляемый метод
    ~SAr_B( void )
    {
        if( arr )
        {
            delete [ ] arr; arr = NULL;
        }
    }
    // Заполнение массива значениями, читаемыми из файла на магнитном
    // диске
    void inp_arr( char f_name[ ] );
    // Вывод содержимого массива в файл на магнитном диске
    void print_arr( char f_name[ ], int mode, char text[ ] );
};
// *****

```

```

// Конструктор
SAr_B :: SAr_B(
    int    s )          // Число элементов массива
{
    if( s < 2 )
    {
        cout << "\n In the array there should be 2 or more units "
              << endl;
        exit( 1 );
    }
    arr = new ELEMENT[ s ];
    if( !arr )
    {
        cout << "\n Error of allocation of the array in dynamic"
              " memory " << endl;
        exit( 2 );
    }
    for( int i=0; i<s; i++ )
    {
        arr[ i ].key = 0;
    }
    size = s;
}
// *****
// Заполнение массива значениями, читаемыми из файла на магнитном
// диске
void SAr_B :: inp_arr(
    // Файл ввода
    char  f_name[ ] )
{
    IOFILE f_in;      // Файловый объект для ввода
    // Открытие файла для чтения
    f_in.open_f( f_name, ios::in, 3 );
    // Заполнение массива
    for( int i=0; i<size; i++ )
    {
        f_in >> arr[ i ].key;
        // Обработка ошибки чтения
        if( !f_in )
        {
            cout << endl << " Read error";
            exit( 4 );
        }
    }
}
// Закрытие файла ввода
f_in.close_f( f_name, 5 );

```

```

    return;
}
// *****
// Вывод содержимого массива в файл на магнитном диске
void SAr_B :: print_arr(
    char   f_name[ ], // Файл вывода
    int    mode,      // Режим открытия файла
    char   text[ ] ) // Заголовок для печати
{
    IOFILE f_out;      // Файловый объект для вывода
    // Открытие файла для записи
    f_out.open_f( f_name, mode, 6 );
    f_out << endl << text;
    for( int i=0; i < size; i++ )
    {
        if( ( i%4 ) == 0 )
            f_out << endl;
        f_out.width( 14 ); f_out << arr[ i ].key;
    }
    // Закрытие файла вывода
    f_out.close_f( f_name, 7 );
    return;
}
#endif

```

Листинг П1.2. Файл SAr_D.H

```

/*
    Сортировка динамически размещенного массива.
    Используется производный класс от базового класса, определенного в файле
    SAr_B.h, со следующим набором операций:
    - простая сортировка массива вставками;
    - рекурсивная сортировка Хоора.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Предотвращение многократного включения данного файла
#ifndef __SAr_D_H
#define __SAr_D_H
    // Базовый класс для сортировки массива
    #include "SAr_B.h"
    // Размер стека отложенных сегментов: (log2(s)+1), где s - размер
    // массива
    const int M = 16;
    struct STACK          // Тип элемента стека
    {
        int    l;          // Левая граница сегмента

```

```

    int    r;           // Правая граница сегмента
};
// *****
// Производный класс для сортировки массивов
class SAR_D : public SAR_B
{
    // Методы
public:
    // Конструктор - обратите внимание на механизм передачи параметра
    //   из конструктора производного класса конструктору базового
    //   класса
    SAR_D( int s ) : SAR_B( s ){ }
    // Сортировка простыми включениями - по неубыванию
    void insertsor( void );
    // Быстрая сортировка массива - нерекурсивный вариант
    void quicksort( void );
private:
    // Занесение в стек сегментов (для сортировки Хоора)
    void push( int left, int right, STACK s[ M ], int &sp );
    // Извлечение сегмента из стека (для сортировки Хоора)
    void pop( int &l, int &r, STACK s[ M ], int &sp );
};
// *****
// Сортировка простыми включениями - по неубыванию: сортируются
//   элементы массива с индексами от 1 до size-1, элемент с индексом
//   0 используется в качестве вспомогательного (левый "барьер")
void SAR_D :: insertsor( void )
{
    int    i,           // Индекс вставляемого элемента
           j;           // Индекс элемента в упорядоченном сегменте
    ELEMENT
    copy;   // copy = arr[ i ]
    // Перебор вставляемых элементов
    for( i=2; i < size; i++ )
    {
        copy = arr[ i ];
        // Установка "барьера"
        arr[ 0 ] = copy;
        // Вставка copy на нужное место
        j = i - 1;
        while( copy.key < arr[ j ].key )
        {
            // Сдвиг
            arr[ j+1 ] = arr[ j ]; j--;
        }
    }
}

```

```

        arr[ j+1 ] = copy;
    }
    return;
}
// *****
// Быстрая сортировка Хоора - рекурсивный вариант
// -----
// Занесение в стек сегментов
void SAR_D :: push(
    int    left,      // Левая граница сегмента
    int    right,     // Правая граница сегмента
    STACK s[ M ],    // Стек границ сегментов
    int    &sp )      // sp - указатель вершины стека
{
    if( ( right-left) >= 1 )
    {
        sp++; s[ sp ].l = left; s[ sp ].r = right;
    }
    return;
}
// -----
// Извлечение сегмента из стека
void SAR_D :: pop(
    int    &l,         // Указатель на левую границу сегмента
    int    &r,         // Указатель на правую границу сегмента
    STACK s[ M ],    // Стек границ сегментов
    int    &sp )      // sp - указатель вершины стека
{
    l = s[ sp ].l; r = s[ sp ].r; sp--;
    return;
}
// -----
// Быстрая сортировка массива - рекурсивный вариант
void SAR_D :: quicksort( void )
{
    int    left,      // Левая граница разделяемого сегмента
           right,     // Правая граница разделяемого сегмента
           i,         // Индекс кандидата на обмен слева - направо
           j;         // Индекс кандидата на обмен справа - налево
    ELEMENT
           median,    // Медиана разделяемого сегмента
           copy;      // Для перестановки кандидатов
    STACK s[ M ];    // Стек границ сегментов
    int    sp;       // Указатель вершины стека
    sp = -1;         // Стек пуст

```

```

// sp >= 0 => стек не пуст
push( 0, size-1, s, sp );
while( sp >= 0 )
{
    // Подготовка верхнего сегмента из стека для разделения
    pop( left, right, s, sp );
    median = arr[ ( left + right )/2 ];
    i = left; j = right;
    // Разделение текущего сегмента
    while( i <= j )
    {
        // Найти кандидата на обмен слева
        while( arr[ i ].key < median.key )
            i++;
        // Найти кандидата на обмен справа
        while( median.key < arr[ j ].key )
            j--;
        // Обмен, если кандидаты находятся в разных подсегментах
        if( i <= j )
        {
            copy = arr[ i ]; arr[ i ] = arr[ j ];
            arr[ j ] = copy; i++; j--;
        }
    }
    // Поместить в стек сначала более длинный подсегмент, а затем -
    // более короткий
    if( ( j-left ) < ( right-i ) )
    {
        // Левый подсегмент - короче
        push( i, right, s, sp );
        push( left, j, s, sp );
    }
    else
    {
        // Правый подсегмент - короче
        push( left, j, s, sp );
        push( i, right, s, sp );
    }
}
return;
}
#endif

```

Листинг П1.3. Файл SAr.CPP

/*

Сортировка массивов с использованием базового SAr_B и производного SAr_D классов, определение которых приведено во включаемых файлах SAr_B.H и SAr_D.H.

Для открытия-закрытия файлов в базовом классе SAr_B используется класс IOFILE, определение которого приведено во включаемом файле IOFILE.H.

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SAr_D.h"          // Класс для сортировки массивов
    // *****
// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    SAr_D      ar( 8 ); // Создаем массив из 8 элементов
    // Заполняем сортируемый массив
    ar.inp_arr( "sort_arr.dat" );
    // Печатаем сортируемый массив
    ar.print_arr( "sort_arr.out", ios::out,
        "\nThe array before sorting:" );
    ar.insertsort( ); // Сортируем массив простыми включениями
    // Печатаем отсортированный массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
        " The array after sorting by simple inclusions "
        "\n (first unit - auxiliary, is not sorted):");
    // Заполняем сортируемый массив
    ar.inp_arr( "sort_arr.dat" );
    // Печатаем сортируемый массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
        "\nThe array before sorting:" );
    // Нерекурсивная быстрая сортировка Хоора
    ar.quicksort( );
    // Печатаем отсортированный массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
        " The array after fast not of recursive"
        " sorting Hoore:" );
    return 0;
}

```

2. Программу из упражнения 1 модифицируйте таким образом, чтобы объявления классов размещались в заголовочных файлах, а реализация методов классов — в файлах с расширением `сpp`. Сказанное не относится к классу `IOFILE`.

Файл `iofile.h` полностью совпадает с аналогичным файлом, приведенным в *разд. 8.2*. Остальные файлы программы приводятся в листингах П1.4—П1.8.

Листинг П1.4. Файл SAr_B.H

```

/*
Сортировка динамически размещенного массива.
Объявление базового класса со следующим набором операций:
- динамическое размещение массива с инициализацией (конструктор);

```

- освобождение занятой динамической памяти (деструктор);
- заполнение массива;
- печать содержимого массива.

Реализация методов базового класса приведена в файле SAR_B.CPP.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
// Предотвращение многократного включения данного файла
#ifndef __SAr_B_H
#define __SAr_B_H

#include <stdlib.h> // Для NULL
// *****
// Базовый класс для сортировки массивов
class SAR_B
{
    // Локальные типы
    struct ELEMENT // Тип элемента массива
    {
        double
            key; // Ключ сортировки
        // Описание других компонентов элемента
    };
    // Данные
protected:
    ELEMENT
        *arr; // Адрес первого элемента массива
    int
        size; // Размер массива
    // Методы
public:
    SAR_B( int s ); // Конструктор
    // Деструктор - подставляемый метод
    ~SAR_B( void )
    {
        if( arr )
        {
            delete [ ] arr; arr = NULL;
        }
    }
    // Заполнение массива значениями, читаемыми из файла на магнитном
    // диске
    void inp_arr( char f_name[ ] );
    // Вывод содержимого массива в файл на магнитном диске
    void print_arr( char f_name[ ], int mode, char text[ ] );
};
#endif

```

Листинг П1.5. Файл SAr_B.CPP

```

/*
    Реализация методов базового класса, объявление которого приведено в файле SAr_B.H.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SAr_B.H"          // Объявление базового класса
// Определение класса для открытия-закрытия файлов на базе стандартного
// класса fstream. Класс обеспечивает также возможность использования
// перегруженных операций "<<" и ">>" для стандартных типов
#include "iofile.h"
// *****
// Конструктор
SAr_B :: SAr_B(
    int          s )          // Число элементов массива
{
    if( s < 2 )
    {
        cout << "\n In the array there should be 2 or more units "
              << endl;
        exit( 1 );
    }
    arr = new ELEMENT[ s ];
    if( !arr )
    {
        cout << "\n Error of allocation of the array in dynamic memory "
              << endl;
        exit( 2 );
    }
    for( int i=0; i<s; i++ )
    {
        arr[ i ].key = 0;
    }
    size = s;
}
// *****
// Заполнение массива значениями, читаемыми из файла на магнитном диске
void SAr_B :: inp_arr(
    // Файл ввода
    char          f_name[ ] )
{
    IOFILE        f_in;       // Файловый объект для ввода
    // Открытие файла для чтения
    f_in.open_f( f_name, ios::in, 3 );
    // Заполнение массива

```

```

for( int i=0; i<size; i++ )
{
    f_in >> arr[ i ].key;
    // Обработка ошибки чтения
    if( !f_in )
    {
        cout << endl << " Read error";
        exit( 4 );
    }
}
// Закрытие файла ввода
f_in.close_f( f_name, 5 );
return;
}
// *****
// Вывод содержимого массива в файл на магнитном диске
void SAr_B :: print_arr(
    char      f_name[ ], // Файл вывода
    int       mode,      // Режим открытия файла
    char      text[ ] ) // Заголовок для печати
{
    IOFILE     f_out;    // Файловый объект для вывода
    // Открытие файла для записи
    f_out.open_f( f_name, mode, 6 );
    f_out << endl << text;
    for( int i=0; i < size; i++ )
    {
        if( ( i%4 ) == 0 )
            f_out << endl;
        f_out.width( 14 ); f_out << arr[ i ].key;
    }
    // Закрытие файла вывода
    f_out.close_f( f_name, 7 );
    return;
}

```

Листинг П1.6. Файл SAr_D.H

/*

Сортировка динамически размещенного массива.

Объявляется производный класс от базового класса, определенного в файле SAr_B.h, со следующим набором операций:

- простая сортировка массива вставками;
- нерекурсивная сортировка Хоора.

Реализация методов производного класса приведена в файле SAr_D.CPP.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
// Предотвращение многократного включения данного файла
#ifndef __SAr_D_H
#define __SAr_D_H
    // Базовый класс для сортировки массива
    #include "SAr_B.h"
    // Размер стека отложенных сегментов: (log2(s)+1), где s - размер
    // массива
    const int M = 16;
    struct STACK          // Тип элемента стека
    {
        int    l;          // Левая граница сегмента
        int    r;          // Правая граница сегмента
    };
    // *****
    // Производный класс для сортировки массивов
    class SAr_D : public SAr_B
    {
        // Методы
    public:
        // Конструктор - обратите внимание на механизм передачи параметра
        // из конструктора производного класса конструктору базового
        // класса
        SAr_D( int s ) : SAr_B( s ){ }
        // Сортировка простыми включениями - по неубыванию
        void insertsort( void );
        // Быстрая сортировка массива - рекурсивный вариант
        void quicksort( void );
    private:
        // Занесение в стек сегментов (для сортировки Хоора)
        void push( int left, int right, STACK s[ M ], int &sp );
        // Извлечение сегмента из стека (для сортировки Хоора)
        void pop( int &l, int &r, STACK s[ M ], int &sp );
    };
    // Конец определения класса
#endif

```

Листинг П1.7. Файл SAr_D.CPP

```

/*
    Реализация методов производного класса, объявление которого приведено
    в файле SAr_D.H.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SAr_D.H"          // Объявление производного класса
// *****

```

```

// Сортировка простыми включениями - по неубыванию: сортируются элементы
// массива с индексами от 1 до size-1, элемент с индексом 0
// используется в качестве вспомогательного (левый "барьер")
void SAR_D :: insertsort( void )
{
    int      i,          // Индекс вставляемого элемента
            j;          // Индекс элемента в упорядоченном сегменте
    ELEMENT   copy;      // copy = arr[ i ]
    // Перебор вставляемых элементов
    for( i=2; i < size; i++ )
    {
        copy = arr[ i ];
        // Установка "барьера"
        arr[ 0 ] = copy;
        // Вставка copy на нужное место
        j = i -1;
        while( copy.key < arr[ j ].key )
        {
            // Сдвиг
            arr[ j+1 ] = arr[ j ]; j--;
        }
        arr[ j+1 ] = copy;
    }
    return;
}

// *****
// Быстрая сортировка Хоора - нерекурсивный вариант
// -----
// Занесение в стек сегментов
void SAR_D :: push(
    int      left,      // Левая граница сегмента
    int      right,     // Правая граница сегмента
    STACK    s[ M ],    // Стек границ сегментов
    int      &sp )      // sp - указатель вершины стека
{
    if( ( right-left) >= 1 )
    {
        sp++; s[ sp ].l = left; s[ sp ].r = right;
    }
    return;
}

// -----
// Извлечение сегмента из стека
void SAR_D :: pop(
    int      &l,        // Указатель на левую границу сегмента

```

```

    int      &r,          // Указатель на правую границу сегмента
    STACK    s[ M ],     // Стек границ сегментов
    int      &sp )       // sp - указатель вершины стека
{
    l = s[ sp ].l; r = s[ sp ].r; sp--;
    return;
}
// -----
// Быстрая сортировка массива - рекурсивный вариант
void SAR_D :: quicksort( void )
{
    int      left,       // Левая граница разделяемого сегмента
            right,      // Правая граница разделяемого сегмента
            i,          // Индекс кандидата на обмен слева - направо
            j;          // Индекс кандидата на обмен справа - налево
    ELEMENT  median,     // Медиана разделяемого сегмента
            copy;        // Для перестановки кандидатов
    STACK    s[ M ];     // Стек границ сегментов
    int      sp;         // Указатель вершины стека
    sp = -1;            // Стек пуст
    // sp >= 0 => стек не пуст
    push( 0, size-1, s, sp );
    while( sp >= 0 )
    {
        // Подготовка верхнего сегмента из стека для деления
        pop( left, right, s, sp );
        median = arr[ ( left + right )/2 ];
        i = left; j = right;
        // Разделение текущего сегмента
        while( i <= j )
        {
            // Найти кандидата на обмен слева
            while( arr[ i ].key < median.key )
                i++;
            // Найти кандидата на обмен справа
            while( median.key < arr[ j ].key )
                j--;
            // Обмен, если кандидаты находятся в разных подсегментах
            if( i <= j )
            {
                copy = arr[ i ]; arr[ i ] = arr[ j ];
                arr[ j ] = copy; i++; j--;
            }
        }
        // Поместить в стек сначала более длинный подсегмент, а затем -
        // более короткий

```

```

    if( ( j-left ) < ( right-i ) )
    {
        // Левый подсегмент - короче
        push( i, right, s, sp );
        push( left, j, s, sp );
    }
    else
    {
        // Правый подсегмент - короче
        push( left, j, s, sp );
        push( i, right, s, sp );
    }
}
return;
}

```

Листинг П1.8. Файл SAR.CPP

/*
 Сортировка массивов с использованием базового SAR_B и производного SAR_D классов, объявления которых приведено во включаемых файлах SAR_B.H, SAR_D.H, а реализация методов - в файлах SAR_B.CPP и SAR_D.CPP. Для открытия-закрытия файлов в базовом классе SAR_B используется класс IOFILE, определение которого приведено во включаемом файле IOFILE.H.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
#include <iostream.h> // Для потокового ввода-вывода
#include "SAR_D.h"    // Класс для сортировки массивов
// *****
// Тестирование
int main( void )      // Возвращает 0 при успехе
{
    SAR_D      ar( 8 ); // Создаем массив из 8 элементов
    // Заполняем сортируемый массив
    ar.inp_arr( "sort_arr.dat" );
    // Печатаем сортируемый массив
    ar.print_arr( "sort_arr.out", ios::out,
        "\nThe array before sorting:" );
    ar.insertsort(); // Сортируем массив простыми включениями
    // Печатаем отсортированный массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
        "The array after sorting by simple inclusions "
        "\n (first unit - auxiliary, is not sorted):");
    // Заполняем сортируемый массив
    ar.inp_arr( "sort_arr.dat" );
    // Печатаем сортируемый массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
        "\nThe array before sorting:" );
    // Нерекурсивная быстрая сортировка Хоора

```

```

ar.quicksort( );
// Печатаем отсортированный массив
ar.print_arr( "sort_arr.out", ios::out | ios::app,
              " The array after fast not of recursive"
              " sorting Hoor:" );

return 0;
}

```

3. Программу из *разд. 8.2* в учебных целях модифицируйте таким образом, чтобы в ней использовались не шаблоны классов, а обычные классы. Считайте, что ключ сортировки имеет тип `long`. Для сокращения объема программы в производном классе оставьте только лучшую из простых сортировок (вставками) и лучшую из сложных сортировок (Хоора). Нерекursивный вариант сортировки Хоора замените рекурсивным, подробно описанным и реализованным в [3].

В соответствии с [3], в рекурсивной сортировке Хоора вместо использования стека отложенных сегментов применяется рекурсивный метод `split()`, выполняющий разделение исходного сегмента на два подсегмента. Этот метод является вспомогательным и должен быть закрытым (`private`).

Файл `iofile.h` полностью совпадает с аналогичным файлом, приведенным в *разд. 8.2*. Файл `SAr_B.H` полностью совпадает с аналогичным файлом, приведенным в ответе на упражнение 1. Остальные файлы программы приводятся в листингах П1.9 и П1.10.

Листинг П1.9. Файл `SAr_D.H`

```

/*
    Сортировка динамически размещенного массива.

    Используется производный класс от базового класса, определенного в файле
    SAr_B.h, со следующим набором операций:
    - простая сортировка массива вставками;
    - рекурсивная сортировка Хоора.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Предотвращение многократного включения данного файла
#ifndef __SAr_D_H
#define __SAr_D_H

    // Базовый класс для сортировки массива
    #include "SAr_B.h"
    // *****
    // Производный класс для сортировки массивов
    class SAr_D : public SAr_B
    {
        // Данные
    private:
        // Эти объекты определяются здесь, чтобы при рекурсивных вызовах
        // метода split( ) можно было использовать каждый из них

```

```

// в одном экземпляре
ELEMENT
    copy,      // Копия элемента массива
    median;    // Медиана сегмента
int    i,      // Индекс кандидата на обмен слева
        j;      // Индекс кандидата на обмен справа

// Методы
public:
    // Конструктор - обратите внимание на механизм передачи параметра
    // из конструктора производного класса конструктору базового
    // класса
    SAr_D( int s ) : SAr_B( s ){ }
    // Сортировка простыми включениями - по неубыванию
    void insertsort( void );
    // Быстрая сортировка массива - рекурсивный вариант
    void quicksort1( void );

private:
    // Рекурсивный метод для разделения сегментов
    void split( int left, int right );
};      // Конец определения класса
// *****
// Сортировка простыми включениями - по неубыванию: сортируются
// элементы массива с индексами от 1 до size-1, элемент с индексом
// 0 используется в качестве вспомогательного (левый "барьер")
void SAr_D :: insertsort( void )
{
    int    i,      // Индекс вставляемого элемента
        j;      // Индекс элемента в упорядоченном сегменте
    ELEMENT
        copy;     // copy = arr[ i ]
    // Перебор вставляемых элементов
    for( i=2; i < size; i++ )
    {
        copy = arr[ i ];
        // Установка "барьера"
        arr[ 0 ] = copy;
        // Вставка copy на нужное место
        j = i -1;
        while( copy.key < arr[ j ].key )
        {
            // Сдвиг
            arr[ j+1 ] = arr[ j ]; j--;
        }
        arr[ j+1 ] = copy;
    }
}

```

```

    return;
}
// *****
// Быстрая сортировка Хоора - рекурсивный вариант
// -----
// Быстрая сортировка массива - рекурсивный вариант
void SAR_D :: quicksort1( void )
{
    split( 0, size-1 );
    return;
}
// -----
// Рекурсивный метод для разделения сегментов
void SAR_D :: split(
    int    left,      // Левая граница сегмента
    int    right )    // Правая граница сегмента

{
    // Выход из рекурсии
    if( ( right-left ) <= 0 )
        return;

    // Подготовка к разделению текущего сегмента
    median = arr[ ( left + right )/2 ];
    i = left; j = right;
    // Разделение текущего сегмента
    while( i <= j )
    {
        // Найти кандидата на обмен слева
        while( arr[ i ].key < median.key )
            i++;
        // Найти кандидата на обмен справа
        while( median.key < arr[ j ].key )
            j--;
        // Обмен, если кандидаты находятся в разных подсегментах
        if( i <= j )
        {
            copy = arr[ i ]; arr[ i ] = arr[ j ];
            arr[ j ] = copy; i++; j--;
        }
    }
    // Финал - разделить сначала более короткий подсегмент, а затем -
    // более длинный
    if( ( j-left ) < ( right-i ) )
    {
        // Левый подсегмент - короче
        split( left, j ); split( i, right );
    }
}

```

```

    }
    else
    {
        // Правый подсегмент - короче
        split( i, right ); split( left, j );
    }
    return;
}
#endif

```

Листинг П1.10. Файл SAr.CPP

```

/*
Сортировка массивов с использованием базового SAr_B и производного SAr_D
классов, определение которых приведено во включаемых файлах SAr_B.H и SAr_D.H.
Для открытия-закрытия файлов в базовом классе SAr_B используется класс IOFILE,
определение которого приведено во включаемом файле IOFILE.H.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SAr_D.h"      // Класс для сортировки массивов
// *****
// Тестирование
int main( void )       // Возвращает 0 при успехе
{
    SAr_D      ar( 8 ); // Создаем массив из 8 элементов
    // Заполняем сортируемый массив
    ar.inp_arr( "sort_arr.dat" );
    // Печатаем сортируемый массив
    ar.print_arr( "sort_arr.out", ios::out,
                  "\nThe array before sorting:" );
    ar.insertsort( );   // Сортируем массив простыми включениями
    // Печатаем отсортированный массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
                  " The array after sorting by simple inclusions "
                  "\n (first unit - auxiliary, is not sorted):");
    // Заполняем сортируемый массив
    ar.inp_arr( "sort_arr.dat" );
    // Печатаем сортируемый массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
                  "\nThe array before sorting:" );
    // Рекурсивная быстрая сортировка Хоора
    ar.quicksort1( );
    // Печатаем отсортированный массив
    ar.print_arr( "sort_arr.out", ios::out | ios::app,
                  " The array after fast of recursive sorting Hoor:" );
    return 0;
}

```

П1.8. Глава 9

1. Программу из *разд. 9.2* модифицируйте таким образом, чтобы объявления классов размещались в заголовочных файлах, а реализация методов классов — в файлах с расширением `cpp`.

Файл `TestGr.cpp` приведен в *разд. 9.2*. Остальные файлы программного проекта представлены в листингах П1.11—П1.16.

Листинг П1.11. Файл `IOFile.h`

```
/*
    Открытие-закрытие файлов на базе класса fstream.
    Использование перегруженных операций вывода (<<) и ввода (>>) для стандартных
    типов. Объявление класса.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Предотвращение многократного включения данного файла
#ifdef __IOFILE_H
#define __IOFILE_H
    #include <iostream>    // Для потокового ввода-вывода
    #include <fstream>     // Для работы с файлами
    using namespace      std;    // Используем стандартное
                                // пространство имен
    // *****
    // Класс для открытия-закрытия файлов на базе класса fstream
    class IOFILE : public fstream
    {
        // Методы
    public:
        void open_f( char *pFileName, int mode, int error_num );
        void close_f( char *pFileName, int error_num );
    };
#endif
```

Листинг П1.12. Файл `IOFile.cpp`

```
/*
    Открытие-закрытие файлов на базе класса fstream.
    Использование перегруженных операций вывода (<<) и ввода (>>) для стандартных
    типов. Реализация методов класса IOFILE, объявление которого приведено в файле
    IOFile.h.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Объявление класса для открытия-закрытия файлов на базе стандартного
// класса fstream и подключение перегруженных операций << и >>
#include "IOFile.h"
// *****
```

```

// Открытие файла
void IOFILE :: open_f(
    char *pFileName,    // Указатель на строку - имя файла
    int mode,            // Режим доступа: 1-63
    int error_num )      // Код ошибки
{
    // Проверка допустимости режима
    if( ( mode < 1 ) || ( mode > 63 ) )
    {
        cout << " Error " << error_num << ". Error of the mode of "
              << " access to the file: mode = " << mode << " (the"
              << " values are admitted 1..63)" << endl;
        exit( error_num );
    }
    // Открытие файла
    open( pFileName, mode );
    if( fail( ) )
    {
        cout << " Error " << error_num << ". The open error of the file "
              << pFileName << " with access " << mode << endl;
        exit( error_num );
    }
    return;
}
// *****
// Закрытие файла
void IOFILE :: close_f(
    char *pFileName,    // Указатель на строку - имя файла
    int error_num )      // Код ошибки
{
    // Закрытие файла
    close( );
    if( fail( ) )
    {
        cout << " Error " << error_num << ". Error of file closing "
              << pFileName << endl;
        exit( error_num );
    }
    return;
}
}

```

Листинг П1.13. Файл Graph_V.h

```
/*
```

Объявление базового класса для объекта графа со следующим набором операций:
 - динамическое размещение данных о графе (конструктор);

- освобождение занятой динамической памяти (деструктор);
- чтение информации о графе;
- печать информации о графе.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
// Предотвращение многократного включения данного файла
#ifndef __GRAPH_B_H
#define __GRAPH_B_H
    // Объявление класса для открытия-закрытия файлов на базе
    // стандартного класса fstream и подключение перегруженных операций
    // << и >>
#include "IOFile.h"
    struct A          // Arc: дуга (ребро) графа
    {
        int    first;    // 1-я вершина ребра
        int    last;     // 2-я вершина ребра
        float  weight;   // Вес ребра
    };
    // *****
    // Базовый класс для объекта графа
    class GR
    {
        // Данные

    protected:
        int    NumTop,    // Число вершин
                NumArc,   // Число ребер
                start,     // Вершина - старт пути
                finish;    // Вершина - финиш пути
        // Адрес первого элемента массива структур с информацией о ребрах
        // графа
        A      *pArc;
        // Методы
    public:
        // Конструктор
        GR( int nt, int na );
        ~GR( void )      // Деструктор: подставляемый метод
        {
            if( pArc )
            {
                delete [ ] pArc; pArc = NULL;
            }
        }
        // Ввод информации о графе из файла на магнитном диске
        void ReadGraph( char f_name[ ] );

```

```

        // Вывод информации о графе в файл на магнитном диске
        void WriteGraph( char f_name[ ], int mode, char text[ ] );
    };
    // Конец объявления класса
#endif

```

Листинг П1.14. Файл Graph_B.cpp

```

/*
    Базовый класс для объекта графа со следующим набором операций:
    - динамическое размещение данных о графе (конструктор);
    - освобождение занятой динамической памяти (деструктор);
    - чтение информации о графе;
    - печать информации о графе.
    Реализация методов базового класса, объявление которого находится в файле
    Graph_B.h.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "Graph_B.h"    // Объявление базового класса
//*****
// Конструктор
GR :: GR(
    int      nt,        // Число вершин графа
    int      na )       // Число ребер графа
{
    // Проверка области допустимых значений nt и na
    if( nt < 2 )
    {
        cout << "\n Error 10. In the graph there should be 2 or more"
              << " top " << endl;
        exit( 10 );
    }
    if( na < 1 )
    {
        cout << "\n Error 20. In the graph there should be 1 or more"
              << " edges " << endl;
        exit( 20 );
    }
    // Размещаем массив ребер в динамической памяти
    pArc = new A[ na ];
    if( !pArc )
    {
        cout << "\n Error 30. The information on edges of a graph was"
              << " not placed in dynamic memory " << endl;
        exit( 30 );
    }
    // Инициализация данных

```

```

    for( int i = 0; i < na; i++ )
    {
        pArc[ i ].first = 0; pArc[ i ].last = 0;
        pArc[ i ].weight = 0.0f;
    }
    NumTop = nt; NumArc = na;
}
//*****
// Ввод информации о графе из файла на магнитном диске
void GR :: ReadGraph(
    // Файл ввода
    char        f_name[ ] )
{
    IOFILE      f_in;        // Файловый объект для ввода
    // Открытие файла для чтения
    f_in.open_f( f_name, ios::in, 50 );
    // Чтение данных о графе
    f_in >> start >> finish;
    if( !f_in )
    {
        cout << "\n Error 60. A read error from the file" << f_name
              << endl;
        exit( 60 );
    }
    for( int i=0; i<NumArc; i++ )
    {
        f_in >> pArc[ i ].first >> pArc[ i ].last >> pArc[ i ].weight;
        if( !f_in )
        {
            cout << "\n Error 70. A read error from the file" << f_name
                  << endl;
            exit( 70 );
        }
    }
    // Закрытие файла ввода
    f_in.close_f( f_name, 80 );
    return;
}
// *****
// Вывод информации о графе в файл на магнитном диске
void GR :: WriteGraph(
    char        f_name[ ], // Файл вывода
    int         mode,      // Режим открытия файла
    char        text[ ] ) // Заголовок для печати
{

```

```

IOFILE      f_out;      // Файловый объект для вывода
// Открытие файла для вывода
f_out.open_f( f_name, mode, 90 );
// Вывод информации о графе
f_out << text << endl << "          Число вершин графа: " << NumTop <<
    " , число ребер: " << NumArc << endl << endl <<
    " Индекс ребра   1 вершина   2 вершина           Вес ребра ";
for( int i=0; i<NumArc; i++ )
{
    f_out << endl; f_out.width( 8 );
    f_out << i; f_out.width( 13 );
    f_out << pArc[ i ].first; f_out.width( 13 );
    f_out << pArc[ i ].last;
    f_out.width( 13 ); f_out << pArc[ i ].weight;
}
// Закрытие файла вывода
f_out.close_f( f_name, 100 );
return;
}

```

Листинг П1.15. Файл Graph_D.h

```

/*
    Задача коммивояжера - нахождение наилучшего пути из заданной вершины графа
    (из заданного города) в другую заданную вершину (в другой заданный город).
    Объявление класса, производного от базового класса (см. файл Graph_B.h) со
    следующим набором операций:
    - поиск оптимального пути в неориентированном взвешенном графе;
    - выполнение шага вперед из достигнутой вершины по заданному ребру;
    - прохождение пути, если он есть, от достигнутой вершины до конечной вершины;
    - печать информации о наилучшем пути между заданными вершинами.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Предотвращение многократного включения данного файла
#ifndef __GRAPH_D_H
#define __GRAPH_D_H
    // Подключение базового класса
    #include "Graph_B.h"
    struct W          // Way: путь до одной вершины
    {
        int    exist;    // ( != 0 ) - путь имеется
        int    ref;      // REFerence: предыдущая вершина, через которую
                        //   проходит путь
        float  SumDist;  // Суммарная длина минимального пути
    };
    /**/

```

```

// Производный класс для решения задачи коммивояжера: Graph for the
// Commercial Traveller
class GR_CT : public GR
{
    // Данные
private:
    // Адрес первого элемента массива структур с информацией
    // о минимальном пути между заданными вершинами
    W      *pMinWay;
    // Следующие далее данные определены здесь для экономии памяти
    // стека, так как они используются в одном экземпляре
    // в рекурсивной функции PassWay
    int     one,          // 1-я вершина текущего ребра
           two;          // 2-я вершина текущего ребра

    // Методы
public:
    // Конструктор
    GR_CT( int nt, int na );
    ~GR_CT( void )        // Подставляемый деструктор
    {
        if( pMinWay )
        {
            delete [ ] pMinWay; pMinWay = NULL;
        }
    }
    // Поиск оптимального пути в неориентированном взвешенном графе
    void solution( void );
    // Печать информации о наилучшем пути от start до finish
    void OutRes( char f_name[ ], int mode, char text[ ] );
private:
    // Прохождение пути от достигнутой вершины InterMediate, если он
    // есть, до вершины finish
    void PassWay( int InterMediate );
    // Шаг вперед из достигнутой вершины по заданному ребру
    void ForStep( int top1, int IndArc, int top2 );
};
#endif

```

Листинг П1.16. Файл Graph_D.cpp

```

/*
Задача коммивояжера - нахождение наилучшего пути из заданной вершины графа
(из заданного города) в другую заданную вершину (в другой заданный город).

Реализация методов класса, производного от базового класса (см. файл
Graph_B.h) со следующим набором операций:
- поиск оптимального пути в неориентированном взвешенном графе;

```

- выполнение шага вперед из достигнутой вершины по заданному ребру;
- прохождение пути, если он есть, от достигнутой вершины до конечной вершины;
- печать информации о наилучшем пути между заданными вершинами.

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
#include "Graph_D.h"      // Объявление производного класса
/*****
// Конструктор
GR_CT :: GR_CT (
    int          nt,        // Число вершин графа
    int          na )       // Число ребер графа
: GR( nt, na )
{
    // Размещаем массив структур с информацией о наилучшем пути
    // в динамической памяти
    pMinWay = new W[ nt ];
    if( !pMinWay )
    {
        cout << "\n Error 40. The information on the best path was"
              " not placed in dynamic memory " << endl;
        exit( 40 );
    }
    // Инициализация данных
    for( int i = 0; i < nt; i++ )
    {
        pMinWay[ i ].exist = 0; pMinWay[ i ].ref = 0;
        pMinWay[ i ].SumDist = 0.0f;
    }
}
/*****
// Поиск оптимального пути в неориентированном взвешенном графе
void GR_CT :: solution( void )
{
    // Подготовка
    pMinWay[ start ].exist = 1;
    pMinWay[ start ].ref = -1;
    PassWay( start );
    return;
}
//-----
// Прохождение пути от достигнутой вершины InterMediate, если он есть, до
// вершины finish
void GR_CT :: PassWay(
    // Достигнутая вершина - отправная точка пути
    int          InterMediate )
{

```

```

    int        k;           // Индекс текущей дуги графа
    if( InterMediate == finish )
        return;           // !!! Выход из рекурсии
    for( k = 0; k < NumArc; k++ )
    {
        // Перебор ребер графа
        one = pArc[ k ].first; two = pArc[ k ].last;
        if( one == InterMediate )
            ForStep( one, k, two );
        else if( two == InterMediate )
            ForStep( two, k, one );
    }
    return;                // Альтернативный вариант выхода из рекурсии
}
//-----
// Шаг вперед из достигнутой вершины по заданному ребру
void GR_CT :: ForStep(
    int        top1,        // Достигнутая вершина, из
                          // которой шагаем вперед
    int        IndArc,      // Индекс ребра, по которому
                          // делается шаг вперед
    int        top2 )      // Вершина на конце ребра
{
    float      NewDist;     // Расстояние до top2 по пути через top1
    NewDist = pMinWay[ top1 ].SumDist + pArc[ IndArc ].weight;
    if( !pMinWay[ top2 ].exist )
    {
        // Пока пути до top2 нет
        pMinWay[ top2 ].exist = 1;
        pMinWay[ top2 ].SumDist = NewDist;
        pMinWay[ top2 ].ref = top1; PassWay( top2 );
    }
    else
    {
        // Путь до top2 существует
        if( pMinWay[ top2 ].SumDist > NewDist )
        {
            // Новый путь короче
            pMinWay[ top2 ].SumDist = NewDist;
            pMinWay[ top2 ].ref = top1;
            PassWay( top2 );
        }
    }
    return;
}
//*****
// Печать информации о наилучшем пути от start до finish
void GR_CT :: OutRes(
    char       f_name[ ], // Файл вывода
    int        mode,      // Режим открытия файла

```

```

char      text[ ] ) // Заголовок для печати
{
    IOFILE      f_out;    // Файловый объект для вывода
    // Открытие файла для вывода
    f_out.open_f( f_name, mode, 110 );
    // Данные для инвертирования списка ссылок
    int          TempTop,  // Текущая вершина пути
                pred,     // Предыдущая вершина пути
                next;      // Следующая вершина пути
    if( !pMinWay[finish]. exist )
    {
        cout << "\n Error 120. The required path does not exist "
              << endl;
        exit( 120 );
    }
    // Инвертирование списка ссылок
    TempTop = finish; next = -1;
    while( TempTop != -1 )
    {
        pred = pMinWay[ TempTop ].ref;
        pMinWay[ TempTop ].ref = next;
        next = TempTop; TempTop = pred;
    }
    // Вывод результатов поиска
    f_out << text << "\n Оптимальный путь от старта до финиша проходит"
          << " через \nследующие вершины (первая вершина списка - старт,\n"
          << "последняя - финиш пути):" << endl;
    int          i = 0;    // Счетчик отпечатанных вершин
    while( next != -1 )
    {
        if( !( i%4 ) )
            f_out << endl;
        i++; f_out.width( 12 ); f_out << next;
        next = pMinWay[ next ].ref;
    }
    // Закрытие файла вывода
    f_out.close_f( f_name, 130 );
    return;
}

```

- 2. Программу из упражнения 1 в учебных целях модифицируйте таким образом, чтобы вместо иерархии классов для решения транспортной задачи использовался один класс. Объявление этого класса поместите в заголовочный файл, а реализацию методов — в файл с расширением сpp.**

Файлы IOFile.h и IOFile.cpp полностью совпадают с одноименными файлами из ответа к упражнению 1. Остальные файлы программного проекта приводятся в листингах П1.17—П1.19.

Листинг П1.17. Файл Graph.h

```

/*
Объявление класса для решения транспортной задачи со следующим набором операций:
- динамическое размещение данных о графе (конструктор);
- освобождение занятой динамической памяти (деструктор);
- чтение информации о графе;
- печать информации о графе;
- поиск оптимального пути в неориентированном взвешенном графе;
- выполнение шага вперед из достигнутой вершины по заданному ребру;
- прохождение пути, если он есть, от достигнутой вершины до конечной вершины;
- печать информации о наилучшем пути между заданными вершинами.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Предотвращение многократного включения данного файла
#ifndef __GRAPH_H
#define __GRAPH_H

    // Объявление класса для открытия-закрытия файлов на базе
    // стандартного класса fstream и подключение перегруженных операций
    // << и >>
#include "IOFile.h"

    struct A                // Arc: дуга (ребро) графа
    {
        int    first;       // 1-я вершина ребра
        int    last;        // 2-я вершина ребра
        float  weight;       // Вес ребра
    };

    struct W                // Way: путь до одной вершины
    {
        int    exist;       // ( != 0 ) - путь имеется
        int    ref;         // REFeRence: предыдущая вершина,
                            // через которую проходит путь
        float  SumDist;     // Суммарная длина минимального пути
    };
// *****
// Объявление класса для решения транспортной задачи
class GR
{
    // Данные
private:
    // Адрес первого элемента массива структур с информацией
    // о минимальном пути между заданными вершинами
    W        *pMinWay;
    // Следующие далее данные определены здесь для экономии памяти

```

```

// стека, так как они используются в одном экземпляре
// в рекурсивной функции PassWay
int    one,        // 1-я вершина текущего ребра
       two;        // 2-я вершина текущего ребра
protected:
    int    NumTop,    // Число вершин
           NumArc,    // Число ребер
           start,     // Вершина - старт пути
           finish;    // Вершина - финиш пути
// Адрес первого элемента массива структур с информацией о ребрах
// графа
A       *pArc;
// Методы
public:
    // Конструктор
    GR( int nt, int na );
    ~GR( void )        // Деструктор: подставляемый метод
    {
        if( pArc )
        {
            delete [ ] pArc; pArc = NULL;
        }
        if( pMinWay )
        {
            delete [ ] pMinWay; pMinWay = NULL;
        }
    }
    // Ввод информации о графе из файла на магнитном диске
    void ReadGraph( char f_name[ ] );
    // Вывод информации о графе в файл на магнитном диске
    void WriteGraph( char f_name[ ], int mode, char text[ ] );
    // Поиск оптимального пути в неориентированном взвешенном графе
    void solution( void );
    // Печать информации о наилучшем пути от start до finish
    void OutRes( char f_name[ ], int mode, char text[ ] );
private:
    // Прохождение пути от достигнутой вершины InterMediate, если он
    // есть, до вершины finish
    void PassWay( int InterMediate );
    // Шаг вперед из достигнутой вершины по заданному ребру
    void ForStep( int top1, int IndArc, int top2 );
};
#endif

```

Листинг П1.18. Файл Graph.cpp

```

/*
Реализация класса для решения транспортной задачи со следующим набором операций:
- динамическое размещение данных о графе (конструктор);
- освобождение занятой динамической памяти (деструктор);
- чтение информации о графе;
- печать информации о графе;
- поиск оптимального пути в неориентированном взвешенном графе;
- выполнение шага вперед из достигнутой вершины по заданному ребру;
- прохождение пути, если он есть, от достигнутой вершины до конечной вершины;
- печать информации о наилучшем пути между заданными вершинами.
Объявление класса находится в файле Graph.h.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// Объявление класса для решения транспортной задачи
#include "Graph.h"
//*****
// Конструктор
GR :: GR(
    int      nt,          // Число вершин графа
    int      na )         // Число ребер графа
{
    // Проверка области допустимых значений nt и na
    if( nt < 2 )
    {
        cout << "\n Error 10. In the graph there should be 2 or more"
              << " top " << endl;
        exit( 10 );
    }
    if( na < 1 )
    {
        cout << "\n Error 20. In the graph there should be 1 or more"
              << " edges " << endl;
        exit( 20 );
    }
    // Размещаем массив ребер в динамической памяти
    pArc = new A[ na ];
    if( !pArc )
    {
        cout << "\n Error 30. The information on edges of a graph was"
              << " not placed in dynamic memory " << endl;
        exit( 30 );
    }
    // Размещаем массив структур с информацией о наилучшем пути

```

```

// в динамической памяти
pMinWay = new W[ nt ];
if( !pMinWay )
{
    cout << "\n Error 40. The information on the best path was not"
          " placed in dynamic memory " << endl;
    exit( 40 );
}
// Инициализация данных
for( int i = 0; i < na; i++ )
{
    pArc[ i ].first = 0; pArc[ i ].last = 0;
    pArc[ i ].weight = 0.0f;
}
for( i = 0; i < nt; i++ )
{
    pMinWay[ i ].exist = 0; pMinWay[ i ].ref = 0;
    pMinWay[ i ].SumDist = 0.0f;
}
NumTop = nt; NumArc = na;
}
// Реализация последующих методов совпадает с реализацией методов из
// упражнения 1, но в заголовках всех методов используется имя класса
// GR, как это в виде примера показано далее
// ...
//*****
// Поиск оптимального пути в неориентированном взвешенном графе
void GR :: solution( void )
{
    // Подготовка
    pMinWay[ start ].exist = 1;
    pMinWay[ start ].ref = -1;

    PassWay( start );
    return;
}
// ...

```

Листинг П1.19. Файл TestGr.cpp

```

/*
    Тестирование решения транспортной задачи (задачи коммивояжера). При решении
    задачи используется класс GR, определение которого приведено во включаемом файле
    Graph.h и файле Graph.cpp. Для открытия-закрытия файлов используется класс
    IOFILE, определение которого приведено во включаемом
    файле IOFile.h и файле IOFile.cpp.

```

```

В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "Graph.h"          // Класс для решения транспортной задачи
/*****
// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    GR          g( 5, 6 ); // Создаем граф из 5 вершин и 6 ребер
    // Ввод информации о графе из файла
    g.ReadGraph( "graph.dat" );
    // Вывод информации о графе в файл
    g.WriteGraph( "graph.out", ios::out,
        "                ИНФОРМАЦИЯ О ГРАФЕ \n" );
    g.solution( );          // Ищем наилучший путь между вершинами
    g.OutRes( "graph.out", ios::out | ios::app,
        "\n\n                РЕЗУЛЬТАТЫ РАБОТЫ ПРОГРАММЫ \n");
    return 0;
}

```

П1.9. Глава 10

1. В программе из *разд. 10.2* модифицируйте тестирующий файл `TestSearch.cpp` таким образом, чтобы вызовы методов шаблонного класса использовали полные списки аргументов и были полностью эквивалентны вызовам из *разд. 10.2*.

Программа представлена в листинге П1.20.

Листинг П1.20. Файл `TestSearch.cpp`

```

/*
    Поиск в таблице с использованием шаблонного класса SEARCH_TABLE, объявление
    которого приведено во включаемом файле SearchT.h. Для открытия-закрытия файлов
    используется класс IOFILE, определение которого приведено во включаемом файле
    IOFile.h.

    В этом файле с учебными целями при вызове методов не используются умалчиваемые
    значения параметров - все аргументы указываются явно.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SearchT.h"        // Шаблонный класс для поиска в таблице
// Тестирование
int main( void )           // Возвращает 0 при успехе
{
    // Создаем таблицу из 4 строк: 8 - длина ключа, 62 - длина данного
    // в строке таблицы
    SEARCH_TABLE< 8, 62 >
        t( 4 );
    bool    found;          // true - нашли ключевое слово

```

```

//      в таблице

unsigned int
    IxLine;    // Индекс найденной строки
// Заполняем таблицу для последовательного поиска и
// печатаем ее
t.SeqInpTab( "SearchT.dat" );
t.PrintTab( "SearchT.out", ios::out,
            "          Состояние таблицы:" );
IOFILE      FOut;    // Файловый объект для вывода
// Вывод заголовка
FOut.open_f( "SearchT.out", ios::out | ios::app, 180 );
FOut << "\n    Тестирование последовательного поиска" << endl;
FOut.close_f( "SearchT.out", 190 );
// Последовательный поиск и его результаты
t.SequentialSearch( "and", found, IxLine );
t.PrintSearch( "and", found, IxLine, "SearchT.out",
               ios::out | ios::app );
t.SequentialSearch( "word", found, IxLine );
t.PrintSearch( "word", found, IxLine, "SearchT.out",
               ios::out | ios::app );
// Заполняем таблицу для логарифмического поиска и печатаем ее
t.LogInpTab( "SearchT.dat" );
t.PrintTab( "SearchT.out", ios::out | ios::app,
            "          Состояние таблицы:" );
// Вывод заголовка
FOut.open_f( "SearchT.out", ios::out | ios::app, 200 );
FOut << "\n    Тестирование логарифмического поиска" << endl;
FOut.close_f( "SearchT.out", 210 );
// Логарифмический поиск и его результаты
t.LogariphmSearch( "and", found, IxLine );
t.PrintSearch( "and", found, IxLine, "SearchT.out",
               ios::out | ios::app );
t.LogariphmSearch( "word", found, IxLine );
t.PrintSearch( "word", found, IxLine, "SearchT.out",
               ios::out | ios::app );
// Заполняем таблицу для хэш-поиска и печатаем ее
t.HashInpTab( "SearchT.dat", 2 );
t.PrintTab( "SearchT.out", ios::out | ios::app,
            "          Состояние таблицы:" );
// Печать заголовка
FOut.open_f( "SearchT.out", ios::out | ios::app, 220 );
FOut << "\n    Тестирование хэш-поиска" << endl;
FOut.close_f( "SearchT.out", 230 );
// Хэш-поиск и его результаты
t.HashSearch( "work", found, IxLine );
t.PrintSearch( "work", found, IxLine, "SearchT.out",

```

```

        ios::out | ios::app );
t.HashSearch( "type", found, IxLine );
t.PrintSearch( "type", found, IxLine, "SearchT.out",
        ios::out | ios::app );

return 0;
}

```

2. Программу из *разд. 10.2* в учебных целях модифицируйте таким образом, чтобы использовалась иерархия шаблонных классов. В шаблонном базовом классе используйте конструктор, деструктор и метод для печати строк таблицы. В шаблонном производном классе для компактности программного кода оставьте только конструктор; метод заполнения таблицы для хэш-поиска; методы, реализующие хэш-поиск и печать результатов поиска.

Определение класса `IOFile` в файле `IOFile.h` приводилось ранее и здесь не рассматривается. Программа представлена в листингах П1.21—П1.23.

Листинг П1.21. Файл `SearchT_B.h`

```

/*
Объявление шаблонного класса со следующим набором операций:
1. Динамическое размещение таблицы с инициализацией (конструктор).
2. Освобождение занятой динамической памяти (деструктор).
3. Печать содержимого таблицы.
В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// *****
// Предотвращение многократного включения данного файла
#ifndef __SEARCHT_B_H
#define __SEARCHT_B_H
    // Класс для открытия-закрытия файлов на базе стандартного класса
    //   fstream и подключение перегруженных операций << и >>
#include "IOFile.h"
    // *****
    // Объявление базового шаблонного класса для работы с таблицей:
    //   LenKey-1 - длина поля ключа, LenData-1 - длина поля данного
    //   в строке таблицы
    template < unsigned int LenKey, unsigned int LenData >
    class SEARCH_TABLE_B
    {
        // Локальные типы
        struct STR_TAB    // Строка таблицы
        {
            // Ключ
            char
                Key[ LenKey ];

```

```

        // Данное
        char
            Data[ LenData ];
};

// Данные
protected:
    STR_TAB
        *pTable; // Адрес первой строки таблицы
    unsigned int
        size;     // Размер таблицы
// Методы
public:
    // Конструктор
    SEARCH_TABLE_B( unsigned int s );
    // Деструктор
    ~SEARCH_TABLE_B( void )
    {
        if( pTable )
        {
            delete [ ] pTable; pTable = NULL;
        }
    }
    // Вывод таблицы
    void PrintTab( char FName[ ] = "SearchT.out",
        int mode = ios::out | ios::app,
        char text[ ] = "                Состояние таблицы:" );
}; // Конец объявления базового шаблонного класса
// *****
// Конструктор
template < unsigned int LenKey, unsigned int LenData >
SEARCH_TABLE_B< LenKey, LenData > :: SEARCH_TABLE_B(
    unsigned int
        s ) // Число строк таблицы
{
    // Проверяем, подходит ли размер таблицы?
    if( s<2 )
    {
        cout << "\n Error 10. In the table should be not less than"
            " two string" << endl;
        exit( 10 );
    }
    // Размещаем таблицу в динамической памяти
    pTable = new STR_TAB[ s ];
    if( !pTable )
    {

```

```

        cout << "\n Error 20. The table in dynamic memory not was"
              " placed " << endl;
        exit( 20 );
    }
    // Инициализация таблицы
    for( unsigned int i=0; i<s; i++ )
    {
        pTable[ i ].Key[ 0 ] = '\0';
        pTable[ i ].Data[ 0 ] = '\0';
    }
    size = s;
}
// *****
// Вывод таблицы
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_B< LenKey, LenData > :: PrintTab(
    char    FName[ ], // Файл вывода
    int     mode,      // Режим открытия файла
    char    text[ ] ) // Заголовок для печати
{
    IOFILE FOut;      // Файловый объект для вывода
    // Открытие файла для записи
    FOut.open_f( FName, mode, 80 );
    // Печать таблицы с заголовком
    FOut << endl << text << endl;
    for( unsigned int i=0; i<size; i++ )
    {
        FOut.width( LenKey-1 );
        FOut << setiosflags( ios::left ) << pTable[ i ].Key;
        FOut.width( LenData-1 );
        FOut << setiosflags( ios::left ) << pTable[ i ].Data << endl;
    }
    // Закрытие файла вывода
    FOut.close_f( FName, 90 );
    return;
}
#endif

```

Листинг П1.22. Файл SearchT_D.h

/*

Поиск в таблице. Объявление производного шаблонного класса.

Базовый шаблонный класс объявлен в файле SearchT_B.h.

Используется следующий набор операций:

1. Заполнение таблицы для хэш-поиска.

2. Хэш-поиск в таблице.
3. Печать результатов поиска.
- В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0

```

*/
// *****
// Предотвращение многократного включения данного файла
#ifndef __SEARCHT_D_H
#define __SEARCHT_D_H
    // Объявление базового шаблонного класса для работы с таблицей
    #include "SearchT_B.h"
    #include <iomanip>    // Для манипуляторов ввода-вывода
    #include <string.h>  // Для работы с C-строками
    // *****
    // Объявление производного шаблонного класса для поиска в таблице:
    //   LenKey-1 - длина поля ключа, LenData-1 - длина поля данного
    //   в строке таблицы
    template < unsigned int LenKey, unsigned int LenData >
    class SEARCH_TABLE_D
    : public SEARCH_TABLE_B< LenKey, LenData >
    {
        // Методы
    public:
        // Конструктор
        SEARCH_TABLE_D( unsigned int s )
        : SEARCH_TABLE_B< LenKey, LenData >( s ){}
        // Заполнение таблицы для хэш-поиска
        void HashInpTab( char FName[ ] = "SearchT.dat",
                        unsigned int TableLen = 2 );
        // Вывод результатов поиска в таблице
        void PrintSearch( char KeyWord[ ], bool found,
                        unsigned int IxLine,
                        char FName[ ] = "SearchT.out",
                        int mode = ios::out | ios::app );
        // Поиск в хэш-таблице
        void HashSearch( char KeyWord[ ], bool &found,
                        unsigned int &IxLine );
    private:
        // Преобразование символа ключа - строчная латинская буква, цифра
        //   или пробел - в его порядковый номер (целое число)
        int Kod( char symbol );
        // Хэш-функция ключа KeyWord[ ] из LenKey-1 символов (символ -
        //   строчная латинская буква, цифра или пробел) для таблицы из
        //   size строк
        unsigned int Hash( char KeyWord[ ] );
    };
    // Конец объявления шаблонного класса

```

```

// *****
// Вывод результатов поиска в таблице
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_D< LenKey, LenData > :: PrintSearch(
    // Ключевое слово для поиска
    char    KeyWord[ ],
    bool    found,      // 1 - нашли в таблице
    unsigned int
        IxLine,        // Индекс строки в таблице
    char    FName[ ], // Файл вывода
    int     mode )      // Режим открытия файла
{
    IOFILE FOut;        // Файловый объект для вывода
    // Открытие файла для записи
    FOut.open_f( FName, mode, 100 );
    FOut << endl << "Результаты поиска для ключевого слова: "
        << KeyWord << endl;
    if( found )
    {
        FOut << "Индекс строки в таблице: " << IxLine <<
            ". Найденная строка:" << endl;
        FOut.width( LenKey-1 );
        FOut << setiosflags( ios::left ) << pTable[ IxLine ].Key;
        FOut.width( LenData-1 );
        FOut << setiosflags( ios::left ) << pTable[ IxLine ].Data
            << endl;
    }
    else
        FOut << "Строка с ключом \"" << KeyWord <<
            "\" в таблице не найдена" << endl;
    // Закрытие файла вывода
    FOut.close_f( FName, 110 );
    return;
}
// *****
// Преобразование символа ключа - строчная латинская буква, цифра или
// пробел - в его порядковый номер (целое число)
template < unsigned int LenKey, unsigned int LenData >
int SEARCH_TABLE_D< LenKey, LenData > :: Kod(
    // Порядковый номер символа
    char    symbol )    // Преобразуемый символ
{
    switch( symbol )
    {
        case 'a': return 0;

```

```

        case 'b': return 1;
        // и т.д.
        case 'y': return 24;
        case 'z': return 25;
        case '0': return 26;
        case '1': return 27;
        // и т.д.
        case '9': return 35;
        case ' ': return 36;
    default:
        cout << "\n Error 130. The invalid character in a key: "
              "\n the key can consist only of line Latin "
              "characters, digits and blanks " << endl;
        exit( 130 );
    }
    return -1;        // Этот оператор не будет выполняться - он
                      // помещен сюда для "глупого" компилятора
}

// *****
// Хэш-функция ключа KeyWord[ ] из LenKey-1 символов (СИМВОЛ -
// строчная латинская буква, цифра или пробел) для таблицы из size
// строк
template < unsigned int LenKey, unsigned int LenData >
unsigned int SEARCH_TABLE_D< LenKey, LenData > :: Hash(
    // Возвращает индекс строки таблицы

    // Ключ
    char    KeyWord[ ] )
{
    unsigned int
        IKey,        // Индекс символа в ключе
        ih = 0;      // Значение хэш-функции
    // Вычисление индекса строки таблицы
    for( IKey=0; IKey<strlen( KeyWord ); IKey++ )
    {
        ih = ih * 37 + Kod( KeyWord[ IKey ] );
        ih = ih % size;
    }
    return ih;
}

// *****
// Заполнение таблицы для хэш-поиска
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_D< LenKey, LenData > :: HashInpTab(
    char    FName[ ], // Файл ввода

```

```

    unsigned int
        TableLen )// Число вводимых строк
{
    unsigned int
        i,          // Индекс строки таблицы
        line;       // Индекс текущей строки файла
    bool    found;   // true - найдена позиция вставки
    // Заносимое слово
    char    KeyWord[ LenKey ];
    // Инициализация таблицы нуль-символами
    for( i=0; i<size; i++ )
    {
        for( unsigned int il=0; il<LenKey; il++ )
            pTable[ i ].Key[ il ] = '\0';
        for( il=0; il<LenData; il++ )
            pTable[ i ].Data[ il ] = '\0';
    }
    // Отметка строк таблицы как свободных
    for( i=0; i<size; i++ )
        pTable[ i ].Key[ 0 ] = ' ';
    // Открытие файла для чтения
    IOFILE FIn;
    FIn.open_f( FName, ios::in, 140 );
    // Занесение в таблицу исходных строк
    for( line=0; line<TableLen; line++ )
    {
        // Цикл чтения исходных строк
        FIn >> KeyWord;
        if( !FIn ) // Обработка ошибки чтения
        {
            cout << endl << "Error 150. A read error";
            exit( 150 );
        }
        // Поиск индекса i строки таблицы для ее заполнения
        // Пока индекс не найден
        found = false;
        i = Hash( KeyWord );
        while( !found )
        {
            if( pTable[ i ].Key[ 0 ] == ' ' )
                // Индекс найден
                found = true;
            else
            {
                // Конфликт - шаг по таблице
                i++; i = ( i>( size-1 )?0:i );
            }
        }
    }
}

```

```

    }
    // Чтение данного
    FIn >> pTable[ i ].Data;
    if( !FIn )
    {
        cout << endl << "Error 160. A read error ";
        exit( 160 );
    }
    // Занесение ключа в строку i таблицы
    strcpy( pTable[ i ].Key, KeyWord );
}
// Закрытие файла ввода
FIn.close_f( FName, 170 );
return;
}
// *****
// Поиск в хэш-таблице
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_D< LenKey, LenData > :: HashSearch(
    // Ключевое слово для поиска
    char    KeyWord[ ],
    bool    &found,    // true - нашли
    unsigned int
        &IxLine ) // Индекс найденной строки
                // в таблице
{
    unsigned int
        i;    // Индекс строки таблицы
    bool    EndTab;    // true - достигли свободной строки
    // Подготовка к поиску
    found = false; EndTab = false; i = Hash( KeyWord );
    // Поиск в таблице
    while( !found && !EndTab )
    {
        if( pTable[ i ].Key[ 0 ] == ' ' )
            // Достигли свободной строки
            EndTab = true;
        else
        {
            if( !strcmp( pTable[ i ].Key, KeyWord ) )
            {
                // Нашли
                found = true; IxLine = i;
            }
            else
            {
                // Шаг по таблице

```

```

        i++; i = ( i > ( size-1 ) ? 0 : i );
    }
}
}
return;
}
#endif

```

Листинг П1.23. Файл TestSearch.cpp

```

/*
    Поиск в таблице с использованием иерархии шаблонных классов, объявление
    которых приведено во включаемых файлах SearchT_B.h и SearchT_D.h. Для открытия-
    закрытия файлов используется класс IOFILE, определение которого приведено
    во включаемом файле IOFile.h.

    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SearchT_D.h"    // Иерархия шаблонных классов для поиска
                        // в таблице

// Тестирование
int main( void )        // Возвращает 0 при успехе
{
    // Создаем таблицу из 4 строк: 8 - длина ключа, 62 - длина данного
    // в строке таблицы
    SEARCH_TABLE_D< 8, 62 >
        t( 4 );

    bool        found;    // true - нашли ключевое слово
                        // в таблице

    unsigned int
        IxLine;    // Индекс найденной строки
    IOFILE      FOut;    // Файловый объект для вывода
    // Заполняем таблицу для хэш-поиска и печатаем ее
    t.HashInpTab( );
    t.PrintTab( "SearchT.out", ios::out );
    // Печать заголовка
    FOut.open_f( "SearchT.out", ios::out | ios::app, 220 );
    FOut << "\n    Тестирование хэш-поиска" << endl;
    FOut.close_f( "SearchT.out", 230 );
    // Хэш-поиск и его результаты
    t.HashSearch( "work", found, IxLine );
    t.PrintSearch( "work", found, IxLine );
    t.HashSearch( "type", found, IxLine );
    t.PrintSearch( "type", found, IxLine );

    return 0;
}

```

3. Программу из упражнения 2 модифицируйте так, чтобы в поле ключа можно было использовать только русские строчные буквы.

Файлы IOFile.h, SearchT_B.h совпадают с соответствующими файлами из упражнения 2 и далее не приводятся. Программа представлена в листингах П1.24—П1.26.

Листинг П1.24. Файл SearchT_D.h

```
/*
    Поиск в таблице. Объявление производного шаблонного класса.
    Базовый шаблонный класс объявлен в файле SearchT_B.h.
    Используется следующий набор операций:
    1. Заполнение таблицы для хэш-поиска.
    2. Хэш-поиск в таблице.
    3. Печать результатов поиска.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
// *****
// Предотвращение многократного включения данного файла
#ifndef __SEARCHT_D_H
#define __SEARCHT_D_H
    // Объявление базового шаблонного класса для работы с таблицей
    #include "SearchT_B.h"
    #include <iomanip> // Для манипуляторов ввода-вывода
    #include <string.h> // Для работы с C-строками
    // *****
    // Объявление производного шаблонного класса для поиска в таблице:
    //   LenKey-1 - длина поля ключа, LenData-1 - длина поля данного
    //   в строке таблицы
    template < unsigned int LenKey, unsigned int LenData >
    class SEARCH_TABLE_D
    : public SEARCH_TABLE_B< LenKey, LenData >
    {
        // Методы
    public:
        // Конструктор
        SEARCH_TABLE_D( unsigned int s )
        : SEARCH_TABLE_B< LenKey, LenData >( s ){}
        // Заполнение таблицы для хэш-поиска
        void HashInpTab( char FName[ ] = "SearchT.dat",
                        unsigned int TableLen = 2 );
        // Вывод результатов поиска в таблице
        void PrintSearch( char KeyWord[ ], bool found,
                        unsigned int IxLine,
                        char FName[ ] = "SearchT.out",
                        int mode = ios::out | ios::app );
    };
}
```

```

// Поиск в хэш-таблице
void HashSearch( char KeyWord[ ], bool &found,
                unsigned int &IxLine );

private:
// Преобразование символа ключа - строчная русская буква - в его
// порядковый номер (целое число)
int Kod( char symbol );
// Хэш-функция ключа KeyWord[ ] из LenKey-1 символов (символ -
// строчная русская буква пробел) для таблицы из size строк
unsigned int Hash( char KeyWord[ ] );
}; // Конец объявления шаблонного класса
// *****
// Вывод результатов поиска в таблице
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_D< LenKey, LenData > :: PrintSearch(
    // Ключевое слово для поиска
    char KeyWord[ ],
    bool found, // 1 - нашли в таблице
    unsigned int
        IxLine, // Индекс строки в таблице
    char FName[ ], // Файл вывода
    int mode ) // Режим открытия файла
{
    IOFILE FOut; // Файловый объект для вывода
    // Открытие файла для записи
    FOut.open_f( FName, mode, 100 );
    FOut << endl << "Результаты поиска для ключевого слова: "
        << KeyWord << endl;
    if( found )
    {
        FOut << "Индекс строки в таблице: " << IxLine <<
            ". Найденная строка:" << endl;
        FOut.width( LenKey-1 );
        FOut << setiosflags( ios::left ) << pTable[ IxLine ].Key;
        FOut.width( LenData-1 );
        FOut << setiosflags( ios::left ) << pTable[ IxLine ].Data
            << endl;
    }
    else
        FOut << "Строка с ключом \" " << KeyWord <<
            "\" в таблице не найдена" << endl;
    // Закрытие файла вывода
    FOut.close_f( FName, 110 );
    return;
}

```

```

// *****
// Преобразование символа ключа - строчная русская буква - в его
// порядковый номер (целое число)
template < unsigned int LenKey, unsigned int LenData >
int SEARCH_TABLE_D< LenKey, LenData > :: Kod(
    // Порядковый номер символа
    char symbol ) // Преобразуемый символ
{
    switch( symbol )
    {
        case 'a': return 0;
        case 'б': return 1;
        // и т.д.
        case 'ю': return 31;
        case 'я': return 32;
        default:
            cout << "\n Error 130. The invalid character in a key: "
                 "\n the key can consist only of line Russian "
                 "characters " << endl;
            exit( 130 );
    }
    return -1; // Этот оператор не будет выполняться - он
              // помещен сюда для "глупого" компилятора
}
// *****
// Хэш-функция ключа KeyWord[ ] из LenKey-1 символов (символ -
// строчная русская буква) для таблицы из size строк
template < unsigned int LenKey, unsigned int LenData >
unsigned int SEARCH_TABLE_D< LenKey, LenData > :: Hash(
    // Возвращает индекс строки таблицы
    // Ключ
    char KeyWord[ ] )
{
    unsigned int
        IKey, // Индекс символа в ключе
        ih = 0; // Значение хэш-функции
    // Вычисление индекса строки таблицы
    for( IKey=0; IKey<strlen( KeyWord ); IKey++ )
    {
        ih = ih * 33 + Kod( KeyWord[ IKey ] );
        ih = ih % size;
    }
    return ih;
}
// *****

```

```

// Заполнение таблицы для хэш-поиска
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_D< LenKey, LenData > :: HashInpTab(
    char   FName[ ], // Файл ввода
    unsigned int
        TableLen )// Число вводимых строк
{
    unsigned int
        i,          // Индекс строки таблицы
        line;       // Индекс текущей строки файла
    bool   found;    // true - найдена позиция вставки
    // Заносимое слово
    char   KeyWord[ LenKey ];
    // Инициализация таблицы нуль-символами
    for( i=0; i<size; i++ )
    {
        for( unsigned int il=0; il<LenKey; il++ )
            pTable[ i ].Key[ il ] = '\0';
        for( il=0; il<LenData; il++ )
            pTable[ i ].Data[ il ] = '\0';
    }
    // Отметка строк таблицы как свободных
    for( i=0; i<size; i++ )
        pTable[ i ].Key[ 0 ] = ' ';
    // Открытие файла для чтения
    IOFILE FIn;
    FIn.open_f( FName, ios::in, 140 );
    // Занесение в таблицу исходных строк
    for( line=0; line<TableLen; line++ )
    {
        // Цикл чтения исходных строк
        FIn >> KeyWord;
        if( !FIn ) // Обработка ошибки чтения
        {
            cout << endl << "Error 150. A read error";
            exit( 150 );
        }
        // Поиск индекса i строки таблицы для ее заполнения
        // Пока индекс не найден
        found = false;
        i = Hash( KeyWord );
        while( !found )
        {
            if( pTable[ i ].Key[ 0 ] == ' ' )
                // Индекс найден
                found = true;

```

```

        else
        {
            // Конфликт - шаг по таблице
            i++; i = ( i > ( size-1 ) ? 0 : i );
        }
    }
    // Чтение данного
    FIn >> pTable[ i ].Data;
    if( !FIn )
    {
        cout << endl << "Error 160. A read error ";
        exit( 160 );
    }
    // Занесение ключа в строку i таблицы
    strcpy( pTable[ i ].Key, KeyWord );
}
// Закрытие файла ввода
FIn.close_f( FName, 170 );
return;
}
// *****
// Поиск в хэш-таблице
template < unsigned int LenKey, unsigned int LenData >
void SEARCH_TABLE_D< LenKey, LenData > :: HashSearch(
    // Ключевое слово для поиска
    char    KeyWord[ ],
    bool    &found,    // true - нашли
    unsigned int
        IxLine ) // Индекс найденной строки в таблице
{
    unsigned int
        i;          // Индекс строки таблицы
    bool    EndTab;  // true - достигли свободной строки
    // Подготовка к поиску
    found = false; EndTab = false; i = Hash( KeyWord );
    // Поиск в таблице
    while( !found && !EndTab )
    {
        if( pTable[ i ].Key[ 0 ] == ' ' )
            // Достигли свободной строки
            EndTab = true;
        else
        {
            if( !strcmp( pTable[ i ].Key, KeyWord ) )
            {
                // Нашли
                found = true; IxLine = i;
            }
        }
    }
}

```

```

        else
        {
            // Шаг по таблице
            i++; i = ( i > ( size-1 ) ? 0 : i );
        }
    }
}

return;
}
#endif

```

Листинг П1.25. Файл TestSearch.cpp

```

/*
    Поиск в таблице с использованием иерархии шаблонных классов,
    объявление которых приведено во включаемых файлах SearchT_B.h и SearchT_D.h.
    Для открытия-закрытия файлов используется класс IOFILE, определение которого
    приведено во включаемом файле IOFile.h.
    В. Давыдов, консольное приложение, Microsoft Visual Studio C++ 6.0
*/
#include "SearchT_D.h" // Иерархия шаблонных классов для поиска
                        // в таблице

// Тестирование
int main( void ) // Возвращает 0 при успехе
{
    // Создаем таблицу из 6 строк: 8 - длина ключа, 62 - длина данного
    // в строке таблицы
    SEARCH_TABLE_D< 8, 62 >
        t( 6 );

    bool found; // true - нашли ключевое слово в таблице
    unsigned int
        IxLine; // Индекс найденной строки
    IOFILE FOut; // Файловый объект для вывода
    // Заполняем таблицу для хэш-поиска и печатаем ее
    t.HashInpTab( "SearchT.dat", 3 );
    t.PrintTab( "SearchT.out", ios::out );
    // Печать заголовка
    FOut.open_f( "SearchT.out", ios::out | ios::app, 220 );
    FOut << "\n    Тестирование хэш-поиска" << endl;
    FOut.close_f( "SearchT.out", 230 );
    // Хэш-поиск и его результаты
    t.HashSearch( "вызов", found, IxLine );
    t.PrintSearch( "вызов", found, IxLine );
    t.HashSearch( "тип", found, IxLine );
    t.PrintSearch( "тип", found, IxLine );
    return 0;
}

```

Листинг П1.26. Результаты выполнения программы, выводимые в файл на магнитном диске

```
Состояние таблицы:
работа work

вызов call
позиция position
```

```
Тестирование хэш-поиска
Результаты поиска для ключевого слова: вызов
Индекс строки в таблице: 2. Найденная строка:
вызов call
Результаты поиска для ключевого слова: тип
Строка с ключом "тип" в таблице не найдена
```

Данные результаты получены для файла ввода (листинг П1.27).

Листинг П1.27. Файл ввода

```
вызов call
позиция position
работа work
слово word
тип type
успех success
```

П1.10. Глава 11

1. Что собой представляют обобщенные связанные списки и когда их следует применять?

Обобщенный связанный список представляет собой структуру данных, в которой элементы содержат явные указания на связи между собой. Списки являются наиболее эффективными структурами данных, если часто требуется выполнять операции добавления или исключения элементов.

2. Перечислите известные вам разновидности списков.

Существуют линейные и циклические списки, которые могут быть однонаправленными и двунаправленными, и мультисписки.

3. На базе какой разновидности списков целесообразно реализовать динамический стек?

Динамический стек эффективно реализуется на базе однонаправленного линейного списка при исключении ненужных операций. В этом случае мы получаем в качестве частного случая очередь типа LIFO.

4. Укажите преимущества циклических списков по сравнению с линейными списками.

Циклический список не имеет первого и последнего элементов. При работе с ним удобно использовать указатель на последний элемент. Это позволяет более эффективно, по сравнению с кольцевым однонаправленным линейным списком, реализовать операции добавления или удаления элемента из конца списка (не нужно продвигаться по всему списку для получения указателя на последний элемент). Вместе с тем, аналогичные операции с началом списка остаются почти такими же эффективными.

5. Какими преимуществами обладают двунаправленные списки?

Каждый элемент двунаправленного списка содержит два указателя — один указывает на предшествующий элемент, а другой — на последующий. Такие списки также могут быть линейными и циклическими. Отличительной особенностью подобных списков является более эффективная реализация операций вставки-удаления элементов. Двунаправленные линейные списки используют обычно два указателя — на левый и правый концы списка. На базе двунаправленного линейного списка можно реализовать, например, универсальную динамическую очередь.

6. Перечислите разновидности очередей.

Существуют следующие виды очередей:

- ☐ универсальная очередь;
- ☐ ограниченная очередь типа FIFO (занесение с одного конца и извлечение с другого конца очереди);
- ☐ ограниченная очередь типа LIFO — стек (занесение и извлечение только с одного конца очереди).

Существуют и другие варианты ограниченных очередей. Очереди можно реализовать на базе списков (динамические очереди неограниченного размера) и на базе массивов (очереди ограниченного размера).

7. Какие операции определены над очередью?

Над очередью определены три основных операции:

- ☐ занесение элемента (заказа на обслуживание) в очередь;
- ☐ выбор элемента из очереди (для его обслуживания);
- ☐ просмотр элементов очереди.

П1.11. Глава 13

1. Чем отличаются C-строки и объекты классов `string` и `wstring` стандартной библиотеки языка C++? Чем отличаются объекты классов `string` и `wstring`?

В языке C строка представляет собой обычный символьный массив типа `char *` (с модификатором `const` или без него). Строка, хранимая в таком массиве, завершается нулевым байтом, в котором все биты нулевые. В стандартной библиотеке языка C++ для представления строк и манипулирования со строками используются объекты классов `string` и `wstring`. Строковые классы стандартной

библиотеки языка C++ позволяют работать со строками как с обычными предопределенными типами, не создающими проблем для пользователей. Это означает, что строки, как и объекты предопределенных типов, можно копировать, присваивать, складывать, сравнивать и т. п., используя традиционные операции. Современная обработка данных во многом ориентирована на работу с текстом.

Классы `string` и `wstring` стандартной библиотеки языка C++ являются специализациями шаблонного класса `basic_string` для типов `char` и `wchar_t`:

```
typedef basic_string<char> string;  
typedef basic_string<wchar_t> wstring;
```

Класс `wstring` позволяет работать со строками, содержащими символы в многобайтовой кодировке (например, в кодировке Unicode или азиатских кодировках).

2. Сравните строковые функции языка C и методы классов `string` и `wstring` стандартной библиотеки языка C++.

Язык C++ не предусматривает строкового типа данных. Вместо этого он поддерживает символьные массивы, завершаемые нуль-символом. Для работы с такими массивами библиотека языка содержит строковые функции, унаследованные от языка C и описанные в заголовочном файле `<string.h>` (`<cstring>`). Строковые функции языка C обладают следующими отличительными особенностями:

- ☐ высокое быстродействие;
- ☐ неудобный интерфейс;
- ☐ опасность их использования, поскольку выход за границы строки в функциях не контролируется.

Последних двух недостатков лишены классы `string` и `wstring` стандартной библиотеки языка C++. Но использование для работы со строками класса `string` приводит к некоторому понижению быстродействия. Тем не менее, использование этого класса для работы со строками весьма целесообразно и является хорошим стилем программирования.

3. Перечислите разновидности конструкторов класса `string` и приведите примеры их использования. Роль деструктора класса `string`.

Разновидности конструкторов класса `string` и примеры их использования приведены в файле `str2.cpp` в *разд. 13.1*.

В классе `string` имеется *деструктор*, который вызывается в программе автоматически, уничтожая все символы строки и освобождая динамическую память.

4. Какие операции предусмотрены над строковыми объектами класса `string`?

Для объектов класса `string` определены следующие операции: `"="` (присваивание), `"+"` (конкатенация), `"=="` (равенство), `"!="` (неравенство), `"<"` (меньше), `"<="` (меньше или равно), `">"` (больше), `">="` (больше или равно), `"[]"` (индексация), `"<<"` (вывод), `">>"` (ввод) и `"+="` (добавление).

Примеры использования этих операций приведены в файлах `str1.cpp`, `str2.cpp` и `str3.cpp` в *разд. 13.1* и *13.2*.

5. Какие группы методов предусмотрены для обработки частей строк в классе `string`?

Для обработки частей строк в классе `string` имеется множество методов, которые можно разбить на следующие подгруппы:

- ☐ методы присваивания и добавления частей строк (`assign`, `append`);
- ☐ методы преобразования строк (`insert`, `erase`, `clear`, `replace`, `swap`, `substr`, `c_str` — преобразование объекта типа `string` в C-строку, `data` — преобразование объекта типа `string` в массив символов — в отличие от предыдущего метода в конец массива символ `'\0'` не помещается, `copy` — копирование в C-строку части объекта с типом `string`);
- ☐ методы поиска подстрок (`find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of`);
- ☐ методы сравнения и получения характеристик строк (`compare`, `size`, `length`, `max_size`, `capacity`, `empty`).

П1.12. Глава 15

1. Что собой представляют контейнеры, зачем они нужны?

Контейнер — это объект, содержащий набор других объектов, организованный определенным образом. Контейнеры предназначены для управления коллекциями объектов определенного типа. У каждой разновидности контейнеров имеются свои достоинства и недостатки, поэтому существование разных контейнеров отражает различие между требованиями к коллекциям в программах. Примерами контейнеров являются массивы (векторы и ассоциативные массивы) и списки (собственно списки, очереди, стеки). Как правило, в контейнер можно добавлять объекты и удалять их из него. Работа с контейнерами поддерживается в стандартной библиотеке с помощью контейнерных классов.

Для каждого типа контейнера в соответствующем контейнерном классе определены методы для работы с его элементами (объектами), не зависящие от конкретного типа объектов, которые хранятся в контейнере. По этой причине один и тот же вид контейнера можно использовать для хранения и работы с объектами различных типов. Эта возможность реализована с помощью *шаблонов классов*.

Зачем же нужны контейнеры, чем они хороши? Использование контейнеров позволяет значительно повысить *надежность* программ, их *переносимость* и *универсальность*. При этом одновременно *уменьшаются сроки* разработки и *стоимость* разработки таких программ. Но универсальность и безопасность контейнерных классов не могут не сказаться на быстродействии использующих их программ. Снижение быстродействия, в зависимости от реализации компилятора языка C++, может оказаться весьма значительным. Уместно заметить также, что на освоение стандартной библиотеки языка C++ придется потратить немалые время и усилия.

2. Что представляют собой последовательные и ассоциативные контейнеры?

Контейнеры STL можно разделить на последовательные и ассоциативные. *Последовательные контейнеры* обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. Разновидностями после-

довательных контейнеров являются векторы (`vector`), двусторонние очереди или, иначе, деки (`deque`), списки (`list`), стеки (`stack`), очереди (`queue`) и очереди с приоритетами (`priority_queue`). В последовательном контейнере каждый объект (элемент) занимает определенную позицию, которая зависит только от времени и места вставки элемента в контейнер, но не зависит от значения элемента. Каждый из перечисленных контейнеров обеспечивает свой набор действий над содержащимися в нем объектами. Выбор контейнера зависит от того, что требуется делать с объектами в программе. Например, если требуется часто вставлять и удалять элементы из середины последовательности, то следует использовать список. Если же включение или удаление объектов выполняется чаще в конце или начало последовательности, то лучше использовать двустороннюю очередь. Первые три из перечисленных последовательных контейнеров являются основными, а остальные — вспомогательными. В реализации вспомогательных контейнеров используются основные контейнеры.

Ассоциативные контейнеры представляют собой *отсортированные коллекции*, в которых позиция объекта (элемента) зависит от его значения по определенному критерию сортировки. Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу и построены на основе сбалансированных деревьев. Разновидностями ассоциативных контейнеров являются словари (`map`), словари с дубликатами (`multimap`), множества (`set`), множества с дубликатами (`multiset`) и битовые множества (`bitset`). Обратите внимание на то, что перечисленные ассоциативные контейнеры полностью независимы друг от друга. Они имеют разные реализации и не являются производными друг от друга. Автоматическая сортировка элементов в ассоциативных контейнерах не означает, что они специально предназначены для сортировки элементов. С таким же успехом можно отсортировать элементы последовательного контейнера. Основным достоинством автоматической сортировки в ассоциативных контейнерах является более высокая эффективность поиска в них элемента с заданными свойствами. При использовании в них двоичного поиска сложность поиска логарифмическая, а не линейная. Например, для поиска в коллекции из 1000 элементов понадобится всего 10 (ближайшее большее целое от $\log_2 1000$) сравнений. Таким образом, автоматическая сортировка элементов является только полезным "побочным эффектом" реализации ассоциативного контейнера, спроектированного в расчете на повышение эффективности поиска.

3. Назовите для контейнеров общие возможности, унифицированные типы и общие операции и методы.

Общие возможности контейнеров сводятся к следующим. Контейнеры должны поддерживать семантику значений вместо ссылочной семантики. Это означает, что при вставке элемента контейнер должен создавать его внутреннюю копию, вместо того чтобы сохранять ссылку на внешний объект. Элементы в контейнере должны располагаться в определенном порядке. Это означает, что при повторном переборе элементов контейнера с применением итератора порядок перебора элементов должен остаться прежним. Итераторы представляют собой основной интерфейс для работы алгоритмов STL. В общем случае, операции с элементами контейнеров не безопасны. Вызывающая сторона должна проследить за тем, чтобы параметры операции соответствовали требованиям. Нарушение правил (например, использование недействительного индекса) приведет к непредсказуемым последствиям.

Унифицированными типами контейнеров являются тип элемента (`value_type`); тип индексов, счетчиков элементов и т. д. (`size_type`); итератор (`iterator`) — аналог указателя на элемент; константный итератор (`const_iterator`) используется тогда, когда значения соответствующих элементов контейнера не изменяются; обратный итератор (`reverse_iterator`) просматривает контейнер в обратном порядке; константный обратный итератор (`const_reverse_iterator`), ссылка на элемент (`reference`), константная ссылка на элемент (`const_reference`); тип ключа (`key_type`) используется только для ассоциативных контейнеров; тип критерия сравнения (`key_compare`) используется только для ассоциативных контейнеров.

Общие операции и методы над контейнерами. При помощи итераторов можно просматривать контейнеры, не заботясь о фактических типах объектов, находящихся в них. Для этого в каждом контейнере определены перечисленные далее методы.

```
iterator begin( );  
const_iterator begin( );
```

Указывают на первый элемент контейнера.

```
iterator end( );  
const_iterator end( );
```

Указывают на элемент контейнера, следующий за последним (но не последний элемент контейнера).

```
reverse_iterator rbegin( );  
const_reverse_iterator rbegin( );
```

Указывают на первый элемент контейнера в обратной последовательности.

```
reverse_iterator rend( );  
const_reverse_iterator rend( );
```

Указывают на элемент контейнера, следующий за последним, в обратной последовательности.

Во всех контейнерах имеются методы, позволяющие определить характеристики контейнера.

```
size_type size( ) const;
```

Возвращает число элементов контейнера.

```
size_type max_size( ) const;
```

Возвращает максимально возможное число элементов контейнера (чуть больше четырех миллиардов).

```
bool empty( ) const;
```

Возвращает `true`, если контейнер пуст.

Во всех контейнерах имеются конструкторы, позволяющие создавать и инициализировать объекты-контейнеры. К таким конструкторам относятся конструктор умолчания (без параметров), обычный конструктор с инициализацией элементов элементами другого контейнера, конструктор копирования. Имеется также и деструктор.

В контейнерных классах предусмотрены операции сравнения объектов-контейнеров ("`==`", "`!=`", "`<`", "`>`", "`<=`", "`>=`"). Перечисленные операции сравнения

определяются по следующим правилам: сравниваемые контейнеры должны относиться к одному типу; два контейнера равны, если их элементы совпадают и следуют в одинаковом порядке; отношение "меньше/больше" между контейнерами проверяется по лексикографическому критерию. Вместе с тем заметим, что с использованием алгоритмов сравнения можно сравнивать разнотипные контейнеры.

Во всех контейнерных классах предусмотрены *операция присваивания и метод `swap()`*. В процессе присваивания все элементы контейнера-приемника удаляются и после этого все элементы контейнера-источника копируются в контейнер-приемник. Это операция является дорогостоящей и имеет линейную сложность. Для повышения эффективности можно использовать метод `swap()`, который *меняет местами два контейнера*. Этот метод требует, чтобы контейнеры были одинакового типа. Метод `swap()` реализован так, что имеет не линейную, а постоянную сложность.

Во всех контейнерных классах имеется метод `clear()`, *удаляющий из контейнера все элементы*, но, в отличие от деструктора, этот метод не уничтожает контейнер.

4. Что собой представляет последовательный контейнер — вектор?

Вектором (одномерным массивом) называется абстрактная модель, имитирующая динамический массив при операциях с элементами. Однако стандарт не утверждает, что в реализации вектора должен использоваться именно динамический массив. Элементы вектора копируются во внутренний динамический массив и хранятся в определенном порядке. Следовательно, вектор относится к категории *упорядоченных коллекций*. Вектор обеспечивает *произвольный доступ* к своим элементам. Это означает, что обращение к любому элементу вектора с известной позицией выполняется напрямую и с постоянным временем. Итераторы векторов являются итераторами прямого доступа, и это позволяет применять к векторам все алгоритмы STL.

Операции присоединения и удаления элементов в конце вектора выполняются с высоким быстродействием. Если элементы вставляются или удаляются в середине или начале вектора, то быстродействие снижается, поскольку все элементы в последующих позициях приходится перемещать на новое место. Один из способов повышения быстродействия векторов заключается в выделении для вектора большего объема памяти, чем необходимо для хранения всех элементов.

5. Что собой представляет последовательный контейнер — вектор логических значений?

Для векторов, содержащих элементы логического типа, в стандартной библиотеке языка C++ определена специализация шаблона классов `vector<bool>`. Сделано это для минимизации затрат памяти — вектор логических значений в классе `vector<bool>` реализован таким образом, что каждый его элемент вместо 1 байта занимает 1 бит. Это в восемь раз экономит оперативную память. Конечно же, оптимизация оперативной памяти достигается не даром. В языке C++ минимальной адресуемой единицей памяти является 1 байт. Следовательно, в классе `vector<bool>` выполняется специальная обработка ссылок и итераторов.

6. Что собой представляет последовательный контейнер — двусторонняя очередь?

Последовательный контейнер `deque` (читается "дек") — это очередь с двумя концами. Дек очень похож на вектор. Он тоже работает с элементами, оформленными в динамический массив, поддерживает произвольный доступ к элементам

и обладает практически тем же интерфейсом. Различие заключается в том, что динамический массив дека открыт с двух сторон. По этой причине дек (двусторонняя очередь) выполняет операции вставки и удаления как с конца, так и с начала очереди за постоянное и небольшое время. Те же операции с элементами внутри очереди являются затратными и занимают время, пропорциональное количеству перемещаемых элементов.

7. Что собой представляет последовательный контейнер — список?

По своей внутренней структуре списки *полностью отличаются* от векторов и деков. К числу наиболее важных особенностей списков относятся следующие. Список *не предоставляет* произвольного доступа к своим элементам. Например, чтобы обратиться к четвертому элементу с начала списка, необходимо перебрать первые три элемента по цепочке ссылок. Поэтому обращение к произвольному элементу списка выполняется относительно медленно. Так как списки не поддерживают произвольный доступ к элементам, то в них не определены ни оператор индексирования "`[]`", ни метод `at()`. Вставка и удаление элементов в любое место списка выполняются *быстро* (за постоянное время), причем не только с его концов. Объясняется это тем, что указанные операции не требуют перемещения других элементов списка. Во внутренней реализации изменяются значения только нескольких указателей. После выполнения операций вставки и удаления элементов списка значения указателей, итераторов и ссылок, относящиеся к другим элементам, остаются *действительными*. Последовательный контейнер `list` реализован в STL в виде *двусвязного (двунаправленного) списка*, каждый узел (элемент) которого содержит ссылки на последующий и предыдущий элементы. Поэтому операции инкремента и декремента для итераторов списка выполняются за постоянное время, а продвижение на n элементов занимает время, пропорциональное n . Обработка исключений в списках реализована так, что практически каждая операция завершается *успешно* или *не вносит изменений*. Таким образом, список не может оказаться в промежуточном состоянии, в котором операция завершена только наполовину. Списки *не поддерживают* операции, связанные с емкостью и перераспределением памяти — такие операции просто не нужны. У каждого элемента списка имеется собственная память, которая остается действительной до удаления элемента. Списки поддерживают много специальных методов для перемещения элементов. Эти методы представляют собой оптимизированные версии одноименных универсальных алгоритмов. Оптимизация основана на замене указателей вместо копирования и перемещения значений.

8. Что собой представляет адаптер последовательного контейнера — стек?

Помимо основных последовательных контейнеров стандартная библиотека языка C++ содержит специальные контейнерные *адаптеры*, к числу которых относятся и *стеки* — контейнеры, элементы которых обрабатываются с использованием дисциплины обслуживания LIFO (последним занесен, первым извлечен). *Стек* — это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняется с одного конца, называемого *вершиной стека*. Другие операции со стеком не определены. В общем случае, стек можно реализовать на основе любого из рассмотренных последовательных контейнеров: вектора, двусторонней очереди или списка. Таким образом, стек является не новым типом контейнера, а вариантом имеющихся контейнеров (*адаптером* контейнеров).

9. Что собой представляет адаптер последовательного контейнера — очередь FIFO?

Помимо основных последовательных контейнеров стандартная библиотека языка C++ содержит специальные контейнерные *адаптеры*, к числу которых относятся и *очереди FIFO*, представляющие собой частный случай однонаправленного списка, добавление элементов в который выполняется в конец, а выборка — из начала. Очередь является *адаптером*, который можно реализовать на основе двусторонней очереди или списка. Обратите внимание, что на базе вектора очередь организовать *нельзя*, поскольку для вектора не определена операция выборки из начала.

Шаблонный класс `queue<>` реализует очередь, работающую по правилу "первым занесен, первым извлечен" (FIFO). Метод `push()` заносит элементы в конец очереди, а метод `pop()` удаляет элементы в порядке их вставки. Таким образом, очередь может рассматриваться как классический буфер данных.

10. Что собой представляет адаптер последовательного контейнера — очередь с приоритетами?

В очереди с приоритетами каждому элементу соответствует приоритет, определяющий порядок выборки из очереди. По умолчанию он определяется с помощью операции `< (less)`. Таким образом, при использовании умолчания из очереди каждый раз выбирается максимальный элемент. Если в очереди существует несколько элементов с "максимальными" приоритетами, то критерий выбора определяется реализацией.

Для реализации очереди с приоритетами подходят последовательные контейнеры, допускающие произвольный доступ к элементам. Таковыми контейнерами являются вектор или двусторонняя очередь.

11. Что представляет собой пара, используемая в ассоциативном контейнере?

Ассоциативные контейнеры обеспечивают быстрый доступ к данным за счет того, что они, как правило, построены на основе сбалансированных деревьев поиска. При этом уместно заметить, что стандартом языка определяется только интерфейс ассоциативных контейнеров, но не их реализация. Типами ассоциативных контейнеров являются словари (`map`), словари с дубликатами (`multimap`), множества (`set`), множества с дубликатами (`multiset`) и битовые множества (`bitset`). *Словарь* построен на основе *пар* "ключ/значение". У каждого элемента словаря имеется ключ, определяющий порядок сортировки элементов, и значение. Иными словами, можно сказать, что ключ *ассоциирован* с элементом. Отсюда и произошло название — *ассоциативный контейнер*. Примером ассоциативного контейнера можно считать англо-русский словарь, в котором ключом является английское слово, а элементом — русское. Массив тоже можно рассматривать как словарь, ключом в котором служит индекс элемента. В словарях, описанных в STL языка C++, в качестве ключа может использоваться значение *произвольного* типа. Каждый ключ должен присутствовать в словаре только в одном экземпляре — дубликаты не разрешаются. *Словарь с дубликатами* представляет собой словарь с возможностью *дублирования* ключей.

Для работы с парами "ключ/элемент" в словарях и словарях с дубликатами используется шаблонный класс `pair`, описанный в заголовочном файле `<utility>` в стандартном пространстве имен `std`.

12. Ассоциативные контейнеры — что представляют собой словари, словари с дубликатами, функциональные классы и функциональные объекты?

Элементами словарей и словарей с дубликатами являются пары "ключ/значение". Сортировка элементов производится автоматически на основании критерия сортировки, применяемого к ключу. По умолчанию используется критерий сортировки `<` (`less`). Словари и словари с дубликатами отличаются только тем, что последние могут содержать дубликаты с одинаковыми значениями ключей, а первые — нет. Ввиду этого в словарях с дубликатами не определена операция индексации `[]`, а добавление с помощью метода `insert()` выполняется успешно в любом случае. Метод `insert()` возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся в словаре с дубликатами в порядке их занесения. При удалении элемента по ключу метод возвращает количество удаленных элементов.

Шаблон словаря содержит три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение "меньше". Кратко остановимся на функциональных классах, предназначенных для перегрузки операций вызова функций.

Класс, в котором определена только операция вызова функции, называется *функциональным классом*. От такого класса не требуется наличия других членов:

```
class greater
{
    public:
        int operator( ) ( int a, int b ) const
        {
            return a>b;
        }
};
```

Использование функционального класса имеет специфический синтаксис:

```
#include <iostream>
// ...
greater      x;           // Создаем объект функции
// Будет выведен 0
cout << x( 1, 5 ) << endl;
// Будет выведена 1
cout << x( 5, 1 ) << endl;
```

Из примера следует, что объект функционального класса используется так, как если бы он был функцией. Попутно заметим, что вместо использования `x(1, 5)` можно было использовать более длинные конструкции `x.operator()(1, 5)` или `greater()(1, 5)`, но это менее удобно.

13. Ассоциативные контейнеры — назовите особенности множеств и множеств с дубликатами.

Множество — это коллекция (набор элементов), в которой элементы сортируются в соответствии с их значениями. Каждый элемент присутствует в коллекции только в одном экземпляре — *дубликаты не разрешаются*. Множество мож-

но считать особой разновидностью словаря, в котором значение идентично ключу. Таким образом, множество — это ассоциативный контейнер, содержащий только значения ключей.

Множество с дубликатами представляет собой множество, в котором могут находиться элементы с одинаковыми значениями. Во множествах с дубликатами ключи могут повторяться и поэтому операция вставки элемента всегда выполняется успешно, а метод `insert()` возвращает итератор на вставленный элемент. Элементы с одинаковыми ключами хранятся во множестве с дубликатами в порядке их занесения. Метод `find()` возвращает итератор на первый найденный элемент или `end()`, если ни одного элемента с заданным ключом не найдено.

14. Специальный контейнер — что представляет собой битовое множество?

Битовое множество представляет собой шаблон для представления и обработки *длинных* последовательностей битов *фиксированного* размера. Напомним, что длинные последовательности битов произвольного размера следует обрабатывать с помощью контейнеров `vector<bool>`. Напомним также, что недлинные последовательности битов можно обрабатывать с помощью битовых операций над целыми операндами или с помощью операций над полями битов. Битовое множество по своей сути представляет собой битовый массив, для которого определены операции произвольного доступа, изменение отдельных битов и массива в целом. Биты множества индексируются *справа налево*, начиная с 0.

15. Напишите законченную программу, в которой создайте матрицу с заданными размерами, инициализируйте ее элементы и напечатайте, используя операцию `[]`. Матрица должна быть представлена в программе как вектор векторов. Используйте последовательный контейнер `вектор`.

Текст программы представлен в листинге П1.28.

Листинг П1.28. Файл `VectorVector.cpp`

```
/*
   Матрица (вектор векторов) на основе последовательного контейнера vector.
   В. Давыдов, консольное приложение, Microsoft Visual Studio C++ .NET
*/
#include <iostream>          // Поточковый ввод-вывод
#include <vector>             // Последовательный контейнер
#include <iomanip>            // Для параметризованных манипуляторов
#include <fstream>           // Файловый ввод-вывод C++
using namespace            // Используем стандартное
    std;                   // пространство имен
// Строчный размер матрицы
const unsigned RowSize = 2;
// Столбцовый размер матрицы
const unsigned ColSize = 5;
int main( void )           // Возвращает 0 при успехе
{
    // Создаем целочисленную матрицу из RowSize пустых строк как вектор
```

```

// векторов
vector< vector< int > >
    matrix( RowSize );
// Резервируем под строки матрицы по ColSize элементов
for( unsigned IxRow = 0; IxRow < RowSize; IxRow++ )
    matrix[ IxRow ].reserve( ColSize );
// Заполняем матрицу по строкам значениями 0, 1, 2, 3, 4, 4, 5, 6, 7,
// 8
for( IxRow = 0; IxRow < RowSize; IxRow++ )
    for( unsigned IxCol = 0; IxCol < ColSize; IxCol++ )
        matrix[ IxRow ][ IxCol ] = IxRow * 4 + IxCol;
// Печатаем матрицу по строкам
// Перенаправляем поток cout в файл scr - такой прием удобен при
// выводе текста на русском языке. В отличие от экранного вывода
// текст в файле scr будет читаемым
ofstream FileScr( "scr" );
cout.rdbuf( FileScr.rdbuf( ) );
cout << setw( 58 ) << "Матрица (вектор векторов) по строкам" << endl
    << setw( 44 ) << "Строк: " << RowSize << endl << setw( 45 )
    << "Столбцов: " << ColSize << endl << endl;
for( IxRow = 0; IxRow < RowSize; IxRow++ )
{
    for( unsigned IxCol = 0; IxCol < ColSize; IxCol++ )
    {
        if( !( IxCol % 4 ) )
            cout << endl;
        cout << setw( 20 ) << matrix[ IxRow ][ IxCol ];
    }
    cout << endl << endl;
}
return 0;
}

```

П1.13. Глава 16

1. Для чего предназначены итераторы?

Итераторы предназначены для просмотра последовательности и обеспечения доступа к каждому ее элементу.

2. Одинакова ли семантика итератора и указателя?

Да, одинакова. По этой причине все функции (методы), принимающие в качестве параметра итераторы, могут использовать и обычные указатели.

3. Какие операции допустимы для любого типа итератора?

Доступ к текущему элементу последовательности выполняется с помощью операций "*" и "->". Переход к следующему элементу выполняется с помощью опе-

рации инкремента "++". Для всех итераторов определены также операции присваивания, проверки на равенство и неравенство.

4. Перечислите категории итераторов, разрешенные для каждой категории операции и области их применения.

Пусть *i* и *j* — итераторы одного типа, *x* — переменная того же типа, что и элемент последовательности, а *n* — целая величина. Тогда для *любого типа итератора* допустимы выражения:

```
i++    ++i    i = j    i == j    i != j
```

В соответствии с набором остальных, варьируемых операций, итераторы делятся на пять групп (табл. П1.1).

Таблица П1.1. Группы итераторов

Категория итератора	Операция	Контейнеры
Входной (input)	<code>x = *i;</code>	Все
Выходной (output)	<code>*i = x;</code>	Все
Прямой (forward)	<code>x = *i; *i = x;</code>	Все
Двунаправленный (bidirectional)	<code>x = *i; *i = x; --i; i--;</code>	Все
Произвольного доступа (random access)	<code>x = *i; *i = x; --i; i--;</code> <code>i+n; i-n; i += n; i -= n;</code> <code>i<j i>j i<=j i>=j</code>	Все, кроме <code>list</code>

5. В каких случаях итератор может быть недействительным?

Итератор недействителен в следующих случаях:

- ☐ если он не был инициализирован;
- ☐ если контейнер, с которым он связан, изменил размеры или уничтожен;
- ☐ если итератор указывает на конец последовательности.

6. В каких случаях следует использовать константные итераторы?

Константные итераторы используются тогда, когда изменять значения соответствующих элементов контейнера не нужно.

7. Перечислите итераторные адаптеры и укажите их назначение.

Для двунаправленных итераторов и итераторов произвольного доступа существуют *итераторные адаптеры* — специальные итераторы, позволяющие выполнять алгоритмы, которые поддерживают:

- ☐ перебор элементов в обратном порядке (обратные итераторы);
- ☐ режим вставки (итераторы вставки);
- ☐ работу с потоками данных (потокковые итераторы).

Итераторы вставки предназначены для добавления новых элементов в начало, конец или произвольное место контейнера.

Итератор входного потока обеспечивает чтение элементов из потока, для которого он был создан. После этого к прочитанному элементу можно обращаться через операцию разадресации:

```
// Чтение целого числа из файла inp
istream      in( "inp" );
istream_iterator< int >
              i( in );
int          tmp = *i;
// Чтение очередного значения из входного потока
++i;
int          tmp1 = *i;
```

При достижении конца входного потока его итератор принимает значение конца ввода. Это же значение имеет умалчиваемый конструктор итератора. Поэтому цикл чтения из файла можно организовать следующим образом:

```
while( i != istream_iterator< int >( ) )
    cout << *i++ << " ";
```

Итератор выходного потока с помощью операции "<<" записывает элементы в выходной поток, для которого он был сконструирован. Если вторым аргументом конструктора итератора была строка символов, то она выводится после каждого выводимого значения.

```
ostream_iterator< int >
              out( cout, " сек.\n" );
*out = 10;                // Будет выведено: 10 сек.
++out; *out = 2;          // Будет выведено: 2 сек.
```

8. Перечислите разновидности функциональных объектов.

Функциональным объектом называют объект с типом класс, в котором определена только операция "()" вызова функции.

Арифметические функциональные объекты. В стандартной библиотеке определены шаблоны функциональных объектов для всех арифметических операций языка C++:

```
plus (бинарная: x + y);
minus (бинарная: x - y);
multiplies (бинарная: x * y);
divides (бинарная: x / y);
modulus (бинарная: x % y);
negative (унарная: - x)
```

Предикаты. В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения и логических операций языка C++:

```
equal_to (бинарная: x == y);
not_equal_to (бинарная: x != y);
greater (бинарная: x > y);
less (бинарная: x < y);
greater_equal (бинарная: x >= y);
less_equal (бинарная: x <= y);
```

```
logical_and (бинарная: x && y);  
logical_or (бинарная: x || y);  
logical_not (унарная: !x)
```

Отрицатели. Отрицатели `not1` и `not2` применяются для получения результата, противоположного унарному и бинарному предикатам соответственно.

Связыватели. Бинарные предикаты стандартной библиотеки, такие как `less`, полезны и достаточно гибки. Однако оказывается, что самый полезный вид предиката — тот, который сравнивает фиксированный аргумент (часто константу) с элементами контейнера. Чтобы в подобном случае использовать тот же самый предикат, требуется связать один из двух его аргументов с фиксированным аргументом (константой). Для этого в стандартной библиотеке используются связыватели `bind1st()` и `bind2nd()`, позволяющие связать с конкретным значением соответственно первый и второй аргумент бинарного предиката стандартной библиотеки.

Адаптеры указателей на функции. Адаптер указателя на функцию позволяет использовать указатель на функцию как аргумент алгоритма.

Адаптеры методов. Адаптер метода класса позволяет использовать метод класса в качестве аргумента алгоритма.

П1.14. Глава 17

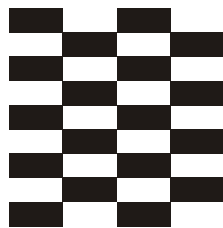
1. В каких файлах содержатся объявления стандартных функциональных объектов и стандартных алгоритмов?

Объявления стандартных алгоритмов находятся в заголовочном файле `<algorithm>`, а стандартных функциональных объектов — в файле `<functional>`.

2. Перечислите основные категории алгоритмов STL.

Все алгоритмы STL можно разделить на пять категорий.

- ☐ Немодифицирующие операции с последовательностями (не меняют значения элементов последовательности или последовательность в целом, извлекают информацию из последовательности или определяют положение элемента в последовательности).
- ☐ Модифицирующие операции с последовательностями (могут изменять последовательность или значения ее элементов).
- ☐ Алгоритмы сортировки последовательностей.
- ☐ Алгоритмы работы с множествами и пирамидами.
- ☐ Обобщенные численные алгоритмы, объявления которых помещены в заголовочный файл `<numeric>`.



Приложение 2

Тесты и программные проекты. Варианты заданий

Далее приводятся варианты тестов по основным темам, рассмотренным в учебном пособии. Их можно использовать при проведении практических занятий и при экзаменационном тестировании. Наряду с тестами в приложении содержатся варианты заданий для курсового проектирования.

П2.1. Классы. Инкапсуляция. Члены класса. Конструкторы и деструктор. Статические члены. Друзья класса. Перегрузка операций. Шаблоны. Варианты тестов

Вариант 1. Член класса имеет спецификатор доступа `private::`. Какие из перечисленных далее функций имеют к нему доступ:

- ☐ методы этого же класса;
- ☐ функции-друзья класса;
- ☐ методы класса, производного от данного класса;
- ☐ обычные функции.

Вариант 2. Член класса имеет спецификатор доступа `protected::`. Какие из перечисленных далее функций имеют к нему доступ:

- ☐ методы этого же класса;
- ☐ функции-друзья класса;
- ☐ методы класса, производного от данного класса;
- ☐ обычные функции.

Вариант 3. Член класса имеет спецификатор доступа `public::`. Какие из перечисленных далее функций имеют к нему доступ:

- ☐ методы этого же класса;
- ☐ функции-друзья класса;
- ☐ методы класса, производного от данного класса;
- ☐ обычные функции.

Вариант 4. Можно ли в блоке класса многократно использовать один и тот же спецификатор доступа (`private:`, `protected:` или `public:`)?

Вариант 5. Какой спецификатор доступа (`private:`, `protected:` или `public:`) имеет по умолчанию член класса, если определение класса начинается со служебного слова `struct`?

Вариант 6. Какой спецификатор доступа (`private:`, `protected:` или `public:`) имеет по умолчанию член класса, если определение класса начинается со служебного слова `class`?

Вариант 7. Какой спецификатор доступа (`private:`, `protected:` или `public:`) имеет по умолчанию член класса, если определение класса начинается со служебного слова `union`?

Вариант 8. Как сделать метод класса подставляемым? Приведите примеры. Чем отличается подставляемый метод от обычного?

Вариант 9. Как выглядит заголовок определения метода класса, если определение помещается вне блока класса? Приведите пример (прототип метода в блоке класса, определение метода вне блока класса и пример вызова метода).

Вариант 10. Как определить класс локальным? Какие особенности имеет локальный класс (область действия, определение метода, возможность использования статических членов)?

Вариант 11. Куда следует помещать определение класса? Как его следует оформлять?

Вариант 12. Как осуществляется доступ к открытым членам некоторого класса? Приведите примеры.

Вариант 13. Укажите назначение конструктора. Перечислите разновидности конструкторов и дайте им краткую характеристику. Укажите особенности оформления конструкторов. Когда и как вызываются конструкторы?

Вариант 14. Укажите назначение деструктора. Укажите особенности оформления деструктора. В каких случаях деструктор вызывается неявно, можно ли его вызывать явно?

Вариант 15. Когда будут вызваны конструкторы и деструкторы для объектов типа `x` (для ответа на этот вопрос укажите, как будут выглядеть строки, выведенные на экран)? Где будет отведена память для каждого объекта (в области обычной или динамической памяти)? Каким значением (и в каких случаях) будут автоматически проинициализированы данные-члены класса?

```
// Конструкторы и деструкторы, статические члены класса
#include <stdio.h>           // Для ввода-вывода
// Определение класса
class X
{
    // Данные
private:
    int      m_x;
    static int count;
```

```

    // Методы
public:
    X( void )
    {
        printf( "Constructor X: count = %d \n", count++ );
    }
    ~X( void )
    {
        printf( "Destructor X: count = %d \n", count-- );
    }
};
// Использование класса
int X ::      count = 0;
X             x1;
int main( void )
{
    //...
    {
        static X
            x3;
        X      *pX = new X;
        X      x2;
        delete pX;
    }
    //...
    return 0;
}

```

Вариант 16. Как сделать члены класса статическими? Особенности использования статических членов-данных класса, их область действия. Можно ли использовать статические члены в локальном классе?

Вариант 17. Как сделать члены класса статическими? Особенности использования статических методов класса, их область действия. Как в статических методах класса использовать обычные, нестатические члены класса?

Вариант 18. Чем отличаются дружественные функции от методов класса? Особенности оформления и использования дружественных функций. Пример (прототип, определение и пример вызова дружественной функции).

Вариант 19. Дано следующее определение класса:

```

// Определение класса
class B
{
    // Данные
protected:
    int      b;
    // Методы

```

```
public:
    B( void )           // Конструктор
    {
        b = 100;
    }
};
```

Напишите прототип, определение и приведите пример вызова дружественной функции, которая выполняет инициализацию данного-члена класса заданным значением.

Вариант 20. Укажите особенности перегрузки унарных операций с помощью методов класса и дружественных функций. Перегрузка префиксной и постфиксной форм операций "++" и "--". Примеры.

Вариант 21. Укажите особенности перегрузки бинарных операций с помощью методов класса и дружественных функций. Можно ли перегрузить операции "=", "[]", "()" и "->" с помощью дружественных функций? Можно ли перегрузить бинарную операцию с помощью метода класса, если левый операнд имеет тип, отличный от типа правого операнда (например, один из predefined типов).

Вариант 22. В приведенном далее программном коде перегрузите операцию суммирования (напишите прототип, поместите его в блок класса, определение перегружающего метода и пример его вызова):

```
// Для работы с геометрическими векторами
class VECTOR
{
    // Данные
private:
    int      x, y;      // Координаты конца вектора
    // Методы
public:
    // Присваивание значений координатам вектора
    void Assign( const int &x1, const int &y1 )
    {
        x = x1; y = y1;
        return;
    }
    // Перегрузка префиксного унарного оператора "++"
    VECTOR operator++( void );
    // Перегрузка бинарной операции "="
    VECTOR & operator=( // Возвращает ссылку на объект (левый операнд)
        // Правый операнд
        const VECTOR &v )
    {
        this->x = v.x; this->y = v.y;
        return *this;
    }
};
```

```
// Перегрузка префиксного унарного оператора "++"
VECTOR VECTOR :: operator++( void )
{
    x++; y++;
    return *this;
}
// Тестирование класса
int main( void )           // Возвращает 0 при успехе
{
    VECTOR    v1;           // Создаем вектор
    v1.Assign( 0, 0 );      // Инициализируем вектор нулями
    v1 = ++v1;              // Тестируем перегруженные
                             // операции

    return 0;
}
```

Вариант 23. Для чего нужен шаблон класса? Особенности определения и размещения в файлах шаблона класса. Можно ли объявление шаблона класса поместить в заголовочный файл, а определения его методов в CPP-файл?

Вариант 24. В приведенном далее программном коде класс превратите в шаблонный класс и протестируйте его для типа `double`.

```
// Для работы с геометрическими векторами
class VECTOR
{
    // Данные
private:
    int    x, y;           // Координаты конца вектора
    // Методы
public:
    // Присваивание значений координатам вектора
    void Assign( const int &x1, const int &y1 )
    {
        x = x1; y = y1;

        return;
    }
    // Перегрузка префиксного унарного оператора "++"
    VECTOR operator++( void );
    // Перегрузка бинарной операции "="
    VECTOR & operator=( // Возвращает ссылку на объект (левый операнд)
        // Правый операнд
        const VECTOR &v )
    {
        this->x = v.x; this->y = v.y;
        return *this;
    }
}
```

```

};
// Перегрузка префиксного унарного оператора "++"
VECTOR VECTOR :: operator++( void )
{
    x++; y++;
    return *this;
}
// Тестирование класса
int main( void )           // Возвращает 0 при успехе
{
    VECTOR    v1;           // Создаем вектор
    v1.Assign( 0, 0 );       // Инициализируем вектор нулями
    v1 = ++v1;              // Тестируем перегруженные
                             // операции

    return 0;
}

```

Вариант 25. Для примера, полученного в варианте 24, перегрузите операцию суммирования и протестируйте ее для типа `double`.

Вариант 26. Для примера, полученного в варианте 24, перегрузите операцию присваивания и протестируйте ее для типа `long double`.

Вариант 27. Дано следующее определение класса:

```

// Определение класса
class B
{
    // Данные
private:
    float    b, k;
    // Методы
public:
    B( void )           // Конструктор
    {
        b = 100; k = 0;
    }
};

```

Напишите прототип, определение и приведите пример вызова дружественной функции, которая выполняет инициализацию данных-членов класса заданными значениями.

Вариант 28. Когда будут вызваны конструкторы и деструкторы для объектов типа `x` (для ответа на этот вопрос укажите, как будут выглядеть строки, выведенные на экран)? Где будет отведена память для каждого объекта (в области обычной или динамической памяти)? Какие члены-данные объектов `x1`, `x2`, `x3` будут автоматически проинициализированы, каким значением и в какой момент (при компиляции или при выполнении программы)?

```

// Конструкторы и деструкторы, статические члены класса
#include <stdio.h>           // Для ввода-вывода

```

```
// Определение класса
class X
{
    // Данные
private:
    int      m_x;
    static int count;
    // Методы
public:
    X( void )
    {
        printf( "Constructor X: count = %d \n", count++ );
    }
    ~X( void )
    {
        printf( "Destructor X: count = %d \n", count-- );
    }
};

// Использование класса
int X ::      count = 3;
X             x1;
int main( void )
{
    //...
    {
        X      x3;
        X      x2;
    }
    //...
    return 0;
}
```

Вариант 29. Когда будут вызваны конструкторы и деструкторы для объектов типа `x` (для ответа на этот вопрос укажите, как будут выглядеть строки, выведенные на экран)? Где будет отведена память для каждого объекта (в области обычной или динамической памяти)? Какие члены-данные объектов `x1`, `x2`, `x3` будут автоматически проинициализированы, каким значением и в какой момент (при компиляции или при выполнении программы)?

```
// Конструкторы и деструкторы, статические члены класса
#include <stdio.h>           // Для ввода-вывода
// Определение класса
class X
{
    // Данные
private:
    int      m_x;
```

```

    static int count;
    // Методы
public:
    X( void )
    {
        printf( "Constructor X: count = %d \n", count++ );
    }
    ~X( void )
    {
        printf( "Destructor X: count = %d \n", count-- );
    }
};
// Использование класса
int X ::      count = 2;
X             x1;
int main( void )
{
    //...
    {
        static X
            x3;
        X      x2;
    }
    //...
    return 0;
}

```

Вариант 30. В приведенном далее программном коде шаблонный класс превратите в класс с типом данных **double** и протестируйте его.

```

// Для работы с геометрическими векторами
template < class T >
class VECTOR
{
    // Данные
private:
    T      x, y;      // Координаты конца вектора
    // Методы
public:
    // Присваивание значений координатам вектора
    void Assign( const T &x1, const T &y1 )
    {
        x = x1; y = y1;

        return;
    }
    // Перегрузка префиксного унарного оператора "++"

```

```

    VECTOR operator++( void );
};
// Перегрузка префиксного унарного оператора "++"
template < class T >
VECTOR VECTOR :: operator++( void )
{
    x++; y++;
    return *this;
}
// Тестирование класса
int main( void )           // Возвращает 0 при успехе
{
    VECTOR < int >
        v1;                // Создаем вектор
    v1.Assign( 0, 0 );      // Инициализируем вектор нулями
    ++v1;                  // Тестируем префиксную
                           // операцию "++"

    return 0;
}

```

Вариант 31. Для примера, полученного в варианте 30, перегрузите постфиксную операцию "++" и протестируйте ее.

Вариант 32. В каких случаях автоматически вызывается конструктор копирования? Что он делает?

Вариант 33. В каких случаях можно использовать умалчиваемый конструктор копирования? Обоснуйте ваш ответ.

Вариант 34. В каких случаях в классе следует спроектировать конструктор копирования? Обоснуйте ваш ответ.

Вариант 35. Чем отличается поверхностное копирование объекта от глубинного?

П2.2. Классы. Наследование. Полиморфизм.

Варианты тестов

Вариант 1. Как вызвать метод базового класса из объекта производного класса, если в производном классе этот метод был замещен?

Вариант 2. Как вызвать метод базового класса из объекта производного класса, если в производном классе этот метод не был замещен?

Вариант 3. Можно ли построить производный класс от базового класса, объявление которого начинается со служебного слова **union**?

Вариант 4. Имеется следующий фрагмент программы:

```

class D: B
{
    ...
};

```

Какой спецификатор доступа действует по умолчанию перед именем базового класса В?

Вариант 5. Имеется следующий фрагмент программы:

```
struct D: B
{
    ...
};
```

Какой спецификатор доступа действует по умолчанию перед именем базового класса В?

Вариант 6. Внешняя функция и метод класса имеют одинаковую сигнатуру. В местах, указанных в следующем далее фрагменте с помощью комментариев, запишите вызовы этих функций.

```
// Внешняя функция
void f( void )
{
    // ...
    return;
}

class B
{
    // ...
public:
    // Метод класса с той же сигнатурой
    void f( void )
    {
        // ...
        return;
    }
};

int main( void )
{
    B          ObjB;
    // ...
    // Здесь запишите вызов внешней функции f( )
    // ...
    // Здесь запишите вызов метода класса f( )
    return 0;
}
```

Вариант 7. Имеется следующий фрагмент программы:

```
class B
{
private:
    int      privi;
protected:
    int      prot1;
```

```
public:
    int      publi;
    // ...
};
class D: private B
{
    // ...
};
```

Как влияет спецификатор доступа **private**: в заголовке объявления производного класса на доступность членов базового класса (влияет или нет)? Мотивируйте ваш ответ. Укажите, какие из членов базового класса доступны в производном классе?

Вариант 8. Имеется следующий фрагмент программы:

```
class B
{
private:
    int      privi;
protected:
    int      proti;
public:
    int      publi;
    // ...
};
class D: protected B
{
    // ...
};
```

Как влияет спецификатор доступа **protected**: в заголовке объявления производного класса на доступность членов базового класса (влияет или нет)? Мотивируйте ваш ответ. Укажите, какие из членов базового класса доступны в производном классе?

Вариант 9. Имеется следующий фрагмент программы:

```
class B
{
private:
    int      privi;
protected:
    int      proti;
public:
    int      publi;
    // ...
};
class D: public B
{
    // ...
};
```

Как влияет спецификатор доступа `public:` в заголовке объявления производного класса на доступность членов базового класса (влияет или нет)? Мотивируйте ваш ответ. Укажите, какие из членов базового класса доступны в производном классе?

Вариант 10. Имеется следующий фрагмент программы:

```
class B
{
private:
    int    privi;
protected:
    int    proti;
public:
    int    publi;
    // ...
};
class D: private B
{
    // ...
};
class DD: public D
{
    // ...
};
```

Как влияет спецификатор доступа `private:` в заголовке объявления производного класса `D` на доступность членов базового класса `B` (влияет или нет на доступность членов класса `B` из класса `D`; влияет или нет на доступность членов класса `B` из класса `DD`)?

Укажите, доступны ли в производном классе `DD` члены базового класса `privi`, `proti`, `pubi`?

Вариант 11. Имеется следующий фрагмент программы:

```
class B
{
private:
    int    privi;
protected:
    int    proti;
public:
    int    publi;
    // ...
};
class D: protected B
{
    // ...
};
class DD: public D
```

```
{  
    // ...  
};
```

Как влияет спецификатор доступа `protected`: в заголовке объявления производного класса `D` на доступность членов базового класса `B` (влияет или нет на доступность членов класса `B` из класса `D`; влияет или нет на доступность членов класса `B` из класса `DD`)?

Укажите, доступны ли в производном классе `DD` члены базового класса `privi`, `proti`, `pubi`?

Вариант 12. Имеется следующий фрагмент программы:

```
class B  
{  
private:  
    int    privi;  
protected:  
    int    proti;  
public:  
    int    pubi;  
    // ...  
};  
class D: public B  
{  
    // ...  
};  
class DD: public D  
{  
    // ...  
};
```

Как влияет спецификатор доступа `public`: в заголовке объявления производного класса `D` на доступность членов базового класса `B` (влияет или нет на доступность членов класса `B` из класса `D`; влияет или нет на доступность членов класса `B` из класса `DD`)?

Укажите, доступны ли в производном классе `DD` члены базового класса `privi`, `proti`, `pubi`?

Вариант 13. Продемонстрируйте на примере механизм передачи данных из конструктора производного класса конструктору базового класса. Укажите последовательность вызовов конструкторов и деструкторов базового и производного классов.

Вариант 14. Что такое множественное наследование? Для чего нужны виртуальные базовые классы? Приведите пример.

Вариант 15. Укажите основные отличия структуры от класса.

Вариант 16. Укажите основные отличия объединения от класса.

Вариант 17. Объявите класс `JetPlane` (реактивный самолет), наследуя его от двух базовых классов: `Rocket` (ракета) и `AirPlane` (самолет).

Вариант 18. Что такое v-таблица?

Вариант 19. Что представляет собой виртуальный деструктор? Когда его необходимо использовать?

Вариант 20. Можно ли объявить виртуальный конструктор? Обоснуйте ваш ответ.

Вариант 21. Если в базовом классе метод объявлен как виртуальный, а в производном классе имеется метод с той же сигнатурой, но его виртуальность не указана, то сохранится ли виртуальность метода в следующем производном классе?

Вариант 22. Объявите в блоке базового класса виртуальный метод с именем `virtFun`, который принимает одно целочисленное значение и не имеет возвращаемого значения.

Вариант 23. Что неправильно в следующем программном коде:

```
class Form
{
    // ...
    // Методы
public:
    Form( void );           // Конструктор умолчания
    // Деструктор
    virtual ~Form( void );
    // Конструктор копирования
    virtual Form( const Form & );
    // ...
};
```

Вариант 24. Если один и тот же метод используется в нескольких производных классах, то как можно улучшить программу?

Вариант 25. Что следует помещать в базовый класс?

Вариант 26. Объявите класс `Computer` как абстрактный класс.

Вариант 27. Если в программе объявлен абстрактный класс с тремя абстрактными методами, то сколько из них нужно заместить в производных классах, чтобы получить возможность создания объектов этих классов?

Вариант 28. Объявите классы `Car` (легковой автомобиль) и `Bus` (автобус), производные от базового класса `Vehicle` (транспортное средство). Опишите класс `Vehicle` как абстрактный класс с двумя абстрактными функциями. Производные классы не должны быть абстрактными.

Вариант 29. Что такое раннее (статическое) связывание и позднее (динамическое) связывание? Приведите примеры подобного связывания.

Вариант 30. Перечислите основные правила работы с виртуальными методами.

Вариант 31. Что такое абстрактный метод класса и абстрактный класс? Зачем они нужны? Можно ли создать объект с типом абстрактного класса? Обоснуйте ваш ответ.

Вариант 32. Перечислите основные правила работы с абстрактными методами и абстрактными классами. Укажите область применения абстрактных классов.

П2.3. Пространства имен. Ввод-вывод в языке C++ средствами стандартной библиотеки. Потокные классы. Обработка исключительных ситуаций. Варианты тестов

Вариант 1. Найдите ошибку в следующем коде:

```
#include <iostream>

int main( void )
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Укажите три способа исправления найденной ошибки.

Вариант 2. Перечислите все известные вам области действия объектов в языке C++. Поясните, какую область действия имеют параметры в прототипе функции, члены класса и члены пространства имен.

Вариант 3. Перечислите пространства имен языка C++, в пределах каждого из которых идентификаторы должны быть уникальными. Охарактеризуйте состав каждого из этих пространств имен.

Вариант 4. Укажите синтаксис и смысл оператора `namespace` объявления (определения) пространства имен. Расширение пространства имен и использование его возможностей. Объявления и определения в пространстве имен. Что целесообразно помещать в оператор `namespace`? Объявление и реализация пространства имен. Приведите примеры.

Вариант 5. Как сделать члены пространства имен доступными в нескольких (в пределе — во всех) файлах программного проекта? Как можно воспользоваться членом из пространства имен? Приведите примеры.

Вариант 6. Синтаксис и назначение операторов `using` и `using namespace`? Как можно обращаться к членам из пространства имен при использовании этих операторов? Взаимный приоритет членов пространства имен при одновременном использовании операторов `using` и `using namespace`.

Вариант 7. Пространство имен стандартной библиотеки языка C++ (`std`). Разновидности стандартных подключаемых файлов и их характеристика. Варианты использования идентификаторов из пространства имен стандартной библиотеки.

Вариант 8. Напишите простую программу, использующую стандартные объекты `cin`, `cout`, `cerr` и `clog` класса `iostream`.

Вариант 9. Напишите программу, предлагающую пользователю ввести свое имя с последующим выводом этого имени на экран.

Вариант 10. Напишите программу, которая выведет заданные аргументы командной строки в обратном порядке.

Вариант 11. Перечислите predefined объекты языка C++ для ввода-вывода и дайте им характеристику.

```
int          i;
float        f;
double       d;
```

Напишите фрагмент программы, обеспечивающий клавиатурный ввод значений указанных ранее объектов. Используйте при вводе predefined поток `cin` и перегруженную операцию `>>`. Предусмотрите проверку достижения конца потока и обработку ошибки ввода. Подключите необходимые заголовочные файлы.

Вариант 12. Определены следующие объекты:

```
int          i = 14;
float        f = 1.5f;
double       d = 12.21;
```

Пользуясь этими объектами, напишите фрагмент программы, обеспечивающий экранный вывод в следующем виде (далее с помощью символа `^` показано местоположение пробельных символов):

```
i^=^^^^^^+14
f^=^1.500
d^=^1.221e+001
```

Используйте при выводе predefined поток `cout`, перегруженную операцию `<<` и необходимые манипуляторы. Подключите необходимые заголовочные файлы.

Вариант 13. Для указанного далее класса перегрузите операцию `>>` для ввода.

```
// Для работы с комплексными данными
class CMP
{
    // Данные
private:
    int      r,          // Действительная часть
           i;          // Мнимая часть

    // Методы
public:
    // Конструктор: подставляемая функция
    CMP( int re, int im )
    {
        r = re; i = im;
    }
    // Здесь поместите прототип функции, перегружающей операцию ">>" для
    //   ввода
};
// Здесь поместите определение функции, перегружающей операцию ">>" для
//   ввода, и пример ее использования
```

Предусмотрите обработку ошибок ввода.

Вариант 14. Для указанного далее класса перегрузите операцию "<<" для вывода.

```
// Для работы с комплексными данными
class CMP
{
    // Данные
private:
    int      r,          // Действительная часть
           i;          // Мнимая часть

    // Методы
public:
    // Конструктор: подставляемая функция
    CMP( int re, int im )
    {
        r = re; i = im;
    }
    // Здесь поместите прототип функции, перегружающей операцию "<<" для
    // вывода
};
// Здесь поместите определение функции, перегружающей операцию "<<" для
// вывода, и пример ее использования
```

Вариант 15. Даны следующие объекты:

```
int      i;          // 2;
float    f;          // -17.4f;
double   d;          // 1.475;
```

Напишите фрагмент программы, обеспечивающий ввод из файла test.inp значений указанных ранее объектов. Используйте при вводе файловый объект и перегруженную операцию ">>". Предусмотрите проверку достижения конца файла и обработку ошибки ввода. Подключите необходимые заголовочные файлы. Укажите вид строк данных, читаемых из файла test.inp.

Вариант 16. Даны следующие объекты:

```
int      i = 14;
float    f = 1.5f;
double   d = 12.21;
```

Напишите фрагмент программы, обеспечивающий вывод в файл test.out значений указанных переменных в следующем виде (далее с помощью символа "^" показано местоположение пробельных символов):

```
i^=^+14^^^^^^+14^f^=^1.5
d^=^12.2100
```

Используйте при выводе файловый объект, перегруженную операцию "<<" и необходимые манипуляторы. Подключите необходимые заголовочные файлы.

Вариант 17. Что такое исключение?

Синтаксис исключений: контролируемый блок (**try**), генерация исключения (**throw**) и обработчики исключений (**catch**). Разновидности обработчиков исключений и их взаимное расположение.

Опишите схему обработки исключения. Что такое раскрутка стека?

Вариант 18. Имеется следующий фрагмент программы:

```
#include <stdio.h>          // Для ввода-вывода
int main( void )           // Возвращает 0 при успехе
{
    // ...
    // Открываем файл для чтения
    if( !( FILE *f_in = fopen( "task1.dat", "r" ) ) )
    {
        printf( "\n Ошибка 10. Файл task1.dat для чтения"
                " не открыт \n\n" );
        return 10;
    }
    // ... Остальная часть блока main
    return 0;
}
```

Выполните обработку ошибки открытия файла путем обработки исключений. Для этого запишите модифицированный фрагмент программы, эквивалентный приведенному ранее.

Вариант 19. Имеется следующий фрагмент программы:

```
#include <stdio.h>          // Для ввода-вывода
int main( void )           // Возвращает 0 при успехе
{
    // ...
    // Открываем файл для чтения
    if( !( FILE *f_in = fopen( "task1.dat", "r" ) ) NULL )
    {
        printf( "\n Ошибка 10. Файл task1.dat для чтения"
                " не открыт \n\n" );
        return 10;
    }
    // Читаем значения аргументов функции
    int a, b;               // Аргументы функции
    int retcode = fscanf( f_in, "%i %i", &a, &b );
    if( retcode != 2 )
    {
        printf( "\n Ошибка 20. Произошла ошибка чтения из"
                " файла task1.dat \n\n" );
        return 20;
    }
    // ... Остальная часть блока main
    return 0;
}
```

Выполните обработку ошибки чтения аргументов путем обработки исключений. Для этого запишите модифицированный фрагмент программы, эквивалентный приведенному ранее.

Вариант 20. Имеется следующий фрагмент программы:

```
#include <stdio.h>          // Для ввода-вывода
int main( void )           // Возвращает 0 при успехе
{
    // ...
    // Открываем файл для чтения
    if( !( FILE *f_in = fopen( "task1.dat", "r" ) ) )
    {
        printf( "\n Ошибка 10. Файл task1.dat для чтения"
                " не открыт \n\n" );
        return 10;
    }
    // Читаем значения аргументов функции
    int      a, b;          // Аргументы функции
    retcode = fscanf( f_in, " %i %i", &a, &b );
    if( retcode != 2 )
    {
        printf( "\n Ошибка 20. Произошла ошибка чтения из
                " файла task1.dat \n\n" );
        return 20;
    }
    // Закрываем файл для чтения
    if( fclose( f_in ) == EOF )
    {
        printf( "\n Ошибка 30. Файл task1.dat не "
                "закрыт \n\n" );
        return 30;
    }
    // ... Остальная часть блока main
    return 0;
}
```

Выполните обработку ошибки закрытия файла ввода путем обработки исключений. Для этого запишите модифицированный фрагмент программы, эквивалентный приведенному ранее.

Вариант 21. Имеется следующий фрагмент программы:

```
#include <stdio.h>          // Для ввода-вывода
int main( void )           // Возвращает 0 при успехе
{
    int      a=1, b=0, // Аргументы функции
            c,         // Значение функции

    // ...
    // Проверяем корректность деления
    if( b==0 )
    {
        printf( "\n Ошибка 40. Делить на 0 нельзя \n" );
    }
}
```

```
        return 40;
    }
    // ... Остальная часть блока main
    return 0;
}
```

Выполните обработку ошибки деления на ноль путем обработки исключений. Для этого запишите модифицированный фрагмент программы, эквивалентный приведенному ранее.

П2.4. Прикладное программирование: сортировка массивов, транспортная задача, поиск в таблице, списки, очереди и стеки, сортировка файлов. Варианты тестов

Для ответа на некоторые из приводимых далее тестов следует изучить в [3] материал, изложенный в *разд. 15* (сортировка массивов), *16* (транспортная задача) и *17* (поиск в таблице).

Вариант 1. Сортировка.

Ответьте, почему для сортировки файлов нецелесообразно использование достаточно эффективных методов сортировки массивов.

Вариант 2. Сортировка массивов.

Что используется в качестве показателей эффективности сортировки массивов? Приведите примеры.

Перечислите простые методы сортировки массивов и расположите их в порядке убывания эффективности. Чему пропорциональны показатели эффективности простых сортировок массивов?

Вариант 3. Сортировка массивов.

Что используется в качестве показателей эффективности сортировки массивов? Приведите примеры.

Перечислите сложные методы сортировки массивов и расположите их в порядке убывания эффективности. Чему пропорциональны показатели эффективности сложных сортировок массивов?

Вариант 4. Сортировка массивов.

Массив имеет следующее начальное состояние:

Начальные значения ключей элементов массива 44 55 12 42

Сколько проходов потребуется при простой сортировке выбором? Укажите состояние массива после каждого прохода.

Вариант 5. Сортировка массивов.

Массив имеет следующее начальное состояние:

Начальные значения ключей элементов массива 44 55 12 42

Сколько проходов потребуется при простой сортировке вставками? Укажите состояние массива после каждого прохода.

Вариант 6. Сортировка массивов.

Массив имеет следующее начальное состояние:

Начальные значения ключей элементов массива 44 55 12 42

Сколько проходов потребуется при простой сортировке обменом? Укажите состояние массива после каждого прохода.

Вариант 7. Сортировка массивов.

Массив имеет следующее начальное состояние:

Начальные значения ключей элементов массива 44 55 12 42

Укажите, на какие подсегменты будет разбит исходный массив после первого разбиения при использовании сортировки Хоора.

Вариант 8. Сортировка массивов.

Требуется отсортировать массив размером n , используя нерекурсивную сортировку Хоора. Какой должен быть при этом минимальный размер стека отложенных сегментов? При каком условии этот размер будет достаточным?

Вариант 9. Сортировка массивов. Транспортная задача.

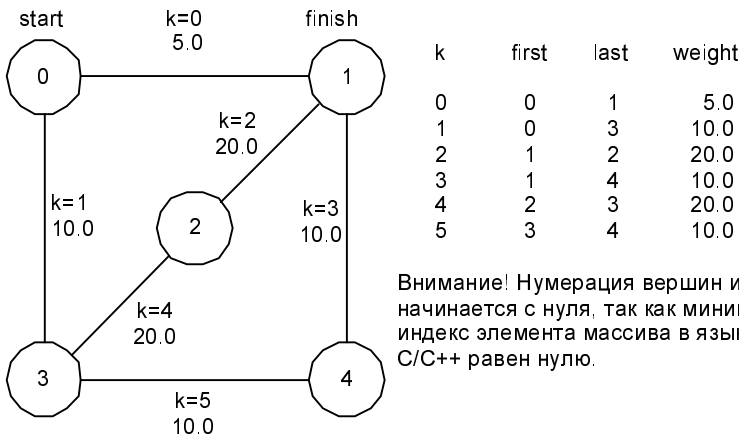
Перечислите основные особенности и правила работы с рекурсивными функциями.

Вариант 10. Транспортная задача.

Укажите возможные формы задания графа и сопоставьте их между собой.

Вариант 11. Транспортная задача.

Задан следующий граф (рис. П2.1).



Внимание! Нумерация вершин и ребер начинается с нуля, так как минимальный индекс элемента массива в языках C/C++ равен нулю.

Рис. П2.1. Пример графа

Информация о наилучшем пути хранится в следующем массиве структур:

```

struct W                                // Way: путь до одной вершины
{
    int      exist;                       // ( != 0 ) - путь имеется
    int      ref;                        // REference: предыдущая вершина, через которую

```

```

                                //  проходит путь
float      SumDist; // Суммарная длина минимального пути
};
// Адрес первого элемента массива структур с информацией о минимальном
//  пути между заданными вершинами
w      *pMinWay;

```

Укажите состояние массива с указателем `pMinWay` после решения транспортной задачи.

Вариант 12. Транспортная задача.

Можно ли утверждать, что в результате решения транспортной задачи будут найдены оптимальные пути из заданной вершины во все остальные вершины? Обоснуйте ваш ответ и для его доказательства приведите примеры топологии соответствующих графов.

Вариант 13. Поиск в таблице.

Чем отличаются внутренний и внешний поиск?

Перечислите известные вам методы внутреннего поиска (поиска в таблице) и укажите значения их показателей эффективности.

Вариант 14. Поиск в таблице.

Файл данных для заполнения таблицы при логарифмическом поиске имеет следующий вид (каждая строка файла предназначена для заполнения отдельной строки таблицы):

```

A      Data for string 0
Z      Data for string 1
C      Data for string 2
D      Data for string 3
E      Data for string 4
F      Data for string 5
G      Data for string 6
H      Data for string 7
I      Data for string 8
J      Data for string 9 (size-1)

```

Подходит ли этот файл (обоснуйте ваш ответ)?

Вариант 15. Поиск в таблице.

Файл данных для заполнения таблицы при логарифмическом поиске имеет следующий вид (каждая строка файла предназначена для заполнения отдельной строки таблицы):

```

A      Data for string 0
C      Data for string 1
D      Data for string 2
E      Data for string 3
F      Data for string 4
H      Data for string 5
I      Data for string 6
J      Data for string 7 (size-1)

```

Изобразите двоичное дерево для логарифмического поиска после его начального заполнения.

Вариант 16. Поиск в таблице.

Укажите, какими основными свойствами должна обладать хэш-функция, укажите ее назначение. Приведите пример хэш-функции.

Вариант 17. Поиск в таблице.

Укажите, в чем состоит конфликт при хэш-поиске, каким образом он разрешается.

Вариант 18. Списки.

Выполните сравнительный анализ линейных и циклических однонаправленных списков. Укажите их области применения.

Вариант 19. Списки.

Выполните сравнительный анализ однонаправленных и двунаправленных списков. Укажите их области применения.

Вариант 20. Очереди.

Перечислите известные вам виды очередей и дайте им характеристику. Какие операции определены над универсальной очередью? Укажите области применения различных видов очередей.

Вариант 21. Списки и очереди.

Можно ли реализовать универсальную очередь неограниченного размера на базе двунаправленного списка? При положительном ответе укажите, что для этого необходимо сделать.

Вариант 22. Списки и очереди.

Что такое стек? Можно ли на базе однонаправленного линейного списка реализовать динамический стек неограниченного размера? При положительном ответе укажите, что для этого необходимо сделать.

Вариант 23. Очереди.

Выполните сравнительный анализ очереди ограниченного размера на базе массива и динамической очереди неограниченного размера. Укажите их области применения.

Вариант 24. Сортировка файлов.

Что используется в качестве показателя эффективности сортировки файлов? Почему при сортировке файлов, в отличие от сортировки массивов, в качестве показателя эффективности не используют количество сравнений элементов? Приведите показатели эффективности сортировок файлов простым и естественным слиянием.

Вариант 25. Сортировка файлов.

Выполните сравнительный анализ сортировок файлов простым и естественным слиянием. Что улучшает сортировка файла естественным слиянием?

Вариант 26. Сортировка файлов.

Проиллюстрируйте алгоритм сортировки файла простым слиянием на примере сортировки файла, содержащего элементы со следующими значениями:

1 2 3 10 8 -14 -13 12 21

Вариант 27. Сортировка файлов.

Проиллюстрируйте алгоритм сортировки файла естественным слиянием на примере сортировки файла, содержащего элементы со следующими значениями:

1 2 3 10 8 -14 -13 12 21

П2.5. Стандартная библиотека языка C++.

Варианты тестов

Вариант 1. Операции над строками и потоковый ввод-вывод.

Напишите законченную программу, в которой создайте четыре строки `s1`, `s2`, `s3` (пустые) и `s4` (инициализируйте ее некоторым значением), в строку `s1` прочтите текст из файла `inp.dat`, строке `s3` присвойте значение строки `s1`, просуммируйте строки `s2=s1+s3+s4`, выведите на экран значения строк `s1`, `s2`, `s3`, `s4` и четвертый символ строки `s2`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 2. Присваивание и добавление частей строк, потоковый ввод-вывод.

Напишите законченную программу, в которой создайте пустые строки `s1`, `s2` и с помощью метода `assign()` инициализируйте их. Добавьте в конец строки `s1` последние два символа строки `s2`. Выведите в файл на магнитном диске `out.dat` значения строк `s1`, `s2`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 3. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте и инициализируйте строки `s1`, `s2`. Выведите на экран их значения. Вставьте в строку `s1`, начиная с позиции 2, строку `s2` и выведите на экран значение строки `s1`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 4. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте и инициализируйте строки `s1`, `s2`. Выведите на экран их значения. Вставьте в строку `s1`, начиная с позиции 3, последние два символа строки `s2` и выведите на экран значение строки `s1`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 5. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте и инициализируйте строку `s`. Выведите на экран ее значение. Удалите из строки `s` два символа, начиная с позиции 3. Выведите на экран значение строки `s`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 6. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте и инициализируйте строки `s1`, `s2`. Выведите на экран их значения. Замените часть строки `s1`, начинающуюся с позиции 1 длиной 3 символа, первыми двумя символами строки `s2`. Выведите на экран значение строки `s1`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 7. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте и инициализируйте строки `s1`, `s2`. Выведите на экран их значения. Выполните обмен содержимого строк и после этого выведите еще раз значения строк на экран.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 8. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте пустую строку `s1` и непустую строку `s2` с ее инициализацией. Выведите на экран их значения. Присвойте строке `s1` значение подстроки `s2`, начинающейся с позиции 2 длиной три символа. Выведите на экран значение строки `s1`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 9. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте и инициализируйте строку `s`. Выведите на экран преобразованное в C-строку с помощью метода `c_str()` значение этой строки.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 10. Преобразования строк и потоковый ввод-вывод.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте и инициализируйте строку `s`. Выведите на экран значение строки `s`. Создайте символьный массив подходящего размера и скопируйте в него с помощью метода `copy()` пять символов строки `s`, начиная с позиции 1. Выведите на экран возвращенное методом `copy()` значение, а также значения первого и последнего скопированных в текстовый массив символов.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

Вариант 11. Последовательный контейнер — вектор, его конструкторы и характеристики.

Напишите законченную программу, в которой с помощью конструктора умолчания, различных разновидностей обычного конструктора и конструктора копирования создаются объекты-векторы. Выведите на экран для контроля значения некоторых из элементов созданных векторов, размеры созданных векторов и количество элементов векторов, которое может храниться без перераспределения памяти.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec1.cpp` из *разд. 15.1*.

Вариант 12. Последовательный контейнер — вектор. Присваивание векторов.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте три вектора `v1`, `v2`, `v3` с элементами целого типа, размерами соответственно 5, 7, 6 и одинаковыми значениями элементов соответственно 1, 2, 3. Выведите на экран размеры векторов, значения их элементов и выполните присваивание `v3=v2-v1`. После этого вновь выведите на экран размеры векторов и значения их элементов.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec2.cpp` из *разд. 15.1*.

Вариант 13. Последовательный контейнер — вектор. Копирование векторов.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте три вектора `v1`, `v2`, `v3` с элементами целого типа, размерами соответственно 4, 5, 7 и одинаковыми значениями элементов соответственно 1, 2, 3. Выведите на экран размеры векторов, значения их элементов. С помощью метода `assign()` первым трем элементам `v1` присвойте значение 4, а первым двум элементам `v2` присвойте значения элементов `v3[4]` и `v3[5]`. После этого вновь выведите на экран размеры векторов и значения их элементов.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec2.cpp` из *разд. 15.1*.

Вариант 14. Последовательный контейнер — вектор. Доступ к элементам вектора.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте вектор `v` с элементами целого типа, размером 5 и одинаковыми значениями элементов 1. Выведите на экран размер вектора и значения его элементов. С помощью операции `[]` второй элемент вектора увеличьте на 3, а с помощью метода `at()` четвертый элемент вектора уменьшите на 1. Вновь выведите на экран размер вектора и значения его элементов. С помощью методов `front()` и `back()` первый элемент вектора уменьшите на 5, а последний — увеличьте на 2.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec3.cpp` из *разд. 15.1*.

Вариант 15. Последовательный контейнер — вектор. Резервирование памяти под вектор и изменение его размера.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте пустой вектор `v1` с элементами целого типа и вектор `v2` размером 4 и значениями элементов 2. С помощью метода `reserve()` зарезервируйте память под вектор `v1` до 15, напечатайте на экране для этого вектора фактический размер и зарезервированный размер. С помощью метода `resize()` измените размер вектора `v2`

до 6, добавленные элементы инициализируйте значением 10, выведите на экран размер вектора `v2` и значения его элементов.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec4.cpp` из *разд. 15.1*.

Вариант 16. Последовательный контейнер — вектор. Изменение вектора.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте векторы `v1`, `v2` с элементами целого типа и размерами 5 и одинаковыми значениями элементов соответственно 1 и -2. Выведите на экран размеры и значения элементов созданных векторов. С помощью метода `push_back()` добавьте элемент со значением 21 в конец вектора `v1` и выведите на экран размер и значения элементов вектора `v1`. С помощью метода `pop_back()` удалите последний элемент вектора `v1` и выведите на экран размер и значения элементов вектора `v1`.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec5.cpp` из *разд. 15.1*.

Вариант 17. Последовательный контейнер — вектор. Изменение вектора.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте векторы `v1`, `v2` с элементами целого типа и размерами 5 и одинаковыми значениями элементов соответственно 1 и -2. Выведите на экран размеры и значения элементов созданных векторов. С помощью метода `insert()` вставьте элемент со значением 12 после третьего элемента вектора `v1` и выведите на экран размер и значения элементов вектора `v1`. С помощью метода `insert()` вставьте два элемента со значением 13 после четвертого элемента вектора `v1` и выведите на экран размер и значения элементов вектора `v1`. С помощью метода `insert()` вставьте в начало вектора `v2` третий и четвертый элементы вектора `v1` и выведите на экран размер и значения элементов вектора `v2`.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec5.cpp` из *разд. 15.1*.

Вариант 18. Последовательный контейнер — вектор. Изменение вектора.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте векторы `v1`, `v2` с элементами целого типа и размерами 5 и одинаковыми значениями элементов соответственно 1 и -2. Выведите на экран размеры и значения элементов созданных векторов. С помощью метода `erase()` удалите второй элемент вектора `v1` и выведите на экран размер и значения элементов вектора `v1`. С помощью метода `erase()` удалите первый и второй элементы вектора `v2` и выведите на экран размер и значения элементов вектора `v2`. С помощью метода `clear()` очистите вектор `v1` и выведите на экран его размер и значения элементов. С помощью метода `swap()` выполните обмен векторов `v1`, `v2` и выведите на экран размеры и значения элементов этих векторов.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec5.cpp` из *разд. 15.1*.

Вариант 19. Последовательный контейнер — вектор. Операции отношений над векторами.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте векторы `v1`, `v2`, `v3`, `v4`, `v5` с элементами целого типа и размерами соответственно 5, 5, 6, 5, 6 и одинаковыми значениями элементов соответственно 2, 2, 2, -2, -2. Выведите на экран размеры и значения элементов созданных векторов. Выведите на экран результаты следующих сравнений векторов: `v1==v2`, `v1==v3`, `v1<v3`, `v1!=v3`, `v1<=v3`, `v3>=v1`, `v3>v1`, `v5>=v4`.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec5.cpp` из *разд. 15.1*.

Вариант 20. Последовательный контейнер — вектор булевских значений.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте вектор булевских значений `v` размером 5 с одинаковыми значениями элементов `true`. Выведите на экран размер и значения элементов созданного вектора с использованием итераторов. Введите с клавиатуры значения элементов вектора `v` с использованием операции `[]`, предусмотрите обработку возможных ошибок ввода. Выведите на экран значения элементов созданного вектора с использованием операции `[]`.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`).

При затруднениях см. файл `vec_bool.cpp` из *разд. 15.2*.

Вариант 21. Последовательный контейнер — двусторонняя очередь.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте очередь `d` размером 5 из элементов целого типа с одинаковыми значениями элементов, равными 1. Выведите на экран размер и значения элементов созданной очереди с использованием операции `[]`. Добавьте два элемента со значениями 7 и 12 в начало очереди (метод `push_front()`) и элемент со значением 21 в конец очереди (метод `push_back()`). Добавьте в очередь элемент со значением 33 после второго элемента (метод `insert()`). Выведите на экран размер и значения элементов созданной очереди с использованием итераторов. Удалите из очереди все добавленные элементы и выведите на экран размер и значения элементов полученной очереди с использованием метода `at()`.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `deque`).

При затруднениях см. файл `deque.cpp` из *разд. 15.3*.

Вариант 22. Последовательный контейнер — список.

Напишите законченную программу, в которой с помощью конструктора умолчания создайте три пустых списка `L1`, `L2`, `L3` из элементов целого типа. В начало первого списка занесите значения 0, 1, 2, 3, 4, в конец второго списка — значения 10, 11, 12 и выполните присваивание `L3=L2`. С помощью итераторов выведите на экран размеры и значения элементов созданных списков.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `list`).

При затруднениях см. файл `list.cpp` из *разд. 15.4*.

Вариант 23. Последовательный контейнер — список.

Напишите законченную программу, в которой аналогично варианту 22 создайте и инициализируйте списки `L1`, `L2` и выведите информацию о них на экран. С помощью метода `splice()` вставьте список `L2` перед вторым элементом списка `L1` и переместите последний элемент списка `L1` в его начало. Выведите на экран размер списка `L1` и его состояние.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `list`).

При затруднениях см. файл `list.cpp` из *разд. 15.4*.

Вариант 24. Последовательный контейнер — список.

Напишите законченную программу, в которой аналогично варианту 22 создайте и инициализируйте списки `L1`, `L2` и выведите информацию о них на экран. С помощью метода `remove()` удалите из списка `L2` элемент со значением 12 и выведите на экран информацию о списке `L2`. С помощью метода `merge()` выполните слияние в список `L1` списков `L1`, `L2` и выведите информацию о списке `L1` на экран. С помощью методов `sort()` и `reverse()` последовательно вначале отсортируйте список `L1` по возрастанию, а затем измените порядок следования элементов на противоположный. После каждой операции выведите информацию о списке `L1` на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `list`).

При затруднениях см. файл `list.cpp` из *разд. 15.4*.

Вариант 25. Стек (`stack`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `vector` три пустых стека `s1`, `s2`, `s3` для хранения целых значений. Стек `s1` инициализируйте значениями из файла `stack.cpp`, стек `s2` — значениями 0, 1, 2, ..., а стек `s3` — значениями 1, 2, 3, С помощью методов `empty()`, `top()` и `pop()` выведите содержимое стека `s1` на экран, после чего стек окажется пустым. Сравните стеки `s2`, `s3` и результат сравнения выведите на экран. С помощью методов `empty()`, `top()` и `pop()` выведите содержимое стеков `s2`, `s3` на экран, после чего стеки окажутся пустыми.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `vector`, `stack`).

При затруднениях см. файл `stack.cpp` из *разд. 15.5*.

Вариант 26. Стек (`stack`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `deque` три пустых стека `s1`, `s2`, `s3` для хранения целых значений. Стек `s1` инициализируйте значениями из файла `stack.cpp`, стек `s2` — значениями 0, 1, 2, ..., а стек `s3` — значениями 1, 2, 3, С помощью методов `empty()`, `top()` и `pop()` выведите содержимое стека `s1` на экран, после чего стек окажется пустым. Сравните

стеки `s2`, `s3` и результат сравнения выведите на экран. С помощью методов `empty()`, `top()` и `pop()` выведите содержимое стеков `s2`, `s3` на экран, после чего стеки окажутся пустыми.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `deque`, `stack`).

При затруднениях см. файл `stack.cpp` из *разд. 15.5*.

Вариант 27. Стек (`stack`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `list` три пустых стека `s1`, `s2`, `s3` для хранения целых значений. Стек `s1` инициализируйте значениями из файла `stack.cpp`, стек `s2` — значениями `0`, `1`, `2`, ..., а стек `s3` — значениями `1`, `2`, `3`, С помощью методов `empty()`, `top()` и `pop()` выведите содержимое стека `s1` на экран, после чего стек окажется пустым. Сравните стеки `s2`, `s3` и результат сравнения выведите на экран. С помощью методов `empty()`, `top()` и `pop()` выведите содержимое стеков `s2`, `s3` на экран, после чего стеки окажутся пустыми.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `list`, `stack`).

При затруднениях см. файл `stack.cpp` из *разд. 15.5*.

Вариант 28. Очередь FIFO (`queue`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `list` три пустых адаптера `queue` `q1`, `q2`, `q3` для хранения целых значений. Адаптер `q1` инициализируйте значениями из файла `queue.cpp`, адаптер `q2` — значениями `0`, `1`, `2`, ..., а адаптер `q3` — значениями `1`, `2`, `3`, С помощью методов `empty()`, `front()` и `pop()` выведите содержимое адаптера `q1` на экран, после чего адаптер окажется пустым. Сравните адаптеры `q2`, `q3` и результат сравнения выведите на экран. С помощью методов `empty()`, `front()` и `pop()` выведите содержимое адаптеров `q2`, `q3` на экран, после чего адаптеры окажутся пустыми.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `list`, `queue`).

При затруднениях см. файл `queue.cpp` из *разд. 15.6*.

Вариант 29. Очередь FIFO (`queue`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `deque` три пустых адаптера `queue` `q1`, `q2`, `q3` для хранения целых значений. Адаптер `q1` инициализируйте значениями из файла `queue.cpp`, адаптер `q2` — значениями `0`, `1`, `2`, ..., а адаптер `q3` — значениями `1`, `2`, `3`, С помощью методов `empty()`, `front()` и `pop()` выведите содержимое адаптера `q1` на экран, после чего адаптер окажется пустым. Сравните адаптеры `q2`, `q3` и результат сравнения выведите на экран. С помощью методов `empty()`, `front()` и `pop()` выведите содержимое адаптеров `q2`, `q3` на экран, после чего адаптеры окажутся пустыми.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `deque`, `queue`).

При затруднениях см. файл `queue.cpp` из *разд. 15.6*.

Вариант 30. Очередь с приоритетами (`priority_queue`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `vector` пустую очередь с приоритетами `pq` для хранения целых значений по убыванию. Очередь с приоритетами `pq` инициализируйте произвольными значениями. С помощью методов `empty()`, `top()` и `pop()` выведите содержимое очереди `pq` на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `vector`, `queue`, `functional`).

При затруднениях см. файл `p_queue.cpp` из *разд. 15.7*.

Вариант 31. Очередь с приоритетами (`priority_queue`).

Напишите законченную программу, в которой создайте на базе последовательного контейнера `deque` пустую очередь с приоритетами `pq` для хранения целых значений по убыванию. Очередь с приоритетами `pq` инициализируйте произвольными значениями. С помощью методов `empty()`, `top()` и `pop()` выведите содержимое очереди `pq` на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `deque`, `queue`, `functional`).

При затруднениях см. файл `p_queue.cpp` из *разд. 15.7*.

Вариант 32. Ассоциативные контейнеры. Словари (`map`).

Напишите законченную программу, в которой создайте пустой словарь — телефонную книгу для хранения записей в лексикографическом порядке. Заполните телефонную книгу данными из файла `map.dat`. Каждая строка файла хранит фамилию абонента и телефонный номер-число, разделенные пробелом. Выведите на экран содержимое телефонной книги. Дополните телефонную книгу новой записью и измените один из номеров телефонной книги. Снова выведите на экран содержимое телефонной книги. Выполните поиск в словаре существующего и несуществующего элемента и выведите результат поиска на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `string`, `map`).

При затруднениях см. файл `map.cpp` из *разд. 15.9*.

Вариант 33. Ассоциативные контейнеры. Словари (`map`).

Напишите законченную программу, в которой создайте две пустых телефонных книги для хранения записей в лексикографическом порядке. Заполните телефонные книги данными из файлов `map1.dat` и `map2.dat`. Каждая строка файла хранит фамилию абонента и телефонный номер-число, разделенные пробелом. Выведите на экран содержимое телефонных книг. Выполните обмен словарей и выведите их на экран. Очистите словари.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `string`, `map`).

При затруднениях см. файл `map.cpp` из *разд. 15.9*.

Вариант 34. Ассоциативные контейнеры. Словари с дубликатами (`multimap`).

Напишите законченную программу, в которой создайте пустую телефонную книгу — словарь с дубликатами — для хранения записей в лексикографическом порядке (фа-

милia абонента — ключ, число — значение, номер телефона). С помощью метода `insert()` заполните телефонную книгу данными, так, чтобы телефонная книга содержала дубликаты с одинаковыми номерами. Выведите на экран содержимое телефонной книги. Выполните поиск в словаре информации с заданными ключами и выведите результаты поиска на экран (ключ с дубликатами, без дубликатов и отсутствующий ключ).

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `string`, `map`).

При затруднениях см. файл `multimap.cpp` из *разд. 15.10*.

Вариант 35. Ассоциативные контейнеры. Множества (`set`).

Напишите законченную программу, в которой с помощью подходящего конструктора создайте множество и инициализируйте его четырьмя строковыми значениями. С помощью итераторов выведите содержимое созданного множества на экран. Добавьте во множество еще два строковых значения, одно из которых уже имеется во множестве. Еще раз выведите содержимое множества на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `string`, `set`).

При затруднениях см. файл `set.cpp` из *разд. 15.11*.

Вариант 36. Ассоциативные контейнеры. Множества с дубликатами (`multiset`).

Напишите законченную программу, в которой с помощью подходящего конструктора создайте множество с дубликатами и инициализируйте его четырьмя строковыми значениями. С помощью итераторов выведите содержимое созданного множества на экран. Добавьте во множество еще два строковых значения, одно из которых уже имеется во множестве. Еще раз выведите содержимое множества на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `string`, `set`).

При затруднениях см. файл `multiset.cpp` из *разд. 15.12*.

Вариант 37. Специальные контейнеры. Битовые множества (`bitset`).

Напишите законченную программу, в которой с помощью различных конструкторов создайте и инициализируйте несколько битовых множеств. Выведите их значения на экран. Измените значения некоторых битов одного из множеств с использованием операции `[]` и выведите значение этого множества на экран. Примените к двум множествам операции составного умножения, сложения, сдвига влево или вправо и т. п. Выведите на экран результаты этих операций.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, классы `string`, `bitset`).

При затруднениях см. файл `bitset.cpp` из *разд. 15.13*.

Вариант 38. Обратные итераторы.

Выполните задание из приведенного ранее варианта 12. При выводе на экран значений элементов векторов используйте обратные итераторы.

При затруднениях см. *разд. 16.2*.

Вариант 39. Функциональные объекты. Связыватели.

Напишите законченную программу, в которой создайте и инициализируйте целочисленный массив. Выведите значения его элементов на экран. Выведите на экран количество элементов массива со значениями меньше 5 и больше 5. С этой целью используйте стандартный алгоритм `count_if` (см. разд. 17.1), в котором в качестве третьего аргумента примените связыватели `bind1st` и `bind2st` (см. разд. 16.5).

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `bind.cpp` из разд. 16.5.

Вариант 40. Алгоритмы. Немодифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте и инициализируйте два целочисленных вектора размерами соответственно 4 и 2 элемента. С помощью метода `for_search()` и функции

```
// Функция, вызываемая для каждого элемента вектора
void show(
    int      a )
{
    cout << a << " ";
    return;
}
```

выведите значения их элементов на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_nomodif.cpp` из разд. 17.1.

Вариант 41. Алгоритмы. Немодифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте и инициализируйте целочисленный вектор. Выведите значения его элементов на экран.

С помощью метода `find()` выполните поиск в векторе элемента с заданным значением и выведите результаты поиска на экран. С помощью метода `count()` подсчитайте в векторе количество элементов с заданным значением и выведите результаты поиска на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_nomodif.cpp` из разд. 17.1.

Вариант 42. Алгоритмы. Немодифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте и инициализируйте целочисленный вектор. Выведите значения его элементов на экран.

С помощью метода `find-if()` и функции

```
// Функция для поиска первого нечетного элемента вектора
```

```
bool odd(           // Возвращает true, если нечетное
    int    a )
{
    if( a%2 )
        return true;
    return false;
}
```

выполните поиск в векторе первого элемента с нечетным значением и выведите результаты поиска на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_nomodif.cpp` из *разд. 17.1*.

Вариант 43. Алгоритмы. Немодифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте и инициализируйте целочисленный вектор. Выведите значения его элементов на экран. С помощью метода `adjacent_find-if()` выполните поиск в векторе первой пары элементов со значениями "меньше-больше" и выведите результаты поиска на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_nomodif.cpp` из *разд. 17.1*.

Вариант 44. Алгоритмы. Немодифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте и инициализируйте два целочисленных вектора `v`, `v1` размерами соответственно 4 и 2 элемента. Выведите значения их элементов на экран. С помощью метода `find-end()` выполните поиск последнего вхождения `v1` в `v` и выведите результаты поиска на экран, в том числе напечатайте значение найденного элемента. С помощью метода `find-first_of()` выполните поиск первого вхождения в `v` элемента из `v1` и выведите результаты поиска на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_nomodif.cpp` из *разд. 17.1*.

Вариант 45. Алгоритмы. Немодифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте и инициализируйте два целочисленных вектора `v`, `v1` размерами соответственно 4 и 2 элемента. Выведите значения их элементов на экран. Проверьте `v`, `v1` на несовпадение и совпадение соответственно с помощью методов `mismatch()` и `equal()` и выведите результаты проверки на экран. С помощью метода `search()` выполните проверку вхождения в `v1` вектора `v` и выведите результаты проверки на экран.

Используйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector`, объявления стандартных функциональных объектов из файла `<functional>` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_nomodif.cpp` из *разд. 17.1*.

Вариант 46. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте два целочисленных массива `a[5]` и `b[4]`. Первый массив инициализируйте значениями 1, 2, 3, 4, 5. С помощью алгоритма `copy()` скопируйте последние четыре элемента массива `a` в массив `b`. Выведите значения элементов `b` на экран. С помощью алгоритма `copy()` скопируйте последние четыре элемента массива `a` в его начало и выведите значения элементов `a` на экран. С помощью алгоритма `copy_backward()` скопируйте первые три элемента массива `b` справа налево в этот же массив, начиная копирование в четвертый элемент этого же массива. Выведите значения элементов `b` на экран. С помощью алгоритма `copy()` выведите на экран значения элементов массива `a`. С этой целью в качестве третьего аргумента в вызове алгоритма `copy()` используйте потоковый итератор `ostream_iterator< int >(cout, " ")`.

Применяйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_copy.cpp` из *разд. 17.2*.

Вариант 47. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте целочисленный массив `a[5]`. С помощью алгоритма `fill()` заполните массив единичными значениями и выведите значения элементов массива на экран. С помощью алгоритма `fill_n()` заполните третий и четвертый элементы массива нулевыми значениями и выведите значения элементов массива на экран.

Применяйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_fill.cpp` из *разд. 17.2*.

Вариант 48. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте целочисленный массив `a[5]`. С помощью алгоритма `generate()` заполните массив значениями, возвращаемыми функцией

```
// Функция, вычисляющая значения для заполнения последовательности
int f( void )
{
    static int i = 1;
    return ( ++i ) * 3;
}
```

и выведите значения элементов массива на экран.

Применяйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_gen.cpp` из *разд. 17.2*.

Вариант 49. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте целочисленные массивы `a[5]={1,2,3,4,5}` и `b[5]={11,12,13,14,15}`. С помощью алгоритма `iter_swap()` поменяйте местами первый элемент `a` и пятый элемент `b` и выведите значения элементов массивов на экран. С помощью этого же алгоритма поменяйте местами первый

и последний элементы массива `a` и выведите значения элементов массива `a` на экран. С помощью алгоритма `swap_ranges()` поменяйте местами первые два элемента массива `a` с четвертым и пятым элементами массива `b` и выведите значения элементов массивов на экран. С помощью этого же алгоритма поменяйте местами первые два элемента массива `a` с его четвертым и пятым элементами и выведите значения элементов массива `a` на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_swap.cpp` из *разд. 17.2*.

Вариант 50. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте целочисленный массив `a[5]={1,2,3,4,5}`. С помощью алгоритма `random_shuffle()` перетасуйте элементы случайным образом и выведите значения элементов массива на экран. С помощью этого же алгоритма перетасуйте часть массива — первые три элемента — и выведите значения элементов массива `a` на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_rand.cpp` из *разд. 17.2*.

Вариант 51. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте два целочисленных вектора `a` — пустой и `b` размером 10 элементов с нулевыми значениями. Занесите в вектор `a` последовательность значений 0, 1, 2, 3, 4, 0, 1, 2, 3, 4. Выведите значения элементов созданных векторов на экран. С помощью алгоритма `remove()` удалите из `a` элементы со значением 2 и выведите значения элементов вектора `a` на экран. С помощью алгоритма `remove_copy()` скопируйте `a` в `b` с удалением в копии элементов со значением 3 и выведите значения элементов обоих векторов на экран. С помощью алгоритма `remove_if()` удалите из `a` элементы со значениями, превышающими единицу. С этой целью в качестве третьего аргумента в вызове алгоритма используйте функцию-предикат

```
// Функция-предикат для выбора элементов вектора
bool great_1(           // Возвращает true при x>1
    int    x )         // Проверяемое значение
{
    return x>1;
}
```

Выведите значения элементов вектора `a` на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_remove.cpp` из *разд. 17.2*.

Вариант 52. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте два целочисленных вектора `v1` — пустой и `v2` размером 10 элементов с нулевыми значениями. Занесите в вектор `v1` последовательность значений 10, 20,

30, 40, 50, 60, 70, 80, 90. Выведите значения элементов созданных векторов на экран. С помощью алгоритма `replace_copy_if()` скопируйте `v1` в `v2` с заменой в копии значений больших 30 и меньших 70 на значение 5. С этой целью в качестве третьего аргумента в вызове алгоритма используйте функцию-предикат вида

```
// Функция-предикат для выбора элементов вектора
bool In_30_70(           // Возвращает true при 30 < x < 70
    int x )              // Проверяемое значение
{
    return x>30 && x<70;
}
```

Выведите значения элементов обоих векторов на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_replace.cpp` из *разд. 17.2*.

Вариант 53. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте два целочисленных массива `a` и `b` одинакового размера и инициализируйте их. Выведите значения элементов созданных массивов на экран. С помощью алгоритма `transform()` выполните попарную обработку элементов этих массивов в соответствии с выражением $a_i = a_i^2 - b_i^2$. С этой целью в качестве пятого аргумента в вызове алгоритма используйте функциональный объект вида:

```
// Функциональный объект пользователя, выполняющий обработку пар
// элементов двух последовательностей
struct conv : binary_function < double, double, double >
{
    double operator( ) ( double x, double y ) const
    {
        return x*x - y*y;
    }
};
```

Выведите значения элементов массива `a` на экран. С помощью алгоритма `transform()` выполните инвертирование значений элементов вектора `b`. С этой целью в качестве четвертого аргумента в вызове алгоритма используйте предикат `negate< int >`. Выведите значения элементов массива `b` на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, объявления стандартных алгоритмов из файла `<algorithm>` и объявления стандартных функциональных объектов из файла `<functional>`).

При затруднениях см. файл `alg_modif_transform.cpp` из *разд. 17.2*.

Вариант 54. Алгоритмы. Модифицирующие операции с последовательностями.

Напишите законченную программу, в которой создайте три целочисленных массива `a`, `b` и `c` одинакового размера. Инициализируйте массивы `a` и `c` так, чтобы они содержали одинаковые элементы. Выведите значения элементов массивов `a` и `c` на экран. С помощью алгоритма `unique()` удалите в последовательностях элементов

массива `a` повторяющиеся элементы и выведите на экран значения элементов этого массива. С помощью алгоритма `unique_copy()` выполните копирование массива `c` в массив `b` с удалением в последовательностях элементов массива `b` повторяющихся элементов и выведите на экран значения элементов массива `b`.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_modif_unique.cpp` из *разд. 17.2*.

Вариант 55. Алгоритмы, связанные с сортировкой.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте три вектора вещественного типа `v`, `v1` и `v2`. Инициализируйте первые два вектора случайными значениями, а вектор `v2` так, чтобы он содержал две отсортированных последовательности. Выведите значения элементов этих векторов на экран. С помощью алгоритмов `sort()` и `stable_sort()` выполните соответственно обычную и устойчивую сортировку векторов `v`, `v1` и выведите на экран значения элементов этих векторов. С помощью алгоритма `binary_search()` выполните поиск в векторе `v` заданного значения и выведите результаты поиска на экран. С помощью алгоритма `inplace_merge()` выполните слияние двух отсортированных последовательностей в векторе `v2` и выведите значения элементов этого вектора на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, стандартный класс `vector` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_sort.cpp` из *разд. 17.3*.

Вариант 56. Алгоритмы, связанные с сортировкой.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте два вектора вещественного типа `v`, `v1` одинакового размера. Инициализируйте оба вектора отсортированными и одинаковыми значениями. Выведите значения элементов этих векторов на экран. С помощью алгоритма `lexicographical_compare()` сравните векторы `v`, `v1` между собой и выведите на экран результаты сравнения. Измените значение одного из элементов вектора `v1`. С помощью того же алгоритма сравните векторы `v`, `v1` между собой с использованием предиката `greater<...>` и выведите на экран результаты сравнения. С помощью алгоритмов `lower_bound()` и `upper_bound()` определите соответственно значения элементов вектора `v`, после которого и перед которым можно вставить элемент с заданным значением, не нарушая упорядоченности вектора, и выведите значения элементов вектора `v` на экран.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, стандартный класс `vector`, объявления стандартных алгоритмов из файла `<algorithm>` и объявления стандартных функциональных объектов из файла `<functional>`).

При затруднениях см. файл `alg_sort.cpp` из *разд. 17.3*.

Вариант 57. Алгоритмы, связанные с сортировкой.

Напишите законченную программу, в которой с помощью подходящих конструкторов создайте два вектора вещественного типа `v1`, `v2` размером 3. Инициализируйте

оба вектора так, чтобы они были отсортированными. С помощью подходящего конструктора создайте вещественный вектор `v3` размером 6. Выведите значения элементов векторов `v1`, `v2` на экран. С помощью алгоритмов `max_element()` и `min_element()` определите и выведите на экран соответственно максимальное и минимальные значения элементов вектора `v1`. С помощью алгоритма `merge()` слейте отсортированные векторы `v1`, `v2` в вектор `v3` и выведите на экран значения элементов вектора `v3`. С помощью алгоритмов `next_permutation()` и `prev_permutation()` сгенерируйте и выведите на экран перестановки элементов в лексикографическом порядке для векторов `v1`, `v2` соответственно.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, стандартный класс `vector` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_sort.cpp` из *разд. 17.3*.

Вариант 58. Алгоритмы работы с множествами.

Напишите законченную программу, в которой создайте целочисленный массив `a` со значениями элементов 2, 5, 7, 9, массив `b` со значениями элементов 1, 5, 9 и массив `tmp` из семи элементов. Выведите значения элементов массивов `a`, `b` на экран. С помощью алгоритма `set_intersection()` создайте отсортированное пересечение массивов `a`, `b`, поместите его в массив `tmp` и выведите на экран значения элементов массива `tmp`. С помощью алгоритма `set_union()` создайте отсортированное объединение массивов `a`, `b`, поместите его в массив `tmp` и выведите на экран значения элементов массива `tmp`. С помощью алгоритма `set_difference()` скопируйте в `tmp` из массивов `a`, `b` значения элементов, входящих только в массив `a` и выведите на экран значения элементов массива `tmp`. С помощью алгоритма `sym_difference()` скопируйте в `tmp` из массивов `a`, `b` значения элементов, входящих только в один из массивов, и выведите на экран значения элементов массива `tmp`. С помощью алгоритма `includes()` проверьте включение массива `b` в массив `a` и выведите на экран результаты проверки.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_set.cpp` из *разд. 17.4*.

Вариант 59. Алгоритмы работы с пирамидами.

Напишите законченную программу, в которой с помощью подходящего конструктора создайте целочисленный вектор `Number` со значениями элементов 4, 10, 70, 10, 30, 69, 96, 100 и выведите значения элементов вектора на экран. С помощью алгоритма `make_heap()` преобразуйте `Number` в пирамиду и выведите на экран значения элементов `Number`. С помощью алгоритма `sort_heap()` преобразуйте пирамиду в отсортированную последовательность и выведите на экран значения элементов `Number`. С помощью метода `push_back()` и алгоритма `push_heap()` добавьте элемент в `Number` и выведите на экран значения элементов `Number`. С помощью алгоритма `make_heap()` преобразуйте `Number` в пирамиду и выведите на экран значения элементов `Number`. Извлеките и выведите на экран корневой элемент пирамиды. Затем с помощью алгоритма `pop_heap()` восстановите пирамиду и выведите на экран значения элементов `Number`.

Применяйте только средства стандартной библиотеки языка C++ (поточный ввод-вывод, класс `vector` и объявления стандартных алгоритмов из файла `<algorithm>`).

При затруднениях см. файл `alg_hear.cpp` из *разд. 17.4*.

П2.6. Экзаменационное тестирование

Наряду с традиционной формой, *экзаменационное тестирование* можно проводить и в более современной на наш взгляд форме — *в форме тестовых вопросов*.

На экзамене каждому студенту может быть предложена комплексная проверочная работа, содержащая пять вопросов по некоторым из перечисленных далее основных разделов курса.

- ☐ Классы. Инкапсуляция. Члены класса. Конструкторы и деструктор. Статические члены. Друзья класса. Перегрузка операций. Шаблоны.
- ☐ Классы. Наследование. Полиморфизм.
- ☐ Пространства имен. Ввод-вывод в языке C++ средствами стандартной библиотеки. Поточные классы. Обработка исключительных ситуаций.
- ☐ Прикладное программирование: сортировка массивов, транспортная задача, поиск в таблице, списки, очереди и стеки, сортировка файлов.
- ☐ Стандартная библиотека языка C++.

Комплексная проверочная работа рассчитана на 1 ч. 30 мин. Ответ на каждый тестовый вопрос, в зависимости от правильности и полноты, оценивается 0, 0,25, 0,5, 0,75 или 1 баллом. Таким образом, максимальная сумма баллов может достигнуть 5.

В соответствии с набранными баллами выставляются следующие экзаменационные оценки:

- ☐ *отлично* (4,25—5 баллов);
- ☐ *хорошо* (3,5—4 балла);
- ☐ *удовлетворительно* (2,5—3,25 балла);
- ☐ *неудовлетворительно* (менее 2,5 баллов).

Представим пример варианта комплексной экзаменационной работы.

Вариант 1.

1. Классы. Инкапсуляция.

Член класса имеет спецификатор доступа `private`. Какие из перечисленных далее функций имеют к нему доступ:

- ☐ методы этого же класса;
- ☐ функции-друзья класса;
- ☐ методы класса, производного от данного класса;
- ☐ обычные функции.

2. Классы. Наследование.

Как вызвать метод базового класса из объекта производного класса, если в производном классе этот метод был замещен?

3. Пространства имен.

Найдите ошибку в следующем коде:

```
#include <iostream>
int main( void )
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Укажите три способа исправления найденной ошибки.

4. Сортировка массивов.

Что используется в качестве показателей эффективности сортировки массивов? Приведите примеры.

Перечислите сложные методы сортировки массивов и расположите их в порядке убывания эффективности. Чему пропорциональны показатели эффективности сложных сортировок массивов?

5. Стандартная библиотека. Строки и потоковый ввод-вывод.

Напишите законченную программу, в которой создайте четыре строки `s1`, `s2`, `s3` (пустые) и `s4` (инициализируйте ее некоторым значением), в строку `s1` прочтите текст из файла `inp.dat`, строке `s3` присвойте значение строки `s1`, просуммируйте строки `s2=s1+s3+s4`, выведите на экран значения строк `s1`, `s2`, `s3`, `s4` и четвертый символ строки `s2`.

Используйте только средства стандартной библиотеки языка C++ (потоковый ввод-вывод, класс `string`).

П2.7. Курсовое проектирование.

Варианты заданий

На практических занятиях и в процессе самостоятельной работы студенты выполняют курсовую работу, целью которой является освоение технологии объектно-ориентированного программирования и технологии обобщенного программирования (стандартная библиотека языка C++). В процессе выполнения курсовой работы необходимо спроектировать и отладить две программы для решения одной и той же предложенной задачи и для первой программы оформить программную документацию (о программной документации см. [3]).

Первая программа должна использовать технологию ООП. В рамках этой программы следует спроектировать базовый и производный классы или их шаблоны и выполнить их тестирование. Для ввода-вывода следует применить средства языка C++, рассмотренные ранее в гл. 5 и 14. Основными этапами выполнения этой программы должны быть:

1. Разработка спецификации.

На этом этапе следует указать назначение каждого из проектируемых классов и сформулировать требования к обработке ошибок и предупреждений, сгруппиро-

вав их по классам. Для каждого предупреждения или сообщения об ошибке в начале диагностического сообщения нужно напечатать номер предупреждения или сообщения об ошибке, обеспечив нумерацию в возрастающем порядке. Предупреждения выдавайте в файл результатов, а сообщения об ошибках — на экран. Каждый из классов следует оформить в виде заголовочного файла, приняв меры для предотвращения возможности их многократного включения. Для каждого из классов нужно перечислить его члены, указав назначение члена класса и модификатор доступа к нему. Для методов класса дополнительно укажите список параметров, их назначение и что метод возвращает, если это требуется.

2. *Разработка программных текстов.*

На этом этапе разработайте программные тексты для каждого из классов и текст тестирующей программы.

3. *Отладка программного проекта.*

Оформление *программной документации* в соответствии с Единой Системой Программной Документации (ЕСПД) в виде отчета.

Перечислим разделы отчета.

- ☐ *Техническое задание* — формулировка решаемой задачи, требования к программе, язык программирования.
- ☐ *Текст программы.* В этом разделе для каждого из спроектированных классов следует указать его назначение и привести листинг с исходным текстом в самодokumentированном виде.
- ☐ *Описание программы.* Здесь нужно привести структуру программного проекта (схему-рисунок с указанием заголовочных файлов с определением спроектированных классов, расположенных в них данных и методов, связей между методами, назначением членов каждого класса и спецификаторов доступа к ним) и, при необходимости, описать метод решения задачи.
- ☐ *Программа и методика испытаний.* Здесь следует привести текст тестирующей программы и набор контрольных примеров с его обоснованием (о контрольных примерах см. *разд. ПЗ.2.3 приложения 3*).

Вторая программа для решения той же самой задачи должна использовать *технологии обобщенного программирования*. Это означает, что при решении задачи следует *максимально* базироваться на возможностях подробно описанной ранее стандартной библиотеки языка C++.

Далее приводятся возможные варианты заданий на выполнение курсовой работы.

П2.7.1. Обработка текстов. Варианты заданий

Вариант 1. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('.', '?'). Составить программу, которая в заданном тексте убирает лишние пробелы между словами, оставляя их по одному. В файле результатов должен быть исходный и преобразованный тексты.

Вариант 2. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов.

Исходный текст должен заканчиваться точкой ('!', '?'). Составить программу, которая в заданном тексте находит слово (слова) максимальной длины. В файле результатов должен быть исходный текст, значение максимальной длины слова, список найденных слов (через запятую) и их количество.

Вариант 3. Исходные данные представлены в двух файлах. Первый файл содержит словарь русских слов (прописными буквами), разделенных запятыми. После последнего слова должна стоять точка. Длина текста — не более `NUMW` слов, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Второй файл содержит приставки, допустимые правилами словообразования. Составить программу, которая в файл вывода выводит исходный текст, список существующих приставок и преобразованный текст, в котором найденные приставки отделены от остальной части слова символом '- '.

Вариант 4. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). Составить программу, которая определяет, сколько слов содержат 1 слог, 2 слога и т. д. В файле результатов должен быть исходный текст и список слов (разделенных запятыми) с 1 слогом, 2 слогами и т. д.

Вариант 5. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном неколецевом списке. Для каждого слова хранить также число согласных букв в слове. В полученном линейном списке найти слова, в которых количество согласных превышает заданное значение. Заданное значение содержится в первой строке исходного файла. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 6. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном неколецевом списке. Для каждого слова хранить также число прописных букв в нем. В полученном линейном списке найти слова, в которых количество прописных букв превышает заданное значение. Заданное значение содержится в первой строке исходного файла. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 7. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном неколецевом списке. В полученном линейном списке найти слова, начинающиеся и заканчивающиеся заданной буквой. В качестве заданной буквы используйте последнюю русскую букву преобразованного исходного текста. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 8. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов.

Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. Полученный линейный список отсортировать. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и отсортированного списка.

Вариант 9. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. В полученном линейном списке найти слова, начинающиеся с гласных букв. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 10. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. Для каждого слова хранить также количество повторов этого слова. В полученном линейном списке найти слова, повторяющиеся заданное число раз. Заданное число повторов содержится в первой строке исходного файла. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 11. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. В полученном линейном списке найти слова, состоящие только из прописных букв. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 12. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться символом '/'. После обработки исходного текста полученные слова хранить в однонаправленном линейном не кольцевом списке. В полученном линейном списке найти слово (слова), встречающееся максимальное число раз. В файл результатов напечатать исходный текст (эхо-печать), состояние сформированного линейного списка и найденные слова.

Вариант 13. Составить программу для замены одной цепочки символов на другую (с тем же количеством символов). Первая строка текста содержит две цепочки символов (что на что заменить), разделенных символом '/'. В конце первой строки (через пробел после второй цепочки) стоит символ '*', отделяющий цепочки от основного текста. Исходный текст на русском языке начинается со второй строки. Длина преобразованного текста — не более `NSYM` символов, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). В файле результатов должен быть исходный текст, затем цепочки символов (что на что заменяется) и преобразованный текст.

Вариант 14. Задан исходный текст на русском языке (в тексте не должно быть английских букв и специальных символов). Длина текста — не более `NL` строк, длина

строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?'). Составить программу, которая определяет частоту повтора русских букв в преобразованном тексте. В файле результатов должен быть исходный текст и список найденных русских букв в алфавитном порядке с указанием частоты их повтора.

Вариант 15. Задан исходный текст на русском языке с произвольным набором фамилий с инициалами, разделенных запятыми. Количество фамилий — не более `NF`, длина строки — не более `NS` символов, длина фамилии — не более `LF`. Исходный текст должен заканчиваться символом '/'. В файле результатов должен быть исходный текст и список фамилий с инициалами в алфавитном порядке.

Вариант 16. Задан текст телеграммы на русском языке. Длина текста не превышает `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('?', '!', '...'). Написать программу для подсчета стоимости телеграммы по следующим правилам: стоимость = сумма стоимости всех предложений + стоимость количества предложений. Стоимость предложения определяется следующим образом: если в предложении 1..5 слов — 5 рублей, 6..10 слов — 15 рублей, 11..15 слов — 45 рублей, от 16 и выше — 100 рублей. Стоимость количества предложений определяется так: 1..2 предложения — 1 рубль, 3..4 предложения — 3 рубля, 4..5 предложений — 9 рублей, от 6 и выше — 50 рублей. В файле результатов должен быть текст телеграммы и ее стоимость.

Вариант 17. Составить программу для шифровки русского текста. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходные данные содержатся в двух файлах. В первом — текст для шифровки, который должен заканчиваться точкой ('!', '?', '...'). Во втором — сам шифр в следующем виде:

аб...я (что заменить),

яп...у (на что заменить).

В файле результатов должен быть шифр, исходный и зашифрованный тексты.

Вариант 18. Составить программу для нахождения в русском тексте слов с заданным количеством букв. Количество букв задается в первой строке. Текст начинается со второй строки. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?', '...'). В файле результатов должен быть исходный текст, сообщение (слова с каким количеством букв искали) и найденные слова (с новой строки каждое и с порядковым номером).

Вариант 19. Составить программу для нахождения в русском тексте определенного словосочетания. Словосочетание вводится в первой строке и заканчивается символом '*' (например, человек*). Текст начинается со второй строки. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?', '...'). В файле результатов должен быть исходный текст, искомое словосочетание и количество обнаруженных словосочетаний заданного вида.

Вариант 20. В начале исходного текста стоит заглавная буква русского алфавита для поиска с символом ':' (например, а:). Далее идет словарь русских слов (прописными буквами), разделенных запятыми. После последнего слова должна

стоять точка. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Найти слова, которые начинаются и заканчиваются на букву (прописную), соответствующую заданной. В файле результатов должен быть исходный словарь, буква для поиска и список найденных слов, каждое с новой строки и с порядковым номером.

Вариант 21. В начале исходного текста стоит заглавная буква русского алфавита для поиска с символом ':' (например, `A:`). Далее идет словарь русских слов (прописными буквами), разделенных запятыми. После последнего слова должна стоять точка. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Найти слова, в которых нет буквы (прописной), соответствующей заданной. В файле результатов должен быть исходный словарь, буква для поиска и список найденных слов, каждое с новой строки и с порядковым номером.

Вариант 22. В начале исходного текста стоит заглавная буква русского алфавита для поиска с символом ':' (например, `A:`). Далее идет словарь русских слов (прописными буквами), разделенных запятыми. После последнего слова должна стоять точка. Длина текста — не более `NL` строк, длина строки — не более `NUMW` слов, длина слова — не более `NW` символов. Найти слова, в которые входит буква (прописная), соответствующая заданной, но не является ни первой, ни последней буквой слова. В файле результатов должен быть исходный словарь, буква для поиска и список найденных слов, каждое с новой строки и с порядковым номером.

Вариант 23. Назовем сложностью предложения сумму слов и знаков препинания. Составить программу, определяющую максимальную сложность предложений в тексте на русском языке и среднюю сложность по всем предложениям. Длина текста — не более `NL` строк, длина строки — не более `NUMW` слов, длина слова — не более `NW` символов. Текст должен заканчиваться символом '/'. В файле результатов должен быть исходный текст, значение максимальной сложности и предложение (одно любое), соответствующее этой сложности и значение средней сложности предложения.

Вариант 24. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?', '...'). В первой строке файла данных имеется следующая информация:

`<отступ абзаца> <левая граница> <правая граница>`

Со второй строки начинается исходный текст. Имеются следующие ограничения: отступ абзаца — от 0 до 5 символов, левая граница — от 1 до 10 символов и правая граница — от 40 до 80 символов. Составить программу, которая отформатирует исходный текст с заданными параметрами. В файле результатов должен быть исходный текст, строка формата и отформатированный текст.

Вариант 25. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой. Составить программу для проверки сбалансированности скобок в тексте. Скобки сбалансированы, если закрывающая скобка расположена после открывающей скобки одного вида и их количества совпадают. Возможные виды скобок: `()`, `{}`, `[]`. В файле результатов должен быть исходный текст и первый обнаруженный фрагмент, начиная с несбалансированной скобки и до конца текста.

Вариант 26. Задан исходный текст на русском языке. Длина текста — не более `NL` строк, длина строки — не более `NS` символов, длина слова — не более `NW` символов. Исходный текст должен заканчиваться точкой ('!', '?', '...'). Составить программу для нахождения буквы, с которой начинается больше всего слов в тексте (прописную и заглавную буквы считать одинаковыми). В файле результатов должен быть исходный текст и слова, начинающиеся с заданной буквы (с новой строки и с порядковым номером).

П2.7.2. Обработка массивов. Варианты заданий

Вариант 27. Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора.

Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, "[]", поиск максимального числа, встречающегося в заданном векторе более одного раза).

ПРИМЕЧАНИЕ

Для работы с матрицами на основе класса `vector` с использованием технологии обобщенного программирования можно использовать прием, рассмотренный в упражнении 15 в разд. 15.14.

Вариант 28. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (определение нормы заданной матрицы, т. е. значения $\max_i(\sum_j |a[i][j]|)$).

Вариант 29. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (по заданной квадратной матрице размером $N \times N$ построить вектор длиной $(2 \cdot N - 1)$, элементы которого — максимумы элементов диагоналей, параллельных главной, включая главную диагональ).

Вариант 30. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (характеристикой строки матрицы назовем сумму ее положительных элементов, имеющих четные значения индексов; переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик).

Вариант 31. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (для заданной квадратной матрицы найти минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали).

Вариант 32. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (говорят, что матрица имеет седловой элемент $a[i][j]$, если элемент $a[i][j]$ является минимальным в i -ой строке и максимальным в j -ом столбце; найти номера строки и столбца какого-либо седлового элемента и его значение).

Вариант 33. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (найти значение максимального элемента матрицы среди всех элементов тех строк матрицы, которые упорядочены либо по возрастанию, либо по убыванию значений элементов).

Вариант 34. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (характеристикой столбца матрицы назовем сумму его отрицательных элементов, имеющих нечетные значения индексов; переставляя столбцы заданной матрицы, расположить их в соответствии с убыванием характеристик).

Вариант 35. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (элемент матрицы называется локальным минимумом, если его значение строго меньше значений всех имеющихся соседей; подсчитать количество локальных минимумов заданной матрицы).

Вариант 36. Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл)

и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, "[]"; найти элемент вектора, имеющий максимальное значение; элементы, стоящие после максимального, заменить нулями и переставить в начало вектора).

Вариант 37. Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, "[]"; найти максимальное значение элемента вектора среди отрицательных и минимальное значение — среди положительных элементов).

Вариант 38. Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, "[]"; упорядочение элементов вектора по знаку, сначала положительных, а затем — отрицательных, в таком же порядке, как в исходном векторе).

Вариант 39. Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, "[]"; поиск максимального элемента вектора и, если он не равен нулю, то деление на него всех элементов вектора, если же максимальный элемент вектора равен нулю, то вектор не изменяется).

Вариант 40. Векторная арифметика. Элементы вектора могут быть любого типа с плавающей точкой. Реализовать перегрузку различных операций над векторами и некоторую обработку вектора. Для решения данной задачи использовать шаблон базовых классов (размещение вектора в динамической памяти и его инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение вектора значениями из файла, печать содержимого вектора в файл) и шаблон производных классов (перегрузить операции сложения, вычитания, умножения и деления векторов, "[]"; поиск элементов, встречающихся в векторе более одного раза, из найденных элементов сформировать новый вектор).

Вариант 41. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (упорядочение по возрастанию элементов каждой строки

матрицы, сортировка строк должна выполняться на месте, это означает, что вспомогательный вектор не должен использоваться).

Вариант 42. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (вычисление количества положительных элементов в левом нижнем треугольнике квадратной матрицы, треугольник включает диагональ матрицы).

Вариант 43. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (обмен местами максимального элемента главной диагонали квадратной матрицы и минимального элемента побочной диагонали).

Вариант 44. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (печать значений элементов той строки матрицы, сумма элементов которой минимальна).

Вариант 45. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (нахождение суммы значений элементов тех строк матрицы, у которых на главной диагонали расположены элементы, имеющие отрицательные значения).

Вариант 46. Матричная арифметика. Элементы матрицы могут быть любого типа с плавающей точкой. Для решения данной задачи использовать шаблон базовых классов (размещение матрицы в динамической памяти и ее инициализация — конструктор, при необходимости — конструктор копирования, деструктор, заполнение матрицы значениями из файла, печать содержимого матрицы в файл) и шаблон производных классов (перестановка строк матрицы по убыванию значения их первого элемента).

П2.7.3. Решение геометрических задач.

Варианты заданий

Общие указания. В следующих далее вариантах заданий координаты точек могут быть любого типа с плавающей точкой. Координаты точек следует хранить в динамической памяти. При вводе исходных данных из файла на магнитном диске нужно предусмотреть контроль достаточности количества исходных данных, кратности

прочитанных данных двум (точки на плоскости) или трем (точки в пространстве), наличия повторяющихся координат точек и т. п.

Для решения заданной задачи использовать шаблон базовых классов (считывание из файла и размещение координат точек в динамической памяти — конструктор, при необходимости — конструктор копирования, деструктор, печать координат точек в файл) и шаблон производных классов (содержательное решение задачи).

Вариант 47. Из заданного множества точек на плоскости выбрать две различные точки так, чтобы количества точек, лежащих по разные стороны прямой, проходящей через эти две точки, различались наименьшим образом.

Вариант 48. Определить радиус и центр окружности, на которой лежит наибольшее число точек заданного на плоскости множества точек.

Вариант 49. Определить радиус и центр такой окружности, проходящей хотя бы через три различные точки заданного множества точек на плоскости, что минимальна разность числа точек, лежащих внутри и вне окружности.

Вариант 50. Расстояние между двумя множествами точек — это расстояние между наиболее близко расположенными точками этих множеств. Найти расстояние между двумя заданными множествами точек на плоскости.

Вариант 51. Задано множество точек в трехмерном пространстве. Найти такую из них, что шар заданного радиуса с центром в этой точке включает в себе максимальное число точек множества.

Вариант 52. Выбрать три разные точки заданного на плоскости множества точек, составляющие треугольник наибольшего периметра.

Вариант 53. Определить радиус и центр окружности минимального радиуса, проходящей хотя бы через три различные точки множества точек на плоскости.

Дополнительные определения и указания для решения геометрических задач.

Множество точек на плоскости — это конечное, возможно пустое множество различных пар вещественных чисел, соответствующих координатам точек на плоскости. Аналогично определяется и множество точек трехмерного пространства.

1. Для построения прямой, проходящей через две различные точки P и Q с координатами (P_X, P_Y) и (Q_X, Q_Y) , достаточно в уравнении прямой $A * x + B * y + C = 0$ взять $A = P_Y - Q_Y$, $B = Q_X - P_X$, $C = (P_X - Q_X) * P_Y + (Q_Y - P_Y) * P_X$.
2. Прямая, проходящая через точку P перпендикулярно отрезку PQ , имеет уравнение $B * x + A * y + D = 0$ с приведенными в п. 1 значениями A , B и $D = B * P_X - A * P_Y$.
3. Подстановка в левую часть уравнения прямой $A * x + B * y + C = 0$ координат двух точек, лежащих по одну сторону от этой прямой, дает значение одного знака.
4. Расстояние между точками P и Q с координатами (P_X, P_Y) и (Q_X, Q_Y)

$$d = \sqrt{(Q_X - P_X)^2 + (Q_Y - P_Y)^2}.$$

5. Точка пересечения прямых $A_1 * x + B_1 * y + C_1 = 0$ и $A_2 * x + B_2 * y + C_2 = 0$ определяется координатами

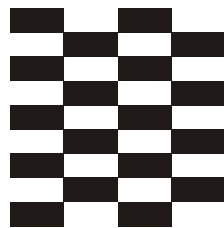
$$y_0 = \frac{B_1 * C_2 - B_2 * C_1}{A_1 * B_2 - A_2 * B_1}; \quad x_0 = \frac{C_1 * A_2 - C_2 * A_1}{A_1 * B_2 - A_2 * B_1}.$$

6. Уравнение окружности с радиусом R и центром окружности с координатами (x_0, y_0)

$$(x - x_0)^2 + (y - y_0)^2 = R^2.$$

7. Алгоритм получения центра окружности, проходящей через три заданные точки — определение точки пересечения перпендикуляров к серединам хорд.
8. Площадь треугольника со сторонами a, b, c .

$$s = \sqrt{p * (p - a) * (p - b) * (p - c)}, \quad p = (a + b + c) / 2.$$



Приложение 3

Создание и отладка программного проекта консольного приложения в Microsoft Visual Studio C++ .NET

Создание и отладка программного проекта консольного приложения в предыдущей версии интегрированной среды разработки (ИСР) Microsoft Visual Studio C++ 6.0 рассмотрены в [3] (приложения П.2 и П.4). Далее рассматривается методика создания и отладки программного проекта консольного приложения для последней версии ИСР — Microsoft Visual Studio C++ .NET.

П3.1. Создание программного проекта консольного приложения

Существует несколько вариантов создания программных проектов консольных приложений, которые мы и рассмотрим далее.

П3.1.1. Создание нового проекта для консольного приложения

Чтобы создать новое приложение (программу), необходимо создать новый проект. Для этого в ИСР выполните команду **File | New | Project** (Файл | Создать | Проект). Появится диалоговое окно **New Project** (новый проект), изображенное на рис. П3.1. Обратите внимание, что в окне иерархической структуры **Project Types** (тип проекта) уже выбран по умолчанию тип проекта **Visual C++ Projects**.

В текстовое поле **Name** (имя) введите новое имя проекта, например, **Project1**, а в текстовом поле **Location** (местоположение) укажите место расположения нового проекта (диск:\путь\подкаталог). Это можно сделать, набрав путь вручную или воспользовавшись расположенной справа кнопкой **Browse** (просмотр). Разумеется, что соответствующий подкаталог должен быть предварительно создан.

В окне списка **Template** (шаблоны) щелкните левой кнопкой мыши на значке **Win32 Project**. В результате появится окно **Win32 Application Wizard – Project1** (рис. П3.2).

Для задания свойств проекта выберите в этом окне вкладку **Application Settings**, тогда вид окна изменится (рис. П3.3). На рисунке показаны установки проекта, соответствующие созданию пустого консольного приложения. Указанное консольное приложение создается после нажатия в этом окне кнопки **Finish**. Вид окна ИСР после выполнения указанных действий показан на рис. П3.4.

Для того чтобы наполнить созданный проект, необходимо добавить в него файл(ы), содержащий(ие) текст программы.

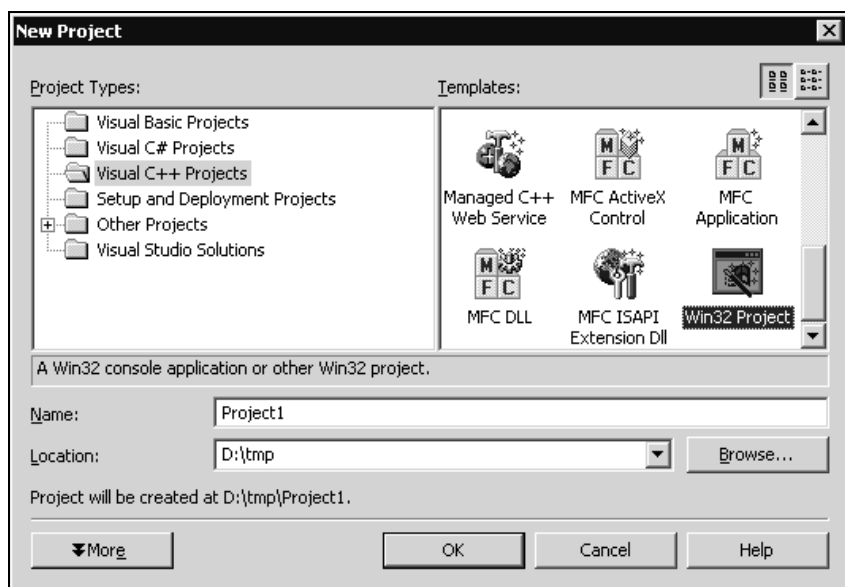


Рис. ПЗ.1. Вид диалогового окна **New Project** при создании нового проекта консольного приложения



Рис. ПЗ.2. Диалоговое окно конфигурирования приложения

Это можно сделать двумя способами:

- ☐ создать новый файл(ы) и включить его(их) в проект;
- ☐ добавить в проект уже существующий файл(ы), созданный(ые) ранее в текстовом редакторе и имеющий(ие) расширение `cpp`. Повторно обращаем ваше внимание на то, что следует предварительно поместить существующий(ие) файл(ы) в каталог проекта (лучше все иметь в одном месте).

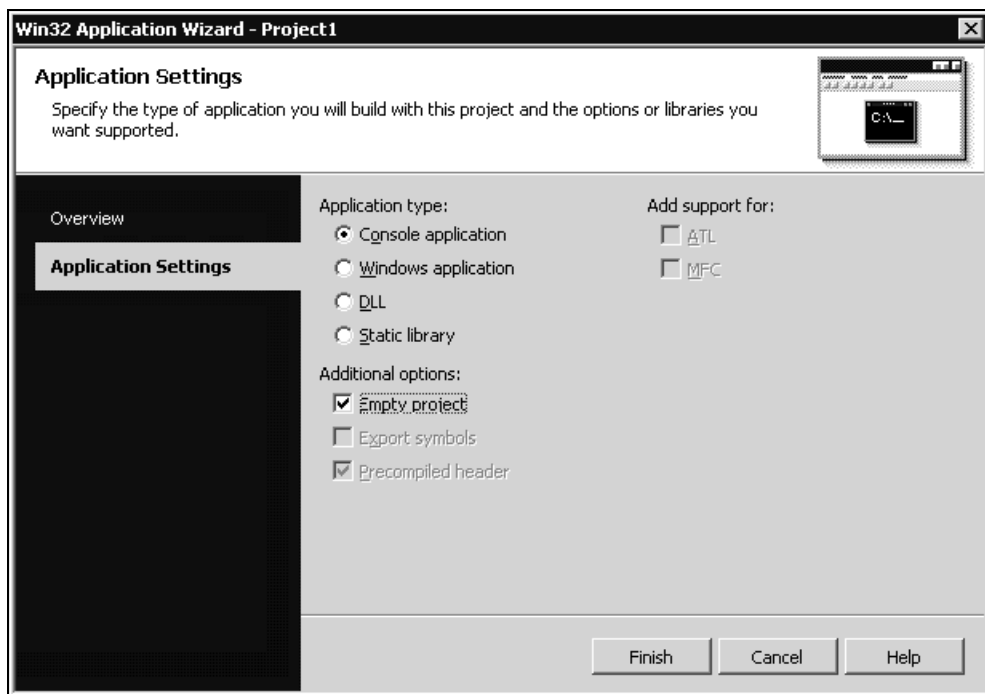


Рис. ПЗ.3. Конфигурирование консольного приложения

ПЗ.1.2. Создание нового файла и включение его в проект

Для создания нового файла выберите в окне **Solution Explorer – Project1** проект **Project1** и для него правой кнопкой мыши вызовите контекстное меню (рис. ПЗ.5).

В контекстном меню выполните команду **Add** (добавить), **Add New Item** (добавить новый элемент), в появившемся окне **Add New Item – Project1** в поле **Categories** выберите категорию файла **Visual C++, C++**, в поле **Templates** выберите вид файла (**C++ File (.cpp)**, или **Header File (.h)**, или др.), в поле **Name** укажите имя файла и нажмите кнопку **Cancel** (рис. ПЗ.6).

В появившемся в ИСР окне редактирования **Project1.cpp** введите исходный текст файла и с помощью команды **File | Save** сохраните его на магнитном диске (рис. ПЗ.7). Таким же образом можно создать и включить в проект все необходимые файлы.

П3.1.3. Добавление в проект существующего файла

Для этого выберите в окне **Solution Explorer – Project1** проект **Project1** и для него правой кнопкой мыши вызовите контекстное меню (см. рис. П3.5). В контекстном меню выполните команду **Add** (добавить), **Add Existing Item** (добавить существующий элемент), в появившемся окне **Add Existing Item – Project1** (рис. П3.8) выберите файл для включения в проект и нажмите кнопку **Open**. Таким же образом можно включить в проект все необходимые существующие файлы.

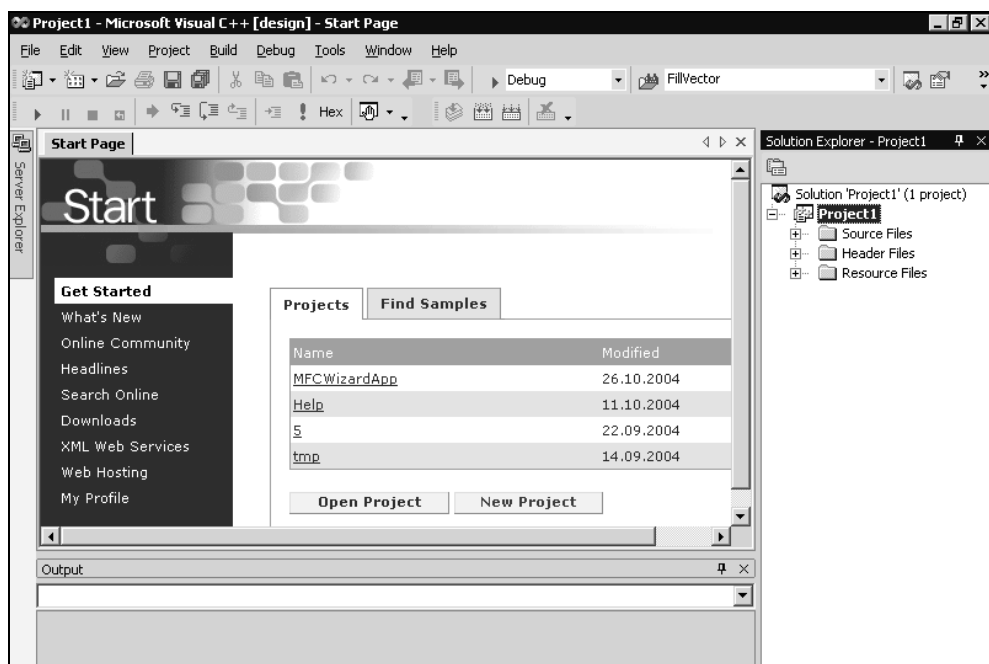


Рис. П3.4. Вид окна ИСР после конфигурирования приложения

После включения в программный проект всех необходимых файлов можно переходить к следующему этапу — отладке программного проекта.

П3.1.4. Открытие для работы существующего проекта

Вид окна ИСР после запуска показан на рис. П3.9.

Для открытия существующего проекта выполните команду **File | Open Project** и в появившемся диалоговом окне **Open Project** (рис. П3.10) войдите в каталог проекта и щелкните по файлу с расширением **sln** (SoLutioN). В результате проект загружается в ИСР для последующей работы.

П3.2. Отладка программного проекта консольного приложения

Известное высказывание о том, что после обнаружения последней ошибки в программе остается еще хотя бы одна, стало аксиомой. Все ошибки можно разделить на три большие категории.

- ❑ **Синтаксические ошибки**, которые выявляются на этапе компиляции. В зависимости от языка программирования, компилятор лучше или хуже выявляет такие ошибки. В ряде случаев синтаксическая ошибка в программе влечет за собой неадекватную реакцию компилятора. Например, отсутствие скобки часто приводит к тому, что компилятор обнаруживает ошибку через десятки строк кода. В последнем случае можно рекомендовать одновременный набор открывающей и закрывающей скобок (например, `{ }`) с последующим вводом текста между ними.
- ❑ **Логические** (часто их также называют *алгоритмическими*) **ошибки**. Их бывает наиболее трудно обнаружить и исправить. Часть из них выявляется на этапе отладки, часть на этапе сопровождения, а некоторые приводят к тяжелым последствиям.
- ❑ **Информационные ошибки**. В частности, к появлению информационных ошибок может привести отсутствие обработки ошибок ввода-вывода, попытки деления на ноль, переполнение разрядной сетки компьютера и т. п. Для исключения и/или обработки информационных ошибок в ряде случаев приходится большую часть кода функций отводить для всевозможных проверок.

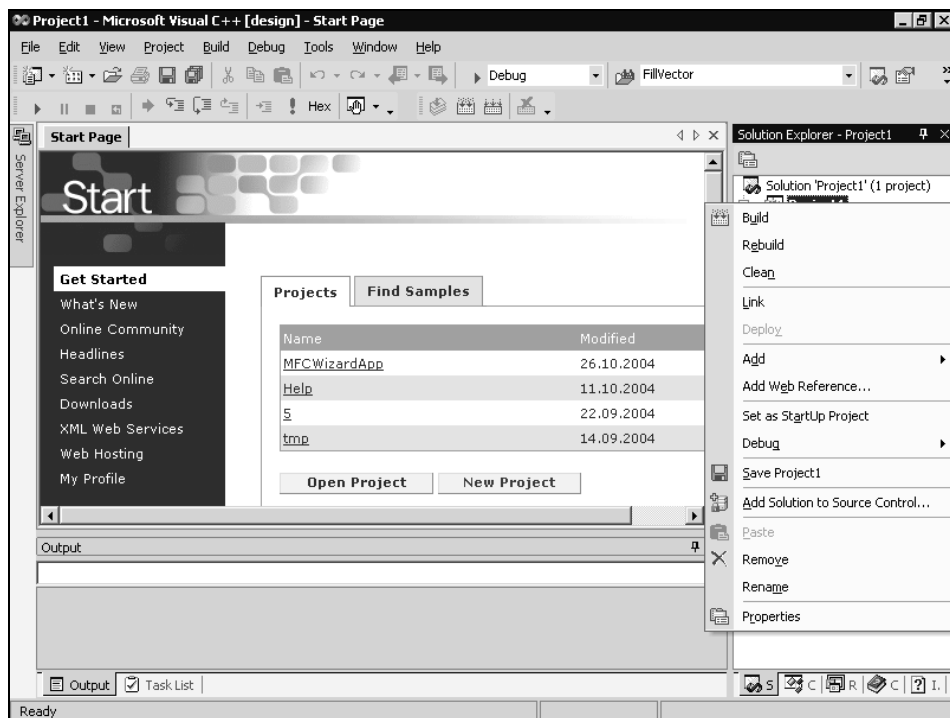


Рис. П3.5. Контекстное меню проекта

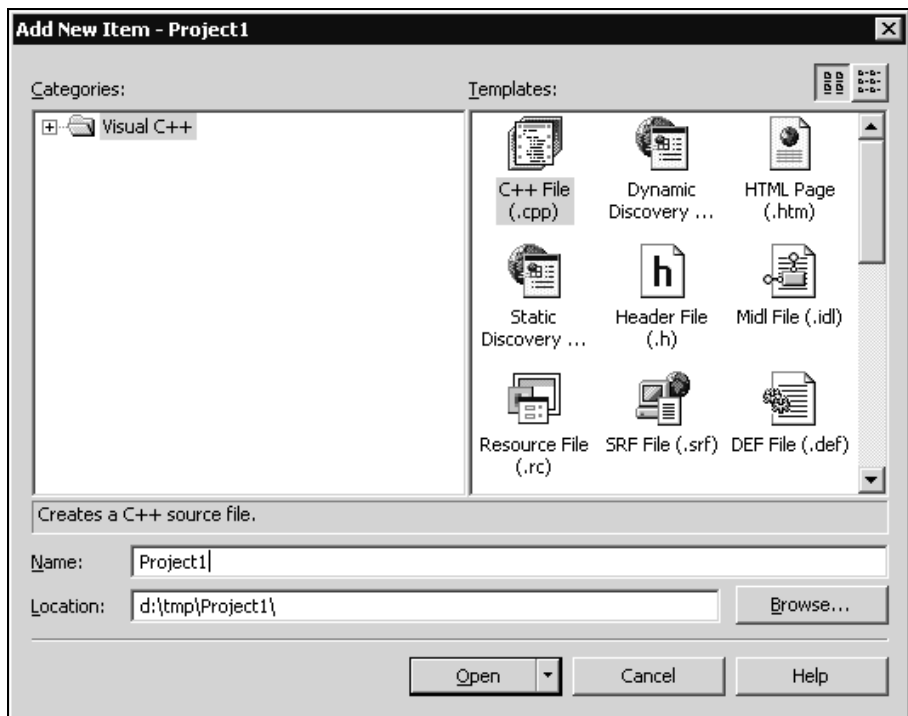


Рис. ПЗ.6. Добавление в проект нового файла

ПЗ.2.1. Компиляция и компоновка программного проекта. Устранение синтаксических ошибок

На этапе отладки мы должны устранить все синтаксические и большую часть логических ошибок в программном проекте, допущенных на предыдущих этапах создания приложения. Для этого необходимо скомпилировать каждый созданный файл. Это можно сделать несколькими способами.

- ❑ Скомпилировать каждый файл с расширением `spp`.

Для этой цели можно использовать команду **Build | Compile** или комбинацию клавиш `<Ctrl>+<F7>`. Такой способ удобно использовать в больших проектах, чтобы сосредоточиться на конкретном файле. При этом следует иметь в виду, что ссылки между файлами не проверяются.

- ❑ Скомпилировать и компоновать все файлы проекта ("собрать" или "построить" исполняемый файл).

Для этой цели можно использовать команду **Build | Build имя_проекта** или **Build | Rebuild имя_проекта**. Единственным отличием этих команд является то, что команда **Rebuild имя_проекта** не проверяет даты создания файлов проекта и компилирует все файлы, а не только те, которые были модифицированы после компиляции. В результате создается исполняемый файл с расширением `exe`.

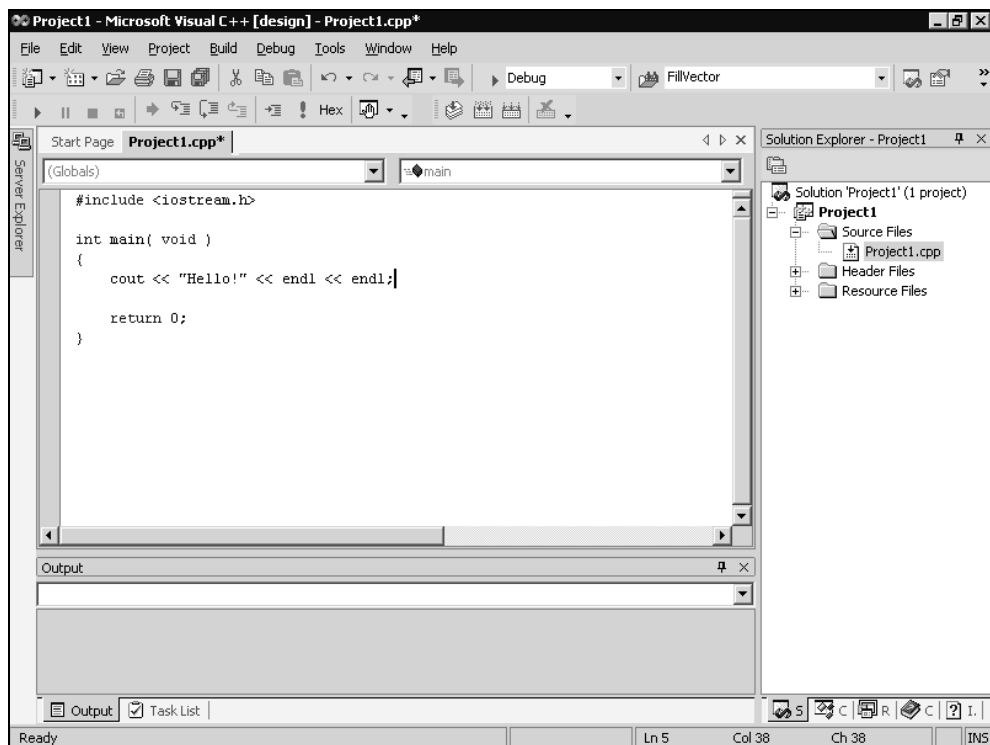


Рис. ПЗ.7. Ввод текста файла, включаемого в проект

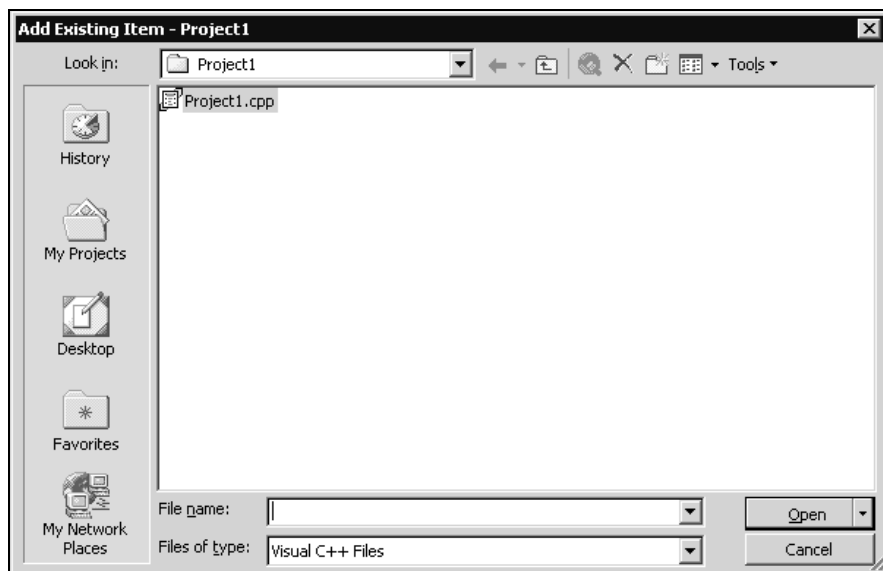


Рис. ПЗ.8. Включение в проект существующего файла

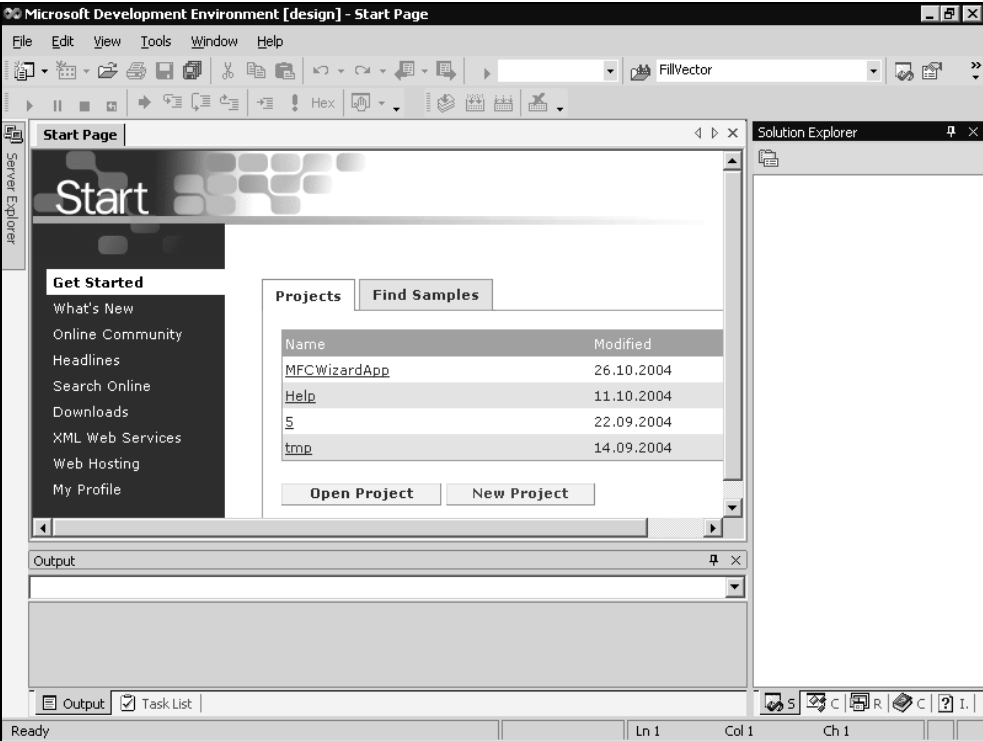


Рис. ПЗ.9. Вид окна ИСР после запуска

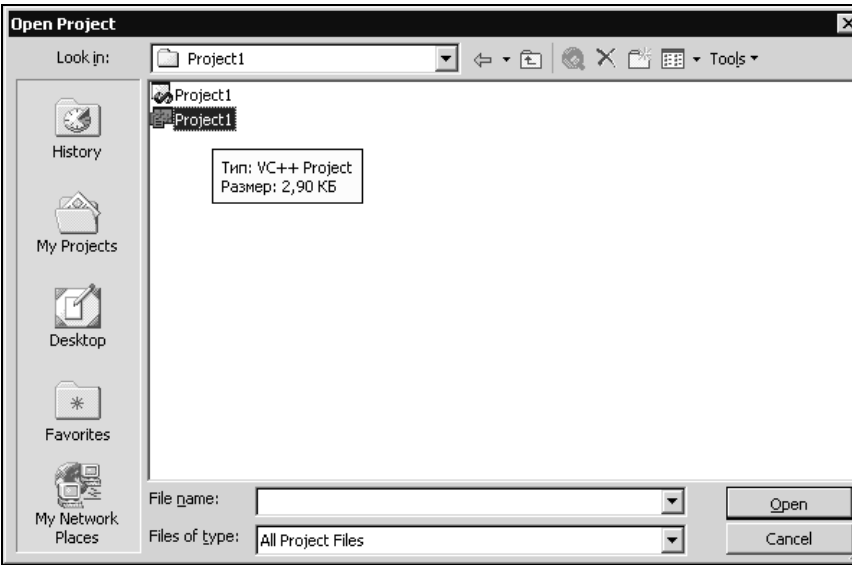


Рис. ПЗ.10. Открытие для работы существующего проекта

- ☐ Сразу запустить приложение.

Для этой цели можно использовать команду **Debug | Start Without Debugging** или запустить приложение по комбинации клавиш **<Ctrl>+<F5>**. Если в программный проект были внесены какие-либо изменения, то на экране будет высвечено диалоговое окно с запросом на построение исполняемого файла. Для построения указанного файла следует нажать кнопку **[Да]**.

Если программный проект содержит синтаксические ошибки, то при выполнении любой из представленных команд информация об ошибках автоматически отображается во вкладке **Build** окна **Output**, по умолчанию расположенного в нижней части окна ИСР. Если окно **Output** было закрыто, то его можно вывести снова на экран с помощью команды **View | OtherWindow | Output** или по комбинации клавиш **<Ctrl>+<Alt>+<O>**. Каждое сообщение об ошибке или предупреждении начинается с имени файла, где они обнаружены, за которым следует номер строки, где это произошло, а далее идут двоеточие и слово "error" (ошибка) или "warning" (предупреждение) и соответствующий номер. В конце приводится краткое описание ошибки или предупреждения. Если дважды щелкнуть левой кнопкой мыши на строке с сообщением или предупреждением, то ошибочная строка будет отмечена стрелкой в левой части в соответствующем окне редактирования. Лучше всего добиться, чтобы в окончательном варианте не было ни того, ни другого, хотя с предупреждениями исполняемый файл создается и может быть запущен.

После устранения синтаксических ошибок следует запустить программу указанными ранее способами. При этом также можно получить сообщение об ошибке (или ошибках). Это тот самый случай, когда программный проект не содержит синтаксических ошибок, а приложение не работает. Вызвано это так называемыми *логическими (алгоритмическими) ошибками*, для обнаружения которых можно использовать разные методы (например, закомментировать фрагменты программы).

Однако лучше всего воспользоваться имеющимся в ИСР *встроенным отладчиком*.

П3.2.2. Отладка программного проекта.

Устранение логических (алгоритмических) ошибок

Встроенный отладчик предоставляет следующие возможности:

- ☐ запуск программы до заданного места;
- ☐ пошаговое выполнение программы;
- ☐ просмотр значений переменных в любом месте программы.

Для пошагового выполнения программы отладчик предоставляет несколько возможностей, основные из которых мы и рассмотрим.

Для более удобной работы с отладчиком рекомендуем разместить в верхней части окна интегрированной среды программирования панель инструментов для отладки и настроить ее. Для этой цели достаточно щелкнуть правой кнопкой мыши по свободной области окна рядом с любой из панелей и в контекстном меню выбрать панель **Debug**. Конечно же, это следует сделать только в том случае, если панель **Debug** была скрыта. Для настройки этой панели в нее следует добавить кнопки **Start Without Debugging** (запуск без отладки) и **Run To Cursor** (запуск до курсора). С этой целью достаточно выполнить команду **Tools | Customize**, в появившемся окне **Customize**

выбрать вкладку **Commands** и в ней выбрать категорию **Debug** (рис. ПЗ.11). Для добавления кнопки **Start Without Debugging** достаточно в списке **Commands** выбрать эту команду и с помощью левой кнопки мыши перетащить ее в нужное место панели инструментов **Debug**. Аналогичным образом на панель **Debug** добавляется кнопка **Run To Cursor**. После этого окно **Customize** можно закрыть и окно ИСР с панелью инструментов **Debug** приобретает вид, показанный на рис. ПЗ.12.

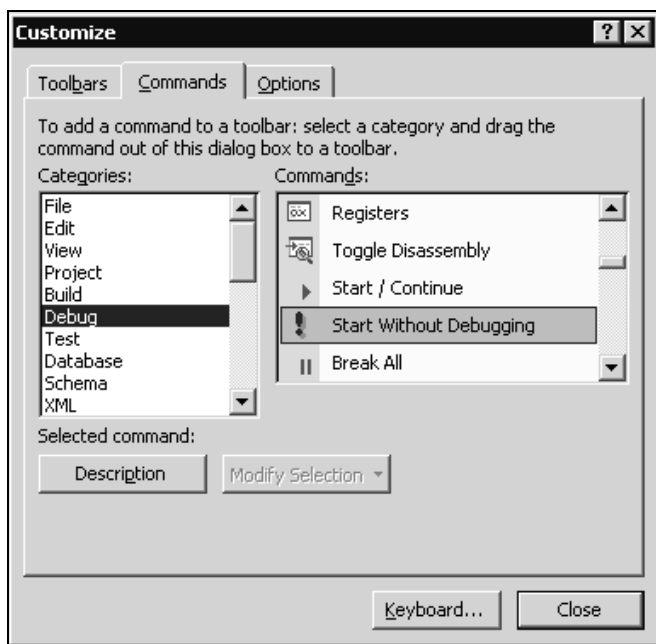


Рис. ПЗ.11. Выбор кнопки **Start Without Debugging** для добавления ее на панель **Debug**

Для запуска исполняемого файла в режиме отладки можно нажать кнопку **Start** на панели инструментов **Debug** или нажать клавишу <F5>. Однако если просто выполнить эту команду, не предпринимая никаких предварительных действий, работа программы не будет отличаться от запуска в обычном режиме, разве что при завершении на вкладке **Debug** в нижней части окна **Output** ИСР появится информация о параметрах завершения работы программы.

Чтобы перейти в режим пошагового выполнения, предварительно перед выполнением команды **Start** необходимо установить так называемые *точки останова* (break-points), которые можно рассматривать как стоп-сигналы для отладчика. Обычно они устанавливаются в местах, которые вызывают сомнение в правильности выполнения. При этом предполагается, что все операторы, предшествующие первой точке останова, выполняются правильно. Самый простой способ установки точки останова заключается в следующем. Курсор устанавливается на строку, на которой нужно остановить работу программы, и нажимается клавиша <F9>. Повторное нажатие клавиши <F9> удаляет точку останова. Строка останова в окне редактирования отмечена темно-красным кружком в крайней левой позиции. Если, после задания

точки останова, программу запустить с помощью кнопки **Start**, либо нажав клавишу <F5>, то все операторы программы, предшествующие точке останова, будут выполняться в обычном режиме и только перед строкой останова выполнение программы приостановится.

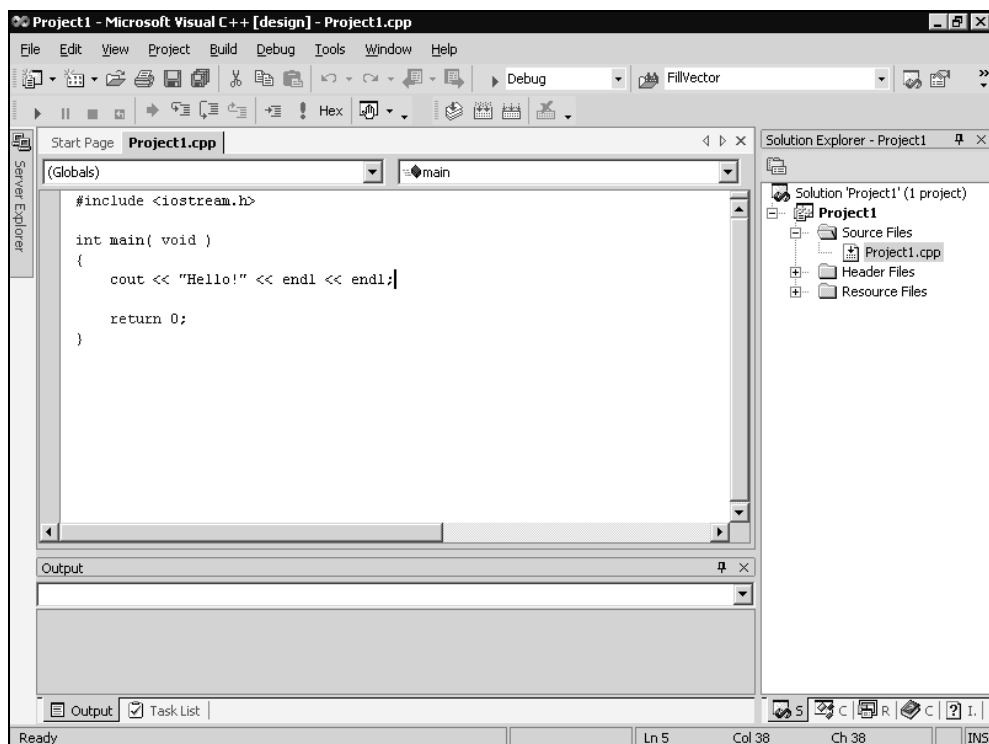


Рис. ПЗ.12. Вид окна ИСР с модифицированной панелью **Debug**

При этом вид ИСР существенно изменится. Во-первых, изменится состав команд некоторых меню и, в частности, меню **Debug**. Во-вторых, строка, которая будет выполняться следующей, будет отмечена желтой (по умолчанию) стрелкой. И, наконец, в нижней части экрана появится новое окно с вкладками **Autos**, **Locals** и **Watch1**, которые позволяют просматривать и менять значения переменных.

Для пошагового выполнения в отладчике имеются следующие команды:

- ☐ **Debug | Step Over** (шаг поверх) или <F10>, или кнопка **Step Over** на панели **Debug** — выполняет текущий оператор или функцию и переходит к следующей строке;
- ☐ **Debug | Step Into** (шаг внутрь) или <F11>, или кнопка **Step Into** на панели **Debug** — выполняет текущий оператор языка C++ или переходит к первому оператору функции;
- ☐ **Debug | Step Out** (шаг вне) или <Shift>+<F11>, или кнопка **Step Out** на панели **Debug** — завершает выполнение текущей функции и переходит к строке, непосредственно следующей за ее вызовом;

❑ кнопка **Run to Cursor** (выполнить до курсора) на панели **Debug** выполняет программу до строки, где в текущий момент находится курсор.

В окне **Locals** (переменные) автоматически отображаются только локальные переменные текущего блока. После каждого шага выполнения программы значения этих переменных обновляются.

Переменные, которые нужно контролировать или изменять по желанию программиста, можно задать в окне **Watch1** (наблюдение). Для этого в свободной строке столбца **Name** для контроля значения переменной достаточно набрать идентификатор переменной и нажать клавишу <Enter>. Для изменения значения переменной в процессе отладки следует выбрать строку с именем этой переменной, с помощью клавиши <Tab> перейти в столбец **Value**, набрать там новое значение и нажать клавишу <Enter>. При дальнейшей отладке, вместо прежнего значения, будет использовано модифицированное таким образом значение переменной.

Для просмотра значения переменной *в режиме отладки*, наряду с использованием окон **Locals** и **Watch1**, можно в окне редактирования поставить курсор на имя интересующей нас переменной. Если переменной было присвоено значение, то появится всплывающее окно со значением этой переменной. Эта возможность *наиболее удобна*, и мы рекомендуем ее использовать как можно чаще.

П3.2.3. Тестирование программного проекта

Как и любой другой продукт производства, программа перед использованием должна быть тщательно проверена. Этот этап является едва ли не самым сложным во всем процессе создания программы — необходимо учесть все варианты ее поведения. Поэтому его нужно начинать не после завершения отладки, а одновременно с разработкой алгоритма.

Одним из путей проверки или тестирования программы является ее выполнение по одному разу с каждой из возможных комбинаций входных данных — т. е. тестирование с использованием заранее подготовленного набора контрольных примеров.

Требования к контрольным примерам:

- ❑ набор контрольных примеров должен быть достаточным, чтобы показать выполнение всех требований технического задания и обеспечить полную проверку программного проекта — протестировать все ветви, имеющиеся в программе;
- ❑ контрольные примеры должны быть простыми в том смысле, чтобы анализ ожидаемых результатов был несложным (примеры должны быть небольшой размерности со значениями исходных данных, удобными для анализа).

Еще раз отметим, что создание достаточного и простого набора контрольных примеров является нетривиальной задачей.

Структура контрольного примера может быть следующей:

- ❑ цель примера;
- ❑ исходные данные (для примеров с нормальным завершением) или как моделировать ошибку (для примеров с аварийным завершением);
- ❑ анализ ожидаемых результатов (для примеров с нормальным завершением — анализ ожидаемых результатов в точках останова).

Правила выбора точек останова:

- ☐ точки останова следует выбирать после выполнения каждой функции программного проекта (в них следует проверить результаты работы функции);
- ☐ если декомпозиция задачи выполнена не очень удачно и функция получилась большой (более страницы текста), то следует в ее теле выбрать промежуточные точки останова, разбив тело функции на алгоритмически законченные части;
- ☐ если функция была отлажена ранее, то после нее точку останова выбирать не следует;
- ☐ если функция результаты своей работы выводит в файл на магнитном диске, на экран или на принтер, то после такой функции точки останова тоже не нужны.

Методика тестирования программы для контрольных примеров с нормальным завершением.

- ☐ Программа запускается до первой точки останова так, как это указывалось ранее. Если полученные машинные результаты совпадают с результатами анализа, приведенного в контрольном примере, то аналогично программа запускается до следующей точки останова и т. д.
- ☐ Если в очередной точке останова машинные результаты отличаются от ожидаемых, то текущий сеанс отладки завершается с помощью команды **Debug | Stop Debugging**. Программа повторно запускается до последней точки останова с хорошими результатами и с этого места выполняется по шагам с проверкой полученных результатов (выполняется построчная трассировка ошибочного участка). По результатам пошаговой проверки обнаруживается ошибка и текущий сеанс отладки также прекращается. Затем в исходный текст программы вносятся необходимые исправления и трассировка ошибочного участка повторяется. Этот процесс заканчивается после исправления ошибок, о чем будет свидетельствовать получение в очередной точке останова ожидаемых результатов.

ПЗ.3. Русификация консольных приложений

В консольных приложениях (DOS-приложениях) при выводе текста на экран используется кодировка OEM, которая для русских букв отличается от кодировки ANSI, используемой в операционных системах типа Windows. Поэтому если выводимый в программе русский текст набран в кодировке ANSI, то его экранный вывод становится нечитаемым.

Для разрешения указанной коллизии можно использовать три несложных приема.

- ☐ Экранный вывод выполнять только на английском языке.
- ☐ Переназначение экранного вывода в файл на магнитном диске (при просмотре содержимого полученного файла в среде Windows русский текст в файле становится читаемым). Подобный прием рассмотрен ранее в *разд. 5.4* (см. файл MANIP2.CPP).
- ☐ Перед экранным выводом выполнить преобразование выводимого на экран текста из кодировки ANSI в кодировку OEM. Сущность этого приема иллюстрируют листинги ПЗ.1, ПЗ.2.

Листинг ПЗ.1. Файл RusCOut.h

```

/*
    Русификация консольного вывода.
    В. Давыдов, Т. Сидорина, консольное приложение, Microsoft Visual Studio C++ .NET
*/

// Предотвращение возможности многократного подключения данного файла
#ifndef __RusCOut_h
#define __RusCOut_h

    #include <iostream> // Поточковый ввод-вывод
    #include <iomanip>   // Манипуляторы с параметрами
    #include <sstream>   // Строковые потоки
    #include <string>    // Строки

    using namespace     // Используем стандартное
        std;           // пространство имен

    #include <windows.h> // Для CharToOem( )

    // Макроопределение с параметром - выводит текст из параметра
    // TextForOut на экран с предварительным преобразованием
    // в DOS-кодировку (OEM)
    #define RusCOut( TextForOut ) \
    { \
        ostringstream os; \
        TextForOut; \
        string s = os.str( ); \
        char *pText = new char [ s.size( )+1 ]; \
        if( !pText ) \
        { \
            cout << "\n RusCOut: error 10 DM" << endl; \
            exit( 10 ); \
        } \
        ::CharToOem( s.c_str( ), pText ); \
        cout << pText << endl; \
        delete pText; \
    }

    // Пояснения к макроопределению с параметром: os - объект выходного
    // строкового потока; TextForOut - параметр с текстом, подлежащим
    // выводу на экран; s - строка с текстом, подлежащим выводу на
    // экран; pText - указатель на область ДП с текстом, подлежащим
    // выводу на экран (тип указателя char *); CharToOem( ) - функция
    // преобразования из ANSI в OEM (s.c_str( )->pText)

#endif

```

Листинг П3.2. Файл RusCOut.cpp

```
/*
    Русификация консольного вывода с использованием включаемого файла RusCOut.h.
    В. Давыдов, Т. Сидорина, консольное приложение, Microsoft Visual Studio C++
    .NET)
*/

#include "RusCOut.h"      // Русификация экранного вывода

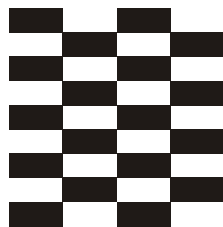
// Так рекомендует оформлять заголовок главной функции стандарт языка
// C++, если командная строка не используется
int main( )
{
    int          i = 12;
    double       d = 21.21;

    // Экранный вывод для консольного приложения с перекодировкой
    // русского текста в OEM. Вместо традиционного вывода
    // cout << endl << "Русский текст" << endl << "i = " << setw( 10 )
    // << i << setprecision( 3 ) << ", d = " << d << endl;
    // используйте следующий макровывод:
    RusCOut( os << endl << "Русский текст" << endl << "i = "
            << setw( 10 ) << i << setprecision( 3 ) << ", d = " << d
            << endl );

    // При использовании данного макровывода следует соблюдать следующие
    // правила:
    // 1. Идентификатор объекта экранного вывода (os) является
    // фиксированным.
    // 2. Правила потокового вывода языка C++ в части манипуляторов и
    // флагов форматирования полностью применимы и в данном случае

    // Так рекомендует заканчивать главную функцию стандарт языка C++ при
    // ее успешном завершении. Указанное умолчание эквивалентно
    // использованию оператора
    // return 0;
}
```

Приложение 4



Прилагаемый компакт-диск

На прилагаемом компакт-диске, в первую очередь, содержатся исходные тексты всех примеров программ, имеющих в пособии. Это очень удобно для экспериментов с демонстрационными программами. Программные проекты демонстрационных примеров включают также и исполняемые файлы (exe), так что вам не надо обязательно компилировать и компоновать заинтересовавшие вас примеры. Все программные проекты примеров "самодостаточны". Это означает, что ни одному из них не требуются файлы других проектов.

Кроме исходных текстов демонстрационных программ автора на компакт-диске имеются:

- ❑ файл с полным текстом *приложения 2* в формате текстового редактора Word 2000 (содержит многочисленные варианты тестов по основным разделам учебного пособия, варианты заданий на курсовое проектирование и материал, посвященный экзаменационному тестированию с использованием тестовых вопросов);
- ❑ файл с текстом *приложений 5—7* в формате текстового редактора Word 2000 (содержит перечень заголовочных файлов стандартной библиотеки языка C++ и сведения об их назначении, перечень констант, макросов, типов данных и функций стандартной библиотеки);
- ❑ ссылки на размещенное в Интернете описание стандарта языка C++ на английском языке (файл CppStd.foc).

Более полные сведения о содержимом компакт-диска имеется в файле ReadMe.doc, расположенном в корневой папке компакт-диска.

Литература

1. Аммерааль Л. STL для программистов на C++. — М.: ДМК, 2000.
2. Бруно Бабэ. Просто и ясно о Borland C++. Пер. с англ. — М.: БИНОМ, 1994.
3. Давыдов В. Г. Программирование и основы алгоритмизации. Учеб. пособие / Давыдов В. Г. — М.: Высш. шк., 2003.
4. Либерти, Джесс. Освой самостоятельно C++ за 21 день: 3-е изд., пер. с англ.: Уч. пос. — М.: Издательский дом "Вильямс", 2001.
5. Рассохин Д. От Си к C++. — М.: Издательство "ЭДЕЛЬ", 1993.
6. Страуструп Б. Язык программирования C++, спец. изд. / Пер. с англ. — М.: СПб.: "Издательство БИНОМ" — "Невский Диалект", 2002 г.
7. Тихомиров Ю. Visual C++ 6. — СПб.: БХВ—Петербург, 1999.
8. От Си к C++ / Е. И. Козелл, Л. М. Романовская, Т. В. Русс и др. — М.: Финансы и статистика, 1993.
9. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. — СПб.: Питер, 2001.
10. C++ Стандартная библиотека. Для профессионалов / Н. Джосьютис. — СПб.: Питер, 2004.

Предметный указатель

С

С-строка 316

А

Адаптер:

- метода класса 449

- указателя на функцию 447

- функции 444

Алгоритм 114

- множества и пирамиды 486

- модифицирующие операции 459

- немодифицирующие операции 452

Ассоциативный контейнер

- множество 422

- множество с дубликатами 427

- пара 406

- словарь 411

- словарь с дубликатами 420

Б

Бинарное дерево 263

Битовое множество 407

Битовое поле 376

Буфер 115

В

Вектор:

- коллекция упорядоченная 360

Время жизни 103

- блок 103

- группа файлов 104

- класс 104

- файл 104

- функция 104

Вывод:

- включение в поток 115

Д

Дек 393

Дерево:

- бинарное 315 499

- двоичное 486

- корень 263

- строго бинарное 264

Деструктор 15, 81, 93, 97, 138, 162,

165, 204, 205, 225, 226

Доступ:

- к элементам классов 114

- к членам классов 63, 66

- произвольный 360

Е

Емкость 361, 383

З

Заголовочный файл 112, 299

- стандартный 113

И

Иерархия 180, 181, 183, 197, 224

Именованная область 105

Инкапсуляция 4

Исключение 159, 160, 161, 162, 172
 выражение возбуждения 161
 явное 160

Исключительная ситуация 159
 обработка 159, 160, 161

Итератор 114, 355, 439
 адаптер 441
 вставки 442
 входного потока 442
 выходного потока 443
 двунаправленный 440
 обратный 441
 параметр алгоритма 451
 поточковый 442
 произвольного доступа 440

К

Класс 4, 58
 абстрактный 97
 базовый 4, 58, 163, 181, 183, 186,
 193, 204
 базовый виртуальный 76
 диагностический 115
 друг 22
 иерархия 4, 203, 298
 контейнерный 114, 353
 математический 115
 поточковый 114
 производный 58, 163, 181, 183, 186, 193
 стандартный 114
 строковый 114
 структура 299
 файловая структура 299
 функциональный 411
 шаблон 35, 203, 267, 276, 299,
 301, 353

Класс string
 деструктор 320
 добавление частей строк 322
 конструкторы 318
 операции 320
 поиск подстрок 330
 преобразования строк 324
 присваивание частей строк 322
 сравнение частей строк 339
 характеристики строк 343

Коллекция 353

Конструктор 15, 16, 17, 18, 42, 81, 93,
 97, 159, 165, 204, 205, 225, 226
 без параметров 136
 глубинное копирование 48
 инициализация 18
 копирования 42, 43
 поверхностное копирование 43
 с параметрами 136
 шаблона классов 286
Контейнер 353, 451

 адаптер 392
 ассоциативный 354
 вектор логических значений 376
 деструктор 356
 конструктор 356
 конструктор копирования 356
 конструктор умолчания 356
 массив 353
 общие методы 355
 обычный конструктор 356
 операции сравнения 356
 операция присваивания 356
 очередь 353
 последовательный 353 392
 список 353
 стек 353
 унифицированные типы 355
 характеристики 356
Контролируемый блок 160, 172
Контроль типов 160
Куча 486

М

Макросы 314, 498
Манипулятор 116, 124, 128, 136, 347
 параметризованный 124
 простой 124
Массив 266, 276
 динамический 315, 499
 манипуляции с индексами 286
 сложная сортировка выбором 264
 сортировка 208
Метод:
 абстрактный 96
 виртуальный 84, 85, 92, 93, 96, 183, 186
 перегруженный 117
 форматирования 124, 347

Методы-манипуляторы 124
Механизм виртуального вызова 92, 93
Множество 406
Множество с дубликатами 407
Мультиписки 263

Н

Наследование 4, 16, 58
 виртуальное 76

О

Область видимости 85, 92
Область действия 103
 блок 103
 группа файлов 104
 класс 104
 пространство имен 104
 прототип функции 104
 файл 104
 функция 104
Обработка исключений 81
Объединение 82
Объектно-ориентированное
 программирование 4, 202
Отрицатель 445
Очередь 264
 неограниченная на базе списка 265
 ограниченная на базе массива 265
 универсальная 276
 универсальная динамическая 262
 универсальная неограниченного
 размера 266
 универсальная ограниченного
 размера 276, 286
 FIFO 265
 LIFO 265
Ошибка 159

П

Пирамида 486
Поддерево
 левое 263
 правое 263
Поиск в таблице 241
 иерархия классов 241

 структура класса 242
 файловая структура 245
 шаблон классов 241
Полиморфизм 4 84
Последовательный контейнер
 очередь 398
 очередь с приоритетами 403
 стек 392
 вектор 358
 дек 378
 список 383
Поток 115
 входной 115
 выходной 115
 двунаправленный 115
 стандартный 115
 строковый 115
 файловый 115, 136
Поток-объект
 ошибка 118
Предикат 443, 444
Преобразование
 перекрестное 183
 повышающее 183
 понижающее 183
Преобразование типов 178, 179
Приведение типов 92, 178, 179,
 186, 190
Прикладное программирование
 классические задачи 202
Пространство имен 104, 105, 106, 107,
 112, 113
 глобальное 113, 115, 315, 499
 стандартное 113, 115
 оператор using 107

Р

Распределение 296

С

Связывание
 динамическое 84, 85
 статическое 84
Связыватель 445
Серия 298
Сигнатура 84, 85

Слияние 297
 трехфайловое 296
 упорядоченных последовательностей 296
Словарь 406
Словарь с дубликатами 406
Сортировка
 автоматическая 354
 массивов 203, 296
 методы 296
 файла 301
 файлов 296, 298
 файлов естественным слиянием 297
 файлов простым слиянием 296
Специальный контейнер
 битовое множество 432
Спецификатор доступа 7, 66, 82, 205
Список:
 двунаправленный 262
 линейный 261
 линейный двунаправленный 266
 линейный двунаправленный некольцевой 267
 обобщенный связанный 261
 однонаправленный 267
 однонаправленный некольцевой 261
 связанный 264, 315, 499
 специализированный линейный на базе массива 261
 циклический 261
Стандартная библиотека 113, 314, 498
 алгоритмы 314, 375, 451, 498
 алгоритмы сортировки 476
 диагностические классы 314, 498
 итераторы 314, 498
 класс string 316
 классы контейнерные 314, 498
 коллекции 315, 499
 комплексные числа 496
 математические классы 314, 498
 объявления алгоритмов 451
 поточковые классы 314, 498
 распределители памяти 493
 средства 493
 средства диагностики 494
 средства локализации 496
 средства поддержки языка 494
 строковые потоки 347
 строковый класс 314, 498

 численные алгоритмы 493
 шаблоны 315, 499
Стек 264, 265, 289
 вызовов 160
 динамический 261
 динамический неограниченного размера 289
 ограниченного размера на базе массива 289
 раскручивание 160, 162, 165
Строка 316
 Строковый поток 347
Структура 82

Т

Таблица виртуальных методов 186
Таблица виртуальных функций 93
 скрытый член-указатель 84
Транспортная задача:
 класс базовый 225
класс производный 225, 227
 коммивояжера 224
 решение 230
 структура классов иерархии 225
 файловая структура классов 229

У

Указатель 163, 261, 276

Ф

Фаза:
 распределения 296
 слияния 296
Флаг 347
Флаг форматирования 124
Функциональный объект 443
Функция 160, 161, 162
 виртуальная 5, 84

Ч

Чтение:
 извлечение из потока 11