

*УЧЕБНОЕ
ПОСОБИЕ*

 ПИТЕР®

Т. А. Павловская Ю. А. Щупак



C/C++

Структурное
и объектно-ориентированное
программирование

ПРАКТИКУМ

ББК 32.973.2-018.1
УДК 004.43(075)
П12

Павловская Т. А., Щупак Ю. А.

П12 С/С++. Структурное и объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2011. — 352 с.: ил. — (Серия «Учебное пособие»).

ISBN 978-5-459-00613-1

Практикум предназначен для изучения языка программирования С++ на семинарах или самостоятельно. Издание дополняет и расширяет учебник Т. А. Павловской «С/С++. Программирование на языке высокого уровня», но может использоваться и как отдельное пособие.

На примерах, сопровождаемых теоретическими сведениями, рассматриваются: основные конструкции, массивы, строки, структуры, функции, шаблоны, динамические структуры данных, классы, шаблоны, наследование, исключения, стандартная библиотека, UML, концепции программной инженерии и паттерны проектирования. Обсуждаются алгоритмы, приемы отладки, вопросы качества и стиля. По каждой теме приведено несколько комплектов заданий для лабораторных работ.

ББК 32.973.2-018.1
УДК 004.43(075)

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-459-00613-1

© ООО Издательство «Питер», 2011

Краткое оглавление

Предисловие	10
--------------------------	-----------

Часть I. Структурное программирование	13
--	-----------

Семинар 1. Линейные программы	14
Семинар 2. Ветвления и циклы.....	26
Семинар 3. Одномерные массивы и указатели.....	40
Семинар 4. Двумерные массивы.....	55
Семинар 5. Строки и файлы.....	68
Семинар 6. Структуры	83
Семинар 7. Функции	99
Семинар 8. Перегрузка и шаблоны функций.....	121
Семинар 9. Динамические структуры данных.....	128

Часть II. Объектно-ориентированное программирование	159
--	------------

Семинар 10. Классы	160
Семинар 11. Наследование.....	194
Семинар 12. Шаблоны классов. Обработка исключительных ситуаций	235
Семинар 13. Стандартные потоки.....	266
Семинар 14. Файловые и строковые потоки. Строки класса string.....	283
Семинар 15. Стандартная библиотека шаблонов	294
Приложение. Основные приемы работы в Microsoft Visual C++.NET 2005.....	332

Литература.....	339
------------------------	------------

Алфавитный указатель	341
-----------------------------------	------------

Оглавление

Предисловие	10
Часть I. Структурное программирование	13
Семинар 1. Линейные программы	14
Задача 1.1. Расчет по формуле.....	14
Вывод кириллицы.....	18
Отладка программы.....	18
Описание переменных.....	21
Задача 1.2. Временной интервал	22
Итоги	23
Задания	24
Семинар 2. Ветвления и циклы	26
Разветвляющиеся программы	26
Задача 2.1. Вычисление значения функции, заданной графически	26
Задача 2.2. Выстрел по мишени.....	29
Задача 2.3. Клавиши курсора	31
Циклы.....	32
Задача 2.4. Таблица значений функции.....	33
Задача 2.5. Вычисление суммы ряда.....	35
Итоги	37
Задания	37
Семинар 3. Одномерные массивы и указатели	40
Задача 3.1. Количество элементов между минимумом и максимумом	42
Задача 3.2. Сумма элементов правее последнего отрицательного	45
Задача 3.3. Быстрая сортировка массива	47
Итоги	50
Задания	50
Семинар 4. Двумерные массивы	55
Задача 4.1. Среднее арифметическое и количество положительных элементов массива.....	58

Задача 4.2. Номер столбца из положительных элементов.....	60
Задача 4.3. Упорядочивание строк матрицы.....	61
Итоги	64
Задания	64
Семинар 5. Строки и файлы.....	68
Описание строк	68
Операции со строками.....	71
Работа с символами	72
Задача 5.1. Поиск подстроки.....	73
Задача 5.2. Подсчет количества вхождений слова в текст	75
Задача 5.3. Вывод вопросительных предложений.....	78
Итоги	81
Задания	81
Семинар 6. Структуры	83
Задача 6.1. Поиск в простой базе (массив структур).....	84
Задача 6.2. Сортировка массива структур	89
Задача 6.3. Структуры и бинарные файлы.....	90
Задача 6.4. Структуры в динамической памяти	92
Итоги	93
Задания	94
Семинар 7. Функции.....	99
Задача 7.1. Передача в функцию параметров стандартных типов	101
Задача 7.1-а. Передача в функцию имени функции.....	105
Задача 7.2. Передача массивов в функцию.....	107
Задача 7.3. Передача строк в функцию	108
Задача 7.4. Передача структур в функцию	109
Задача 7.5. Рекурсивные функции	111
Многофайловые проекты	113
Что и как следует размещать в заголовочном файле.....	113
Задача 7.6. Многофайловый проект — форматирование текста.....	115
Итоги	119
Задания	120
Функции и массивы.....	120
Функции, строки и файлы	120
Функции, структуры и бинарные файлы.....	120
Семинар 8. Перегрузка и шаблоны функций.....	121
Перегрузка функций	121
Задача 8.1. Перегрузка функций.....	122
Шаблоны функций	124
Задача 8.2. Шаблоны функций.....	125
Итоги	126
Задания	127
Семинар 9. Динамические структуры данных	128
Задача 9.1. Стек.....	128
Задача 9.2. Линейный список.....	131

Задача 9.3. Бинарное дерево	138
Итоги	150
Задания	151

Часть II. Объектно-ориентированное программирование 159

Семинар 10. Классы.....	160
Появление объектно-ориентированного программирования	160
Критерии качества декомпозиции проекта.....	161
Что принесло с собой ООП	162
От структуры — к классу.....	163
Задача 10.1. Поиск в простой базе (массив объектов)	163
Конструктор по умолчанию.....	170
Инициализаторы конструктора	170
Конструктор копирования.....	171
Перегрузка операций	172
Перегрузка операций инкремента	173
Перегрузка операции присваивания	174
Статические элементы класса.....	175
Задача 10.2. Реализация класса треугольников.....	175
Этап 1	177
Этап 2	182
Этап 3	184
Этап 4	187
Итоги	189
Задания	189
Семинар 11. Наследование.....	194
Наследование классов	194
Замещение функций базового класса	195
Конструкторы и деструкторы в производном классе	196
Устранение неоднозначности при множественном наследовании.....	196
Доступ к объектам иерархии	197
Виртуальные методы.....	198
Абстрактные классы. Чисто виртуальные методы	199
Отношения между классами. Диаграммы классов на языке UML.....	199
Ассоциация	200
Наследование.....	201
Агрегация	201
Зависимость	202
Паттерны проектирования.....	202
Паттерн Стратегия (Strategy).....	204
Проектирование программы с учетом будущих изменений	205
Задача 11.1. Функциональный калькулятор	207
Задача 11.2. Продвинутый функциональный калькулятор	215
Задача 11.3. Работа с объектами символьных и шестнадцатеричных строк	221

Итоги	231
Задания	232
Общая часть заданий для вариантов 1–20.....	232
Варианты 1–10.....	232
Варианты 11–20	234
Семинар 12. Шаблоны классов. Обработка исключительных ситуаций	235
Шаблоны классов.....	235
Определение шаблона класса.....	235
Использование шаблона класса	236
Организация исходного кода	237
Параметры шаблонов	238
Специализация.....	238
Использование функциональных объектов для настройки шаблонных классов.....	239
Разработка шаблонного класса для представления разреженных массивов.....	241
Задача 12.1. Шаблонный класс для разреженных массивов.....	241
Обработка исключительных ситуаций.....	246
Определение исключений	247
Перехват исключений	248
Неперехваченные исключения	249
Классы исключений. Иерархии исключений	250
Спецификации исключений.....	251
Исключения в конструкторах	252
Исключения в деструкторах.....	254
Задача 12.2. Шаблонный класс векторов.....	255
Итоги	263
Задания	264
Семинар 13. Стандартные потоки.....	266
Потоковые классы.....	266
Классы стандартных потоков.....	267
Заголовочные файлы библиотеки ввода-вывода C++	267
Объекты и методы стандартных потоков ввода-вывода.....	267
Обработка ошибок потоков	269
Перегрузка операций извлечения и вставки для типов, определенных программистом	270
Задача 13.1. Первичный ввод и поиск информации в базе данных.....	271
Итоги	276
Задания	276
Семинар 14. Файловые и строковые потоки. Строки класса string	283
Файловые потоки.....	283
Строковые потоки.....	286
Строки класса string.....	287
Задача 14.1. Подсчет количества вхождений слова в текст.....	290
Задача 14.2. Вывод вопросительных предложений.....	291
Итоги	293
Задания	293

Семинар 15. Стандартная библиотека шаблонов	294
Основные концепции стандартной библиотеки.....	294
Контейнеры	294
Итераторы.....	295
Общие свойства контейнеров	297
Алгоритмы	298
Использование последовательных контейнеров	299
Задача 15.1. Сортировка вектора	301
Шаблонная функция print для вывода содержимого контейнера.....	302
Адаптеры контейнеров	303
Стек	303
Очередь	304
Очередь с приоритетами.....	304
Использование алгоритмов	305
Алгоритмы count и find.....	305
Алгоритмы count_if и find_if	306
Алгоритм for_each	306
Алгоритм search	306
Алгоритм sort	307
Функциональные объекты	308
Обратные итераторы	309
Итераторы вставки и алгоритм sorsu.....	309
Алгоритм merge	310
Использование ассоциативных контейнеров.....	311
Множества	311
Словари.....	313
Задача 15.2. Формирование частотного словаря.....	313
Задача 15.3. Морской бой	315
Итоги	326
Задания	326
Приложение. Основные приемы работы в Microsoft Visual C++.NET 2005	332
Запуск интегрированной среды.Создание пустого проекта	332
Добавление файлов к проекту	333
Многофайловые проекты	334
Редактирование текста программы.....	334
Сохранение проекта и программы, завершение работы.....	335
Продолжение работы над проектом	335
Компиляция и компоновка программы	335
Выполнение и отладка программы	336
Работа со справочной системой	337
Литература.....	339
Алфавитный указатель	341

Предисловие

Эта книга не просто объединяет под одной обложкой практикумы «Структурное программирование» и «Объектно-ориентированное программирование» [22, 23]: материалы были переработаны и обновлены, листинги программ обновлены и проверены в среде Microsoft .NET.

Наша цель — научить читателя самостоятельно создавать грамотные и, по возможности, профессиональные программы на C++. Практикум рассчитан на планомерное освоение этого языка с начального и до профессионального уровня. Извлечь пользу из этой книги смогут в первую очередь студенты, обучающиеся по направлению «Информатика и вычислительная техника» и смежным с ним, а также любознательные школьники, упорные пенсионеры, домохозяйки, безработные менеджеры — в общем, все, кто твердо решил освоить этот непростой язык для практического применения.

Книга состоит из двух частей: «Структурное программирование» и «Объектно-ориентированное программирование». В *первой части* рассматриваются возможности C++, используемые в рамках процедурной парадигмы: стандартные типы данных, управляющие конструкции языка, массивы, указатели, строки, структуры, функции, шаблоны функций, а также динамические структуры данных. Обсуждаются алгоритмы, методы и приемы написания программ, типичные ошибки, которые совершают начинающие (и не только) программисты, вопросы качества и стиля. Большое внимание уделяется процессу отладки и тестирования.

Во *второй части* обсуждаются методы создания объектно-ориентированных программ, шаблонов классов, использование наследования, исключений, классов и алгоритмов стандартной библиотеки, введение в UML и паттерны проектирования.

В подавляющем большинстве учебников по C++ приводятся конструкции языка и примеры, иллюстрирующие их работу. Любой программист, работавший над реальными проектами, понимает, что знания синтаксиса и правил выполнения операторов далеко не достаточно для того, чтобы писать программы приемлемого качества. Это особенно справедливо для такого многогранного языка, как C++.

Если вы не знакомы с основами ООП и с базовыми концепциями *программной инженерии* (software engineering), то написанная вами программа, если и заработает,

скорее всего будет неудобной для сопровождения и модификации, а повторное использование программного кода окажется практически невозможным. Именно поэтому вопросам программной инженерии в нашей книге уделяется особое внимание. Вот краткий перечень этих вопросов.

- ❑ Из каких компонентов (модулей, функций, классов) должна состоять программа?
- ❑ Как распределяются обязанности между этими компонентами?
- ❑ Как компоненты программы взаимодействуют друг с другом?
- ❑ Каким критериям должен удовлетворять программный проект, чтобы его было легко сопровождать и модифицировать?
- ❑ Как применять шаблоны (паттерны) проектирования для достижения этих целей?

Все эти проблемы рассматриваются и разбираются на конкретных задачах, причем особая роль отведена первым двум семинарам второй части, посвященным изучению базовых концепций ООП. Здесь особенно подробно разбирается процесс проектирования программы, активно используются средства отладки и тестирования. Иногда мы специально вносим в текст программы ошибку или проявляем «забывчивость», чтобы продемонстрировать возникающие последствия и, кроме того, привить читателю вкус к аналитической работе детектива, идущего по следу коварного «преступника» — программной ошибки.

Для эффективного восприятия новых технологических идей сегодня не обойтись без знания основ унифицированного языка моделирования UML, ставшего стандартным средством представления проектных решений. Поэтому в практикуме показано применение UML-диаграмм для отображения взаимоотношений между классами.

Практикум является дополнением к учебнику Т. А. Павловской «С/С++. Программирование на языке высокого уровня»¹ [21], но вполне может рассматриваться и как самостоятельное издание, поскольку при разборе программ поясняются все использованные в них возможности языка. Изложение соответствует логике Учебника, а во второй части выходит далеко за его рамки.

Программа обучения рассчитана на 1–3 семестра: в зависимости от объема курса и степени подготовленности студентов количество и содержание лабораторных работ можно варьировать. В конце каждого семинара приводится один или более комплектов из *20 вариантов заданий*, рассчитанных на учебную группу студентов. Тексты заданий большей частью соответствуют Учебнику, но некоторые разделы переработаны и дополнены для более углубленного изучения материала.

Средства С++, рассматриваемые в данной книге, соответствуют последнему *стандарту языка С++* — ANSI ISO/IEC 14882 (2003). Из распространенных компиляторов в достаточной степени подходят, например, Microsoft Visual C++ 2005 и выше (ОС Windows) и gcc (GNU C Compiler — ОС Linux, Cygwin — ОС Windows). Приведенные в книге примеры программ были протестированы с помощью первого из указанных средств.

¹ В дальнейшем для краткости ссылок мы будем называть его просто «Учебник».

Для успешного освоения языка C++ по этой книге пользоваться упомянутыми выше средами совершенно необязательно. Можно работать с любыми компиляторами, рассчитанными на любые платформы (различные версии Windows, Linux, Solaris, FreeBSD и так далее), но при этом необходимо проверять по справочной системе, что изучаемые возможности языка поддерживаются конкретным компилятором. В программах этой книги используются только средства, соответствующие стандарту.

Все ключевые слова, стандартные типы, константы, функции, макросы и классы, описанные в книге, можно найти по алфавитному указателю, что позволяет использовать ее в качестве *справочника*.

В практикуме не рассматривается программирование под Windows: все примеры представляют собой так называемые *консольные приложения*. В основу книги положены семинары, проводимые авторами в Санкт-Петербургском государственном университете информационных технологий, механики и оптики (СПбГУ ИТМО).

В связи с ограничениями на объем книги листинги программ иногда приводятся с сокращениями. Полные тексты листингов вы можете найти на веб-сайте издательства <http://www.piter.com>.

Ваши замечания, пожелания и дополнения к практикуму не ленитесь присылать авторам по адресам pta-ipm@yandex.ru — Т. А. Павловской (по первой части практикума) и shupak@mail.ru — Ю. А. Щупаку (по второй части). Это позволит сделать последующие издания более качественными и полезными.

Часть I. Структурное программирование

В этой части практикума рассматриваются возможности C++, используемые в рамках процедурной (структурной) парадигмы: стандартные типы данных, управляющие конструкции языка, указатели, массивы, строки старого стиля, структуры, функции, шаблоны функций, а также реализация динамических структур данных.

Семинар 1. Линейные программы

Теоретический материал: с. 15–38, 387–390.

Если в программе все операторы выполняются последовательно, один за другим, такая программа называется *линейной*. Рассмотрим в качестве примера программу, вычисляющую результат по заданной формуле.

Задача 1.1. Расчет по формуле

Написать программу, которая переводит температуру в градусах по Фаренгейту в градусы Цельсия по формуле $C = 5/9 (F - 32)$, где C — температура по Цельсию, а F — температура по Фаренгейту.

Перед написанием любой программы надо определить, что является ее исходными данными и результатом работы. В данном случае все просто: вводится одно вещественное число, представляющее собой температуру по Цельсию, а в результате выводится другое вещественное число — температура по Фаренгейту. Алгоритм задан простой формулой, поэтому попробуем сразу написать программу (листинг 1.1). Оговоримся, что первый блин будет комом, но программирование тем и отличается от реальной жизни, что ошибки всегда можно найти и исправить.

Листинг 1.1. Вычисление по формуле — классы ввода-вывода

```
#include <iostream> // 1
using namespace std; // 2
int main() { // 3
    double fahr, cels; // 4
    cout << endl << " Введите температуру по Фаренгейту" << endl; // 5
    cin >> fahr; // 6
    cels = 5 / 9 * ( fahr - 32 ); // 7
    cout << "По Фаренгейту: " << fahr << ", по Цельсию: " << cels << endl; // 8
    return 0; // 9
}
```

Рассмотрим каждую строку программы отдельно. Не расстраивайтесь, если что-то пока останется непонятным, — ведь вся книга еще впереди!

В *первой строке* записана *директива препроцессора*¹, по которой к исходному тексту программы подключается заголовочный файл `<iostream>`. Это текстовый файл, который содержит описания элементов стандартной библиотеки, отвечающих за ввод и вывод. Там описан набор классов для управления вводом-выводом, стандартные объекты-потоки `cin` для ввода с клавиатуры и `cout` для вывода на экран, а также операции *вывода на экран* `<<` (помещения в поток) и *ввода с клавиатуры* `>>` (извлечения из потока). Объекты изучаются во второй части практикума, а пока давайте пользоваться ими, как некими волшебными словами, не пытаясь полностью осознать их смысл, ведь и в реальной жизни с большинством благ цивилизации мы обращаемся подобным же образом.

ВНИМАНИЕ

Директивы препроцессора записываются в отдельной строке, перед знаком `#` могут находиться только пробельные символы.

Строка 2 является описанием, по которому программе становятся доступными все имена, определенные в стандартном заголовочном файле `<iostream>`. Дело в том, что внутри этого файла все описания спрятаны в так называемую *именованную область* (пространство имен) с именем `std`. Это касается всех стандартных заголовочных файлов и сделано для того, чтобы стандартные имена не конфликтовали с теми, которые ввел программист. Конечно, открытие всех стандартных имен из заголовочного файла не может предотвратить подобный конфликт, но есть средства, позволяющие сделать доступными не все, а только нужные имена. Они будут рассмотрены позже.

ПРИМЕЧАНИЕ

Для многих заголовочных файлов существуют их двойники «старого стиля» с расширением `.h`, описания, находящиеся в них, доступны без дополнительных усилий — например, `<iostream.h>`, `<stdio.h>`. Заголовочные файлы, унаследованные из языка C, записываются при этом без ведущей буквы `c`. Этот способ использования заголовочных файлов считается устаревшим.

Программа на C++ состоит из функций. *Функция* — это именованная последовательность операторов. Функция состоит из заголовка и тела. *Оператор 3* представляет собой заголовок главной (а в данном случае и единственной) функции программы. Она должна иметь имя `main`, указывающее, что именно с нее требуется начинать выполнение. Заголовок любой функции пишется по определенным правилам. За именем функции в скобках обычно следует список передаваемых ей параметров. В данном случае он пуст, но скобки необходимы для того, чтобы компилятор мог распознать, что это именно функция, а не другая конструкция языка. Перед именем записан тип значения (`int` — целое), возвращаемого функцией в точку ее вызова (в данном случае — во внешнюю среду).

¹ *Препроцессором* называется предварительная фаза *компиляции* (перевода программы с C++ на машинный язык).

По стандарту главная функция должна возвращать целочисленное значение. Этим в нашей программе занимается *оператор 9*. Впрочем, многие компиляторы реагируют спокойно и в случае его отсутствия, поэтому в большинстве примеров этой книги этот оператор для экономии места не приводится. Тело функции, то есть те операторы, которые требуется выполнить, записываются в фигурных скобках. Фигурные скобки служат для группировки операторов в *блок*.

Для хранения исходных данных и результатов надо выделить достаточно места в оперативной памяти. Для этого служит *оператор 4*. В нашей программе требуется хранить два значения: температуру по Цельсию и температуру по Фаренгейту, поэтому в операторе определяются две переменные. Одна, для хранения температуры по Фаренгейту, названа *fahr*, другая (по Цельсию) — *cels*.

ВНИМАНИЕ

Имена переменным дает программист, исходя из их назначения. Имя может состоять только из латинских букв, цифр и знака подчеркивания и не должно начинаться с цифры. От того, насколько удачно подобраны имена, зависит *читабельность* — одна из важнейших характеристик качества программы.

При описании любой переменной нужно указать ее *тип*, чтобы компилятор знал, сколько выделить места в памяти, как интерпретировать значение переменной (то есть ее внутреннее представление), а также какие действия можно будет выполнять с этой величиной. Например, для вещественных чисел в памяти хранится мантисса и порядок, а целые представляются просто в двоичной форме, поэтому внутреннее представление одного и того же целого и вещественного числа будет различным. Более того, для действий с целыми и вещественными величинами формируются различные наборы машинных команд. Поэтому-то указание типа для каждой переменной является таким важным.

Поскольку температура может принимать не только целые значения, для переменных выбран вещественный тип *double*, позволяющий представлять вещественные числа большого диапазона значений с большой точностью, хотя для данной задачи это даже излишняя роскошь.

ПРИМЕЧАНИЕ

Тип переменных выбирается исходя из требуемой точности представления данных и возможного диапазона значений. Например, нет смысла заводить вещественную переменную для хранения величины, которая может принимать только целые значения, — ведь целочисленные операции выполняются гораздо быстрее.

Для того чтобы пользователь программы (пока что это вы сами) знал, в какой момент требуется ввести с клавиатуры данные, применяется так называемое *приглашение* к вводу (*оператор 5*). На экран выводится указанная в операторе строка символов, и курсор переводится на следующую строку. Строка символов, более строго называемая *символьным литералом*, — это последовательность любых представимых в компьютере символов, заключенная в кавычки.

Стандартный объект, с помощью которого выполняется вывод на экран, называется `cout`. Ему с помощью операции вставки (вывода) `<<` передается то, что мы хотим вывести. Для вывода нескольких элементов используется цепочка таких операций. Для перехода на следующую строку записывается волшебное слово `endl`. Это так называемый *манипулятор*, он управляет («манипулирует») стандартным объектом `cout`. Существуют и другие манипуляторы, с помощью которых можно задать вид выводимой информации. Мы будем рассматривать их по мере необходимости.

СОВЕТ

Не забывайте в дружелюбной манере, но без излишнего многословия, пригласить пользователя ввести исходные данные и указать порядок их ввода. В некоторых случаях может понадобиться указать тип величин.

В *операторе 6* выполняется ввод с клавиатуры одного числа в переменную `fahr`. Для этого используется стандартный объект `cin` и операция извлечения (ввода) `>>`. Как видите, семантика ввода проста и интуитивно понятна: значение со стандартного ввода передается в переменную, указанную справа. Если требуется ввести несколько величин, используется цепочка операций `>>`. При вводе последовательность символов, набранных на клавиатуре, преобразуется во внутреннее представление вещественного числа, которое помещается в отведенную для переменной `fahr` ячейку памяти.

В *операторе 7* вычисляется выражение, записанное справа от *операции присваивания* (обозначаемой знаком `=`), и результат присваивается переменной `cels`, то есть заносится в отведенную этой переменной память. *Выражение* — это правило для вычисления некоторого значения, можно назвать его формулой. Порядок вычислений определяется приоритетом операций (Учебник, с. 384). Уровней приоритетов в языке C++ огорчительно много, поэтому в случае сомнений надо не лениться обращаться к справочной информации.

Основные правила тем не менее просты и соответствуют принятым в математике: вычитание имеет более низкий приоритет, чем умножение, поэтому для того, чтобы оно было выполнено раньше, соответствующая часть выражения заключается в скобки. Деление и умножение имеют одинаковый приоритет и выполняются слева направо, то есть сначала целая константа 5 будет поделена на целую константу 9, а затем результат этой операции умножен на результат вычитания числа 32 из переменной `fahr`. Мы вернемся к обсуждению этого оператора позже, а пока рассмотрим два оставшихся.

Для вывода результата в *операторе 8* применяется уже знакомый нам объект `cout`. Выводится цепочка, состоящая из четырех элементов. Это строка “ По Фаренгейту: “, значение переменной `fahr`, строка “ , по Цельсию: “ и значение переменной `cels`. Обратите внимание, что при выводе строк все символы, находящиеся внутри кавычек, включая и пробелы, выводятся без изменений. При выводе значения переменной выполняется преобразование из внутреннего представления числа в строку символов, представляющую это число. Под значение отводится столько позиций,

сколько необходимо для вывода всех его значащих цифр. Это значит, что, если вывести две переменные подряд, их значения «склеятся»:

```
cout << fahr << cels;           // плохо
cout << fahr << "    " << cels; // чуть лучше
```

СОВЕТ

Всегда предваряйте выводимые значения текстовыми пояснениями.

Наберите текст программы и скомпилируйте ее. Если вы видите сообщения об ошибках, сравните текст на экране с текстом в книге (последний — лучше).

Вывод кириллицы

Мы предполагаем, что вы уже посмотрели в Приложении, как создаются приложения консольного типа, создали пустой проект, набрали текст листинга 1.1, скомпилировали его и запустили на выполнение. Вам приготовлен неприятный сюрприз: вместо приглашения «Введите температуру по Фаренгейту» вы увидите набор каких-то странных символов. Это связано с тем, что вывод в консольное окно выполняется в кодировке¹ ASCII, которая использовалась еще в операционной системе MS-DOS, а текст программы в окне редактора набран в другой кодировке.

Решений задачи вывода кириллицы, которая является частью более общей проблемы *локализации приложений*, несколько. Одно из самых простых — использование функции `setlocale`, унаследованной из библиотеки C. Для ее применения необходимы две вещи: чтобы в операционной системе Windows в настройках языков и стандартов была по умолчанию установлена кириллица и подключить к программе заголовочный файл `<locale>`:

```
#include <locale>           // вставить в листинг 1.1 после оператора 1
    setlocale( LC_ALL, "Russian" ); // вставить в листинг 1.1 после оператора 3
```

Теперь приглашение должно выводиться нормально и можно приступить к отладке.

Отладка программы

Запустите программу на выполнение несколько раз, задавая различные значения температуры. Не забудьте, что дробная часть вещественного числа при вводе отделяется точкой, а не запятой. Можно задавать и целые числа — они будут автоматически преобразованы в вещественную форму.

Как вы можете видеть, результат выполнения программы со стабильностью, достойной лучшего применения, оказывается равным нулю! Это происходит из-за способа вычисления выражения. Давайте вновь обратимся к оператору 7 листинга 1.1. Константы 5 и 9 имеют целый тип, поэтому результат их деления также целочисленный. Округления при этом не происходит, дробная часть всегда отбрасывается.

¹ Кодировкой, или кодовой страницей, называется набор двоичных кодов для некоторого множества символов.

Естественно, что результат дальнейших вычислений не может быть ничем, кроме нуля. Исправить эту ошибку просто — достаточно записать хотя бы одну из констант в виде вещественного числа или изменить порядок вычислений:

```
cels = 5.0 / 9 * ( fahr - 32 );           // вариант 1
cels = 5 * ( fahr - 32 ) / 9;           // вариант 2
```

Вещественная константа 5.0 по умолчанию имеет тип `double`, и при делении происходит автоматическое преобразование к этому же типу константы 9. Во втором варианте сначала выполняется преобразование константы 32 к типу `double`, как к наиболее длинному из участвующих в вычислении разности, а потом к нему автоматически приводятся и остальные константы. Теперь программа работает верно и выдает в результате, например, следующее:

```
Введите температуру по Фаренгейту
451
По Фаренгейту: 451, по Цельсию: 232.778
```

Обратите внимание, что наряду с результатом вычислений мы вывели и исходные данные. Это правильная привычка, и надеемся, что вы тоже будете ей следовать.

СОВЕТ

Начинающие часто тратят время на поиск ошибки в алгоритме, не удостоверившись, что программа работает с правильными данными. Рекомендуем всегда выводить исходные данные сразу же после ввода, чтобы исключить ошибки.

Как видите, даже в таком простом примере можно допустить ошибки! В данном случае заметить их легко, но так происходит далеко не всегда, поэтому запомните важное правило: *надо всегда заранее знать, что должна выдать программа*. Добиться этого можно разными способами, например, вычислением результатов в уме или на калькуляторе, а в более сложных случаях — расчетами по альтернативной или упрощенной методике. Это убережет вас от многих часов, бесплодно и безрадостно проведенных за компьютером. Не поленимся повторить еще раз.

ВНИМАНИЕ

Для отладки необходимо иметь *тестовые примеры*, содержащие исходные данные и ожидаемые результаты. Количество примеров зависит от алгоритма.

Мы будем неоднократно возвращаться к обсуждению состава тестовых примеров, поскольку самое важное качество любой программы — *надежность*. Программы, работающие только с определенными исходными данными, да и то лишь в бережных руках хозяина, никому не нужны.

Давайте теперь напишем ту же программу другим способом — с использованием функций библиотеки C++, унаследованных из языка C (листинг 1.2). Этот способ также применяется достаточно часто, потому что в использовании этих функций есть свои преимущества. Когда вы их оцените, сможете выбирать для каждой программы наиболее подходящий способ ввода-вывода.

Листинг 1.2. Вычисление по формуле – функции библиотеки C

```
#include <stdio> // 1
#include <locale> // 2
using namespace std; // 3
int main() { // 4
    setlocale( LC_ALL, "Russian" ); // 5
    double fahr, cels; // 6
    printf( "\n Введите температуру по Фаренгейту\n" ); // 7
    scanf( "%lf", &fahr ); // 8
    cels = 5.0 / 9 * ( fahr - 32 ); // 9
    printf( " По Фаренгейту: %6.2lf, по Цельсию: %6.2lf\n", fahr, cels ); // 10
}
```

Рассмотрим отличия этой программы от предыдущей. Как видите, к программе подключается другой заголовочный файл — `<stdio>`. Он содержит описание функций, констант и других элементов, относящихся ко вводу-выводу «в стиле C».

Функция `printf` в *операторе 7* выполняет вывод переданного ей *строкового литерала*, то есть последовательности любых символов в кавычках, на стандартное устройство вывода (дисплей). Символы `\n` называются *управляющей последовательностью*. Есть разные управляющие последовательности, все они начинаются с обратной косой черты. Данная задает переход на следующую строку.

Для ввода исходных данных в *операторе 8* используется функция `scanf`. В ней требуется указать формат вводимых значений, а также адреса переменных, которым они будут присвоены. Параметры любой функции перечисляются через запятую. В первом параметре функции `scanf` в виде строкового литерала задается *спецификация формата* вводимой величины, соответствующая типу переменной. Спецификация `%lf` соответствует типу `double`¹. Вторым параметром функции передается адрес переменной, по которому будет помещено вводимое значение. *Операция взятия адреса* обозначается `&`.

Для вывода результата в *операторе 10* применяется уже знакомая вам функция `printf`. Теперь в ней три параметра. Первый, имеющий вид строкового литерала, задает вид и формат выводимой информации. Второй и третий параметры представляют собой имена переменных. При выводе форматные спецификации `%lf` заменяются конкретными значениями переменных `fahr` и `cels`, курсор переходит на следующую строку в соответствии с последовательностью `\n`, а все остальные символы литерала выводятся без изменений.

Вид вывода значений можно настроить при помощи так называемых *модификаторов формата* — чисел, которые записаны перед спецификацией. Первое число задает минимальное количество позиций, отводимых под выводимую величину, второе — сколько из этих позиций отводится под ее дробную часть. Необходимо учитывать, что десятичная точка тоже занимает одну позицию. Если заданного количества позиций окажется недостаточно для размещения числа, компилятор нам это простит и автоматически выделит поле достаточной длины.

¹ Приведем еще две наиболее употребительные спецификации: `%d` — для величин целого типа в десятичной системе счисления, `%f` — для величин типа `float`. Более полный список спецификаций см. в Учебнике на с. 387.

На наш взгляд, листинг 1.1 более нагляден и лучше защищен от ошибок кодирования. С другой стороны, с помощью функций в стиле C легче настраивать вид выводимой информации. Однако при записи форматов легко ошибиться, и компилятор об этом ничего не сообщит; вы будете ломать голову в поисках ошибки в алгоритме, в то время как источником неприятностей будет функция `printf`.

Надо заметить, что при использовании классов также можно задавать любую форму представления информации, но пока не время вдаваться в подобные детали. Для каждой программы желательно выбрать один способ вывода (либо с помощью функций, либо с помощью классов), поскольку смешивать их не рекомендуется.

СОВЕТ

Используйте функции ввода-вывода там, где требуется тщательное форматирование результатов, а классы — в остальных случаях.

Рассмотрим в более общем виде очень важный для дальнейшей работы вопрос — описание переменных.

Описание переменных

Любая переменная обладает двумя основными характеристиками: *временем жизни* и *областью действия*. Они зависят от места и способа описания переменной.

Если переменная описана вне любого блока (в частности, функции), она называется *глобальной* и изначально обнуляется, если вы не предусмотрели ее инициализацию каким-то другим значением. Время ее жизни — с начала выполнения программы и до ее окончания, а область действия (область, в которой эту переменную можно использовать, обратившись к ней по имени) — весь файл, в котором она описана, начиная с точки описания.

Переменная, описанная внутри блока (в частности, внутри функции `main`), является *локальной*, память под нее выделяется в момент выполнения оператора описания и не обнуляется. Областью ее действия является блок, в котором она описана, начиная с точки описания. Время ее жизни также ограничено этим блоком. Сколько раз выполняется блок, столько раз «рождается» и «умирает» локальная переменная.

Есть разновидность локальных переменных — *статические*. Они, подобно глобальным, существуют на всем протяжении выполнения программы и инициализируются однократно. С другой стороны, они, как локальные переменные, видны только в своем блоке. Для описания статических переменных используется ключевое слово `static`. Ниже приведен пример описания трех переменных и таблица, в которой суммируются сведения о видах переменных:

```
int a;                // глобальная переменная
int main() {
    static int b = 1;  // локальная статическая переменная
    int c;             // локальная переменная
}
```

	Глобальная	Локальная статическая	Локальная
Имя	a	b	c
Размещение	сегмент данных	сегмент данных	сегмент стека
Время жизни	вся программа	вся программа	блок
Область видимости	файл	блок	блок
Инициализация	да	да	нет

Память под все эти переменные выделяет компилятор. Кроме перечисленных, существуют *динамические* переменные, память под которые резервируется во время выполнения программы с помощью операции `new` в динамической области памяти, или *хипе* (heap, куча). Доступ к таким переменным осуществляется не по имени, а через указатели. Мы рассмотрим их на третьем семинаре.

Во всех рассмотренных выше программах переменные являются локальными. Глобальные переменные нужно стремиться использовать как можно реже. Запомните, что *переменная должна иметь минимальную из возможных областей действия*, поскольку это значительно облегчает поиск ошибок. На следующих семинарах вам предстоит неоднократно в этом убедиться. Еще одно важное правило — всегда стремиться *инициализировать переменную при описании*.

Задача 1.2. Временной интервал

Заданы моменты начала и конца некоторого промежутка времени в часах, минутах и секундах (в пределах одних суток). Найти продолжительность этого промежутка в тех же единицах.

Исходными данными для этой задачи являются шесть целых величин, задающих моменты начала и конца интервала, *результатами* — три целых величины.

Вы уже знаете, что тип переменной выбирается исходя из диапазона и требуемой точности представления данных, а имя дается в соответствии с ее содержимым. Нам потребуется хранить исходные данные, не превышающие величины 60 для минут и секунд и величины 24 для часов, поэтому можно ограничиться коротким целым типом (`short int`, сокращенно `short`). Назовем переменные для хранения начала интервала `hour1`, `min1` и `sec1`, для хранения конца интервала — `hour2`, `min2` и `sec2`, а результирующие величины — `hour`, `min` и `sec`.

Для решения этой задачи необходимо преобразовать оба момента времени в секунды, вычесть первый из второго, а затем преобразовать результат обратно в часы, минуты и секунды. Следовательно, нам потребуется промежуточная переменная, в которой будет храниться интервал в секундах. Она может принимать весьма большие значения, ведь в сутках 86 400 секунд.

В величинах типа `short` могут храниться значения, не превышающие 32 767 для величин со знаком (`signed short`) и 65 535 для величин без знака (`unsigned short`), поэтому тип `short` здесь использовать нельзя. Вот почему для этой переменной следует выбрать длинный целый тип (`long int`, сокращенно `long`). «Обычный» целый

тип `int` в зависимости от архитектуры компьютера может совпадать либо с коротким, либо с длинным целым типом. Текст программы приведен в листинге 1.3.

Листинг 1.3. Вычисление интервала времени

```
#include <iostream>
using namespace std;
int main() {
    short hour1, min1, sec1, hour2, min2, sec2, hour, min, sec;
    cout << endl << " Введите время начала интервала (час мин сек)" << endl;
    cin >> hour1 >> min1 >> sec1;
    cout << endl << " Введите время конца интервала (час мин сек)" << endl;
    cin >> hour2 >> min2 >> sec2;
    long sum_sec = ( hour2 - hour1 ) * 3600 + ( min2 - min1 ) * 60 + sec2 - sec1;
    hour = sum_sec / 3600;
    min  = ( sum_sec - hour * 3600 ) / 60;
    sec  = sum_sec - hour * 3600 - min * 60;
    cout << " Продолжительность промежутка от " <<
        hour1 << ':' << min1 << ':' << sec1 << " до " <<
        hour2 << ':' << min2 << ':' << sec2 << endl << " равна " <<
        hour << ':' << min  << ':' << sec  << endl;
}
```

Для перевода результата из секунд обратно в часы и минуты используется знакомый вам эффект отбрасывания дробной части при делении целого числа на целое. Протестируйте программу на различных наборах исходных данных.

ВНИМАНИЕ

Данные при вводе разделяются пробелами, символами перевода строки или табуляции (но не запятыми!).

Итоги

1. Приступая к написанию программы, четко определите, что является ее исходными данными и что требуется получить в результате.
2. Выбирайте тип переменных с учетом диапазона и требуемой точности представления данных. Давайте переменным имена, отражающие их назначение.
3. Отдавайте предпочтение локальным переменным перед глобальными. Переменная должна иметь минимальную из возможных областей действия. Всегда инициализируйте переменные при описании.
4. При записи выражений обращайте внимание на приоритет операций и типы операндов.
5. При использовании стандартных функций или классов с помощью директивы `#include` подключите к программе соответствующие заголовочные файлы. Не смешивайте в одной программе ввод-вывод с помощью классов (в стиле C++) и с помощью функций библиотеки (в стиле C).

- Ввод с клавиатуры предваряйте приглашением, а выводимые значения — пояснениями. Данные при вводе разделяйте пробелами, символами перевода строки или табуляции. Для контроля сразу же после ввода выводите исходные данные на экран.
- В функциях `printf` и `scanf` для каждой переменной указывайте спецификацию формата, соответствующую ее типу. Не забывайте, что в `scanf` передается адрес переменной, а не ее значение.
- До запуска программы подготовьте тестовые примеры, содержащие исходные данные и ожидаемые результаты. Отдельно проверьте реакцию программы на неверные исходные данные.

Задания

Напишите программу расчета по двум формулам. Предварительно подготовьте тестовые примеры (результат вычисления по первой формуле должен в большинстве вариантов совпадать со второй). Список математических функций библиотеки C++ приведен в Учебнике на с. 410. Для их использования необходимо подключить к программе заголовочный файл `<cmath>`. Отсутствующие в библиотеке функции выразите через имеющиеся.

- $$z_1 = 2 \sin^2(3\pi - 2\alpha) \cos^2(5\pi + 2\alpha),$$

$$z_2 = \frac{1}{4} - \frac{1}{4} \sin\left(\frac{5}{2}\pi - 8\alpha\right).$$
- $$z_1 = \cos \alpha + \sin \alpha + \cos 3\alpha + \sin 3\alpha,$$

$$z_2 = 2\sqrt{2} \cos \alpha \cdot \sin\left(\frac{\pi}{4} + 2\alpha\right).$$
- $$z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos \alpha + 1 - 2 \sin^2 2\alpha},$$

$$z_2 = 2 \sin \alpha.$$
- $$z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos \alpha - \cos 3\alpha + \cos 5\alpha},$$

$$z_2 = \operatorname{tg} 3\alpha.$$
- $$z_1 = 1 - \frac{1}{4} \sin^2 2\alpha + \cos 2\alpha,$$

$$z_2 = \cos^2 \alpha + \cos^4 \alpha.$$
- $$z_1 = \cos \alpha + \cos 2\alpha + \cos 6\alpha + \cos 7\alpha,$$

$$z_2 = 4 \cos \frac{\alpha}{2} \cdot \cos \frac{5}{2}\alpha \cdot \cos 4\alpha.$$
- $$z_1 = \cos^2\left(\frac{3}{8}\pi - \frac{\alpha}{4}\right) - \cos^2\left(\frac{11}{8}\pi + \frac{\alpha}{4}\right),$$

$$z_2 = \frac{\sqrt{2}}{2} \sin \frac{\alpha}{2}.$$
- $$z_1 = \cos^4 x + \sin^2 y + \frac{1}{4} \sin^2 2x - 1,$$

$$z_2 = \sin(y + x) \cdot \sin(y - x).$$
- $$z_1 = (\cos \alpha - \cos \beta)^2 - (\sin \alpha - \sin \beta)^2,$$

$$z_2 = -4 \sin^2 \frac{\alpha - \beta}{2} \cdot \cos(\alpha + \beta).$$

$$10. \quad z_1 = \frac{\sin\left(\frac{\pi}{2} + 3\alpha\right)}{1 - \sin(3\alpha - \pi)},$$

$$z_2 = \operatorname{ctg}\left(\frac{5}{4}\pi + \frac{3}{2}\alpha\right).$$

$$11. \quad z_1 = \frac{1 - 2\sin^2 \alpha}{1 + \sin 2\alpha},$$

$$z_2 = \frac{1 - \operatorname{tg} \alpha}{1 + \operatorname{tg} \alpha}.$$

$$12. \quad z_1 = \frac{\sin 4\alpha}{1 + \cos 4\alpha} \cdot \frac{\cos 2\alpha}{1 + \cos 2\alpha},$$

$$z_2 = \operatorname{ctg}\left(\frac{3}{2}\pi - \alpha\right).$$

$$13. \quad z_1 = \frac{\sin \alpha + \cos(2\beta - \alpha)}{\cos \alpha - \sin(2\beta - \alpha)},$$

$$z_2 = \frac{1 + \sin 2\beta}{\cos 2\beta}.$$

$$14. \quad z_1 = \frac{\cos \alpha + \sin \alpha}{\cos \alpha - \sin \alpha},$$

$$z_2 = \operatorname{tg} 2\alpha + \sec 2\alpha.$$

$$15. \quad z_1 = \frac{\sqrt{2b + 2\sqrt{b^2 - 4}}}{\sqrt{b^2 - 4} + b + 2},$$

$$z_2 = \frac{1}{\sqrt{b + 2}}.$$

$$16. \quad z_1 = \frac{x^2 + 2x - 3 + (x + 1)\sqrt{x^2 - 9}}{x^2 - 2x - 3 + (x - 1)\sqrt{x^2 - 9}},$$

$$z_2 = \sqrt{\frac{x + 3}{x - 3}}.$$

$$17. \quad z_1 = \frac{\sqrt{(3m + 2)^2 - 24m}}{3\sqrt{m} - \frac{2}{\sqrt{m}}},$$

$$z_2 = -\sqrt{m}.$$

$$18. \quad z_1 = \left(\frac{a + 2}{\sqrt{2a}} - \frac{a}{\sqrt{2a} + 2} + \frac{2}{a - \sqrt{2a}} \right) \cdot \frac{\sqrt{a} - \sqrt{2}}{a + 2}, \quad z_2 = \frac{1}{\sqrt{a} + \sqrt{2}}.$$

$$19. \quad z_1 = \left(\frac{1 + a + a^2}{2a + a^2} + 2 - \frac{1 - a + a^2}{2a - a^2} \right)^{-1} (5 - 2a^2), \quad z_2 = \frac{4 - a^2}{2}.$$

$$20. \quad z_1 = \frac{(m - 1)\sqrt{m} - (n - 1)\sqrt{n}}{\sqrt{m^3n + nm + m^2 - m}},$$

$$z_2 = \frac{\sqrt{m} - \sqrt{n}}{m}.$$

Семинар 2. Ветвления и циклы

Разветвляющиеся программы

Теоретический материал: с. 40–44.

В линейной программе все операторы выполняются последовательно, один за другим. Таким способом можно записать только очень простые алгоритмы. Для того чтобы в зависимости от конкретных значений исходных данных обеспечить выполнение разных последовательностей операторов, применяются операторы ветвления `if` и `switch`. Оператор `if` обеспечивает передачу управления на одну из двух ветвей вычислений, а оператор `switch` — на одну из произвольного количества ветвей. Рассмотрим сначала задачи с применением оператора `if`.

Задача 2.1. Вычисление значения функции, заданной графически

Написать программу, которая по введенному значению аргумента вычисляет значение функции, заданной в виде графика (рис. 2.1).

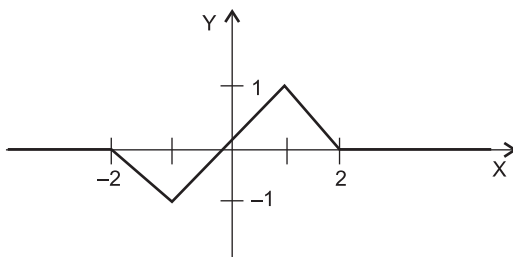


Рис. 2.1. Функция, заданная в виде графика

Очевидно, что *исходными данными* для этой задачи является вещественное значение аргумента x , который определен на всей числовой оси, а *результатом* — вещественное значение функции y .

Перед написанием программы следует составить *алгоритм* ее решения — сначала в общем виде, а затем постепенно детализируя каждый шаг. Такой способ,

называемый *нисходящей разработкой*, позволяет создавать простые по структуре программы. По мере приобретения опыта вы убедитесь, насколько это важно. Для начала запишем функцию в виде системы формул:

$$y = \begin{cases} 0, & x < -2; \\ -x - 2, & -2 \leq x < -1; \\ x, & -1 \leq x < 1; \\ -x + 2, & 1 \leq x < 2; \\ 0, & x \geq 2. \end{cases}$$

Алгоритм можно описать в различном виде, например в виде блок-схемы. Мы выбрали неформальный словесный вид, который настоятельно рекомендуем: ведь только после того, как задача четко сформулирована и описана на естественном языке, ее можно успешно записать на языке программирования. Для такого простого алгоритма это, быть может, и не имеет большого смысла, но вы ведь не собираетесь в дальнейшем ограничиваться подобными задачами!

1. Ввести значение аргумента x .
2. Определить, какому интервалу из области определения функции принадлежит введенное значение аргумента, и вычислить значение функции y по соответствующей формуле.
3. Вывести значение y .

Опишем второй пункт алгоритма более подробно:

- 2.1. Если аргумент x принадлежит интервалу $[-\infty; -2)$, то $y = 0$.
- 2.2. Если аргумент x принадлежит интервалу $[-2; -1)$, то $y = -x - 2$.
- 2.3. Если аргумент x принадлежит интервалу $[-1; 1)$, то $y = x$.
- 2.4. Если аргумент x принадлежит интервалу $[1; 2)$, то $y = -x + 2$.
- 2.5. Если аргумент x принадлежит интервалу $[2; +\infty]$, то $y = 0$.

По такому алгоритму можно «один-в-один» написать программу (листинг 2.1).

Листинг 2.1. Вычисление значения функции, заданной графически

```
#include <iostream>
using namespace std;
int main() {
    float x, y;
    cout << " Введите значение аргумента" << endl;
    cin >> x;
    if ( x < -2 )           y = 0;
    if ( x >= -2 && x < -1 ) y = -x - 2;
    if ( x >= -1 && x < 1 ) y = x;
    if ( x >= 1 && x < 2 ) y = -x + 2;
    if ( x >= 2 )           y = 0;
    cout << " Для x = " << x << " значение функции y = " << y << endl;
}
```

Тестовые примеры для этой программы должны включать по крайней мере по одному значению аргумента из каждого интервала, а для проверки граничных условий — еще и все точки перегиба (если это кажется вам излишним, попробуйте в последнем условии «забыть» знак `=`, а затем ввести значение `x`, равное 2).

Обратите внимание на запись условий, содержащих два сравнения. Операции отношения (`<`, `>`, `==`, `<=`, `>=`, `!=`) являются бинарными, то есть имеют два операнда, и формируют результат типа `bool`, равный `true` или `false`. Поскольку необходимо, чтобы эти условия выполнялись одновременно, они объединены с помощью операции логического И (`&&`) — не путать с поразрядным И! Приоритет у операции И ниже, чем у операций отношения, поэтому заключать их в скобки не требуется.

Весьма распространенная ошибка начинающих — запись подобных условий в виде кальки с математической формулы, то есть как `a<b<c`. Синтаксической ошибки в этом выражении нет, компилятор выдает предупреждение (а более старые версии компилятора и вообще молчат). Давайте посмотрим, что же происходит при вычислении. Операции отношения одного приоритета выполняются слева направо, поэтому сначала будет выполнена операция `a<b` и сформирован результат в виде `true` или `false`. Следующая операция будет выглядеть как `true<c` или `false<c`. Для ее выполнения значения `true` и `false` преобразуются, соответственно, в единицу и ноль того же типа, что и `c`, и формируется результат, смысл которого вряд ли оправдывает ожидания.

При работе приведенной выше программы всегда выполняются один за другим все пять условных операторов, при этом истинным оказывается только одно условное выражение и, соответственно, присваивание значения переменной `y` выполняется один раз. Запишем условные операторы так, чтобы уменьшить количество проверок:

```
if      ( x <= -2 ) y = 0;
else if ( x < -1 )  y = -x - 2;
else if ( x <  1 )  y = x;
else if ( x <  2 )  y = -x + 2;
else                y = 0;
```

Здесь проверка на принадлежность аргумента очередному интервалу выполняется только в том случае, если `x` не входит в предыдущий интервал. Программа получилась более компактной и эффективной, но менее понятной, поскольку содержит вложенные операторы. В отличие от предыдущей версии, порядок следования условных операторов имеет здесь важное значение.

Какой вариант лучше? В современной иерархии критериев качества программы на первом месте стоят ее надежность, простота поддержки и модификации, а эффективность и компактность отходят на второй план. Поэтому в общем случае, если нет специальных требований к быстродействию, лучше более наглядный вариант. Первый вариант понятен с первого взгляда, поскольку он фактически представляет собой «кальку» с формулы, задающей поведение функции.

Задача 2.2. Выстрел по мишени

Дана заштрихованная область (рис. 2.2) и точка с координатами (x, y) . Написать программу, определяющую, попадает ли точка в область. Результат вывести в виде текстового сообщения.

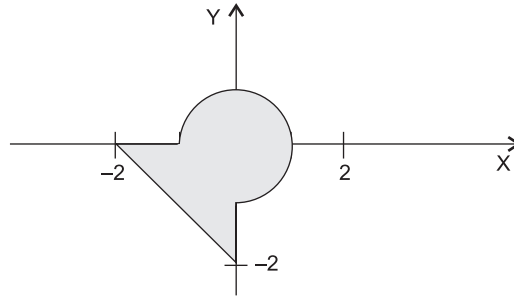


Рис. 2.2. Заштрихованная область для задачи 2.2

Запишем условия попадания точки в область в виде формул. Область можно описать как круг, пересекающийся с треугольником. Точка может попадать либо в круг, либо в треугольник, либо в их общую часть:

$$\{x^2 + y^2 \leq 1\} \quad \text{или} \quad \begin{cases} x \leq 0 \\ y \leq 0 \\ y \geq -x - 2 \end{cases}$$

Первое условие задает попадание точки в круг, второе — в треугольник. Программа для решения задачи приведена в листинге 2.2.

Листинг 2.2. Попадание в заштрихованную область

```
#include <iostream>
using namespace std;
int main() {
    double x, y;
    cout << " Введите значения x и y:" << endl;
    cin >> x >> y;
    if ( x * x + y * y <= 1 || x <= 0 && y <= 0 && y >= - x - 2 )
        cout << " Точка попадает в область" << endl;
    else cout << " Точка не попадает в область" << endl;
}
```

Все условия попадания в треугольник должны выполняться одновременно, поэтому они объединены операцией И (&&). Ее приоритет выше, чем у ИЛИ (||), и ниже, чем у операций отношения, поэтому заключать эти условия в скобки не требуется. Выполните программу несколько раз, задавая различные положения точки. Задайте заштрихованную область какого-либо другого вида и измените программу в соответствии с этой областью.

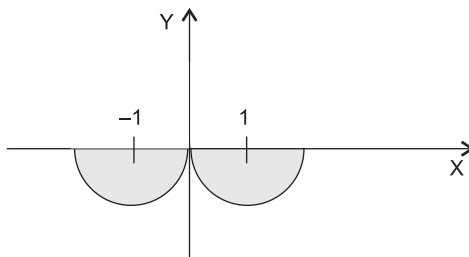


Рис. 2.3. Другая заштрихованная область для задачи 2.2

Рассмотрим пример другой заштрихованной области (рис. 2.3). Условный оператор для определения попадания точки в эту область имеет вид.

```
if ( y < 0 && ( (x - 1) * (x - 1) + y * y <= 1 ||
               (x + 1) * (x + 1) + y * y <= 1 ) )
    cout << " Точка попадает в область";
else cout << " Точка не попадает в область";
```

Точка может попасть либо в правый полукруг, либо в левый, в обоих случаях значение y должно быть отрицательным. Для того чтобы операция ИЛИ была выполнена раньше, чем операция И, необходимы круглые скобки.

СОВЕТ

Для улучшения читабельности программы можно ставить скобки и в тех местах, где они не обязательны, — например, для визуальной группировки условий.

Рассмотрим другие особенности условного оператора, которые надо учитывать при написании программ. Внутри круглых скобок можно объявить одну переменную и присвоить ей выражение, например:

```
if ( int k = f(x) ) a = k * k; else a = k - 1;
```

Область видимости этой переменной ограничивается условным оператором.

Если по какой-либо ветке условия необходимо выполнить более одного действия, их следует объединить в блок с помощью фигурных скобок. В блоке можно также объявлять локальные переменные, их областью видимости является этот блок:

```
if ( x < 0 ) { int i; i = 2; cout << i; }
```

Следует избегать проверки вещественных величин на равенство, вместо этого лучше сравнивать модуль их разности с некоторым малым числом. Это связано с погрешностью представления вещественных значений в памяти:

```
float a, b; ...
if ( a == b ) cout << " равны" ;
else          cout << " не равны";           // Не рекомендуется!
if ( fabs(a - b) < 1e-6 ) cout << " равны" ;
else          cout << " не равны";           // Верно!
```

Значение величины, с которой сравнивается модуль разности, следует выбирать в зависимости от решаемой задачи и точности участвующих в выражении пере-

менных. Снизу эта величина ограничена для типа `float` определенной в заголовочном файле `<cfloat>` константой `FLT_EPSILON = 1.192092896e-07F` — это минимально возможное значение переменной типа `float` такое, что `1.0 + FLT_EPSILON != 1.0`.

Для присваивания какой-либо переменной двух различных значений в зависимости от выполнения условия лучше пользоваться не оператором `if`, а *тернарной условной операцией*, например:

```
if ( a < b ) c = x; else c = y;           // Нерационально
c = a < b ? x : y;                       // Рекомендуется
```

Тернарной эта операция называется потому, что у нее три операнда. Первый операнд представляет собой выражение, результат вычисления которого преобразуется в значение `true` или `false`. После знака вопроса через двоеточие записываются два выражения. Результат вычисления первого из них принимается за результат всей операции, если первый операнд имеет значение `true`. В противном случае результатом всей операции является результат вычисления второго выражения. Таким образом, переменной `c` будет присвоено значение либо переменной `x`, либо `y`.

Задача 2.3. Клавиши курсора

Написать программу, определяющую, какая из курсорных клавиш была нажата.

В составе библиотеки, унаследованной от языка C, есть функция `getch`, возвращающая код нажатой пользователем клавиши. При нажатии функциональной или курсорной клавиши эта функция возвращает 0 либо 0xE0 (в зависимости от компилятора), а ее повторный вызов позволяет получить расширенный код клавиши. В листинге 2.3 приведен текст программы.

Листинг 2.3. Клавиши курсора

```
#include <cstdio>
#include <conio.h>
using namespace std;
int main() {
    int key;
    printf("\n Нажмите одну из курсорных клавиш:\n");
    key = getch(); key = getch();
    switch ( key ) {
        case 77: printf( "стрелка вправо\n" ); break;
        case 75: printf( "стрелка влево\n" ); break;
        case 72: printf( "стрелка вверх\n" ); break;
        case 80: printf( "стрелка вниз\n" ); break;
        default: printf( "не стрелка\n" );
    }
}
```

Выражение, стоящее в скобках после ключевого слова `switch`, а также константные выражения в `case` должны быть целочисленного типа (они неявно приводятся к типу выражения в скобках). Если требуется выполнить одни и те же действия при нескольких различных значениях констант, метки перечисляются одна за другой:

```
case 77: case 75: case 72: case 80: printf( "стрелки" ); break;
```

Метки сами по себе не вызывают изменения порядка выполнения операторов, поэтому, если вы не хотите, чтобы управление было передано на первый оператор следующей ветви, необходимо после каждой ветви использовать оператор `break`.

СОВЕТ

Хотя наличие слова `default` не обязательно, рекомендуется всегда обрабатывать случай, когда значение выражения не совпадает ни с одной из констант. Это облегчает поиск ошибок при отладке программы.

Оператор `switch` предпочтительнее оператора `if` в тех случаях, если в программе требуется разветвить вычисления на количество направлений, большее двух, и выражение, по значению которого производится переход на ту или иную ветвь, является целочисленным. Часто это справедливо даже для двух ветвей, поскольку улучшает наглядность программы.

Циклы

Теоретический материал: с. 44–49, 237.

Цикл — это последовательность операторов, повторяемая многократно. В C++ три взаимозаменяемых оператора цикла — `while`, `do while` и `for`. При написании любого цикла надо иметь в виду, что в нем всегда явно или неявно присутствуют четыре элемента: начальные установки, тело цикла, модификация параметра цикла и выражение, проверяющее условие продолжения цикла (рис. 2.4). Начинающие чаще всего забывают про первое и(или) третье.

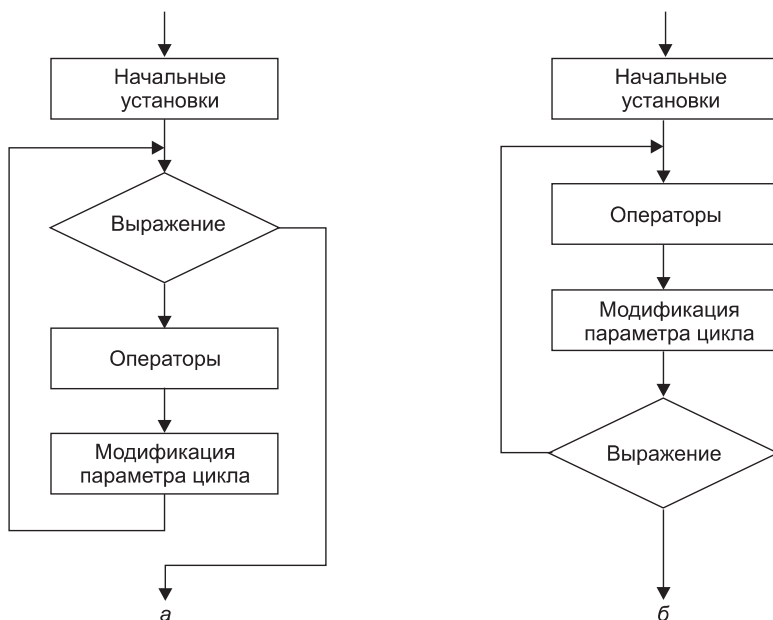


Рис. 2.4. Структурная схема операторов цикла: а — цикл с предусловием; б — цикл с постусловием

Задача 2.4. Таблица значений функции

Написать программу печати таблицы значений функции

$$y = \begin{cases} t, & x < 0; \\ t \cdot x, & 0 \leq x < 10; \\ 2 \cdot t, & x \geq 10. \end{cases}$$

для аргумента, изменяющегося в заданных пределах с заданным шагом. Если $t > 100$, значения функции должны выводиться в целочисленном виде.

Исходными данными являются начальное значение аргумента X_n , конечное значение аргумента X_k , шаг изменения аргумента dX и параметр t . Все величины — вещественные. Программа должна выводить таблицу, состоящую из двух столбцов: значений аргумента и соответствующих им значений функции.

В словесном виде алгоритм можно сформулировать так:

1. Ввести исходные данные.
2. Взять первое из значений аргумента.
3. Определить, какому интервалу из области определения функции принадлежит значение аргумента, и вычислить значение функции по соответствующей формуле.
4. Если $t > 100$, преобразовать значение y в целое.
5. Вывести строку таблицы.
6. Перейти к следующему значению аргумента.
7. Если оно не превышает конечное значение, повторить шаги 3–6, иначе закончить выполнение.

В каждый момент времени требуется хранить одно значение функции, поэтому для него достаточно завести одну переменную вещественного типа. Шаги 3–6 повторяются многократно, поэтому для их выполнения надо организовать цикл. В листинге 2.4 используется цикл `while`:

Листинг 2.4. Таблица значений функции (оператор `while`)

```
#include <stdio>
#include <cmath>
using namespace std;
int main() {
    double Xn, Xk, dX, t, y;
    printf( "Enter Xn, Xk, dX, t \n" );
    scanf( "%lf%lf%lf%lf",&Xn, &Xk, &dX, &t );
    printf( " ----- \n" );
    printf( "|      X      |      Y      |\n" );
    printf( " ----- \n" );
    double x = Xn;
    while ( x <= Xk ) {
        if ( x < 0 )          y = t;
        if ( x >= 0 && x < 10 ) y = t * x;
```

// Начальные установки

продолжение ➤

Листинг 2.4 (продолжение)

```

        if ( x >= 10 )          y = 2 * t;
        if ( t > 100 ) printf("|%9.2lf    |%9d    |\n", x, (int)y );
        else                 printf("|%9.2lf    |%9.2lf    |\n", x, y );
        x += dX;                                     // Модификация параметра цикла
    }
    printf( " ----- \n" );
}

```

В программу введена вспомогательная переменная x , которая последовательно принимает значения от X_n до X_k с шагом dX . Она определена непосредственно перед использованием. Это является хорошим стилем, поскольку снижает вероятность ошибок (например, таких, как использование неинициализированной переменной).

СОВЕТ

В общем случае надо стремиться к минимизации области видимости переменных.

В листинге 2.5 приведена та же программа с использованием оператора `for`.

Листинг 2.5. Таблица значений функции (оператор `for`)

```

#include <stdio>
#include <cmath>
using namespace std;
int main() {
    double Xn, Xk, dX, t, y;
    printf( "Enter Xn, Xk, dX, t \n" );
    scanf( "%lf%lf%lf%lf", &Xn, &Xk, &dX, &t );
    printf( " ----- \n" );
    printf( "|      X      |      Y      |\n" );
    printf( " ----- \n" );
    for ( double x = Xn; x <= Xk; x += dX ) {
        if ( x < 0 )          y = t;
        if ( x >= 0 && x < 10 ) y = t * x;
        if ( x >= 10 )          y = 2 * t;
        if ( t > 100 ) printf("|%9.2lf    |%9d    |\n", x, (int)y );
        else                 printf("|%9.2lf    |%9.2lf    |\n", x, y );
    }
    printf( " ----- \n" );
}

```

В листинге 2.4 область видимости x простирается от точки описания до конца программы, в листинге 2.5 областью ее видимости является только цикл, что предпочтительнее, поскольку переменная x вне цикла не требуется. Другим преимуществом второго варианта программы является то, что все управление циклом `for` сосредоточено в его заголовке. Это делает программу более читабельной.

Для преобразования в целое использована конструкция `(int)y`, унаследованная из языка C. Вообще-то для этого рекомендуется применять операцию преобразования типа `static_cast`, но нам не хотелось сразу пугать вас вот такой конструкцией:

```
printf( "%9.2lf    %9d    \\n", x, static_cast<int>(y) );
```

Выполните программу несколько раз, задавая различные значения исходных данных. С помощью ручного подсчета убедитесь в правильности вычислений.

Задача 2.5. Вычисление суммы ряда

Написать программу вычисления значения функции $\sin x$ (синус) с помощью бесконечного ряда Тейлора с точностью ϵ по формуле

$$y = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Этот ряд сходится при $|x| < \infty$. Точность достигается при $|R_n| < \epsilon$, где R_n —остаточный член ряда, который для данного ряда можно заменить величиной C_n очередного члена ряда, прибавляемого к сумме.

Общий алгоритм решения этой задачи достаточно прост: требуется задать начальное значение суммы ряда, а затем многократно вычислять очередной член ряда и добавлять его к ранее найденной сумме. Вычисления заканчиваются, когда абсолютная величина очередного члена ряда станет меньше заданной точности.

До выполнения программы предсказать, сколько членов ряда потребуется просуммировать, невозможно. В цикле такого рода есть опасность, что он никогда не завершится — как из-за возможных ошибок в вычислениях, так и из-за ограниченной области сходимости ряда (данный ряд сходится на всей числовой оси, но существуют ряды Тейлора, которые сходятся только для определенного интервала значений аргумента). Поэтому для надежности программы необходимо предусмотреть аварийный выход из цикла с печатью предупреждающего сообщения по достижении некоторого максимально допустимого количества итераций.

Прямое вычисление члена ряда по приведенной выше общей формуле, когда x возводится в степень, вычисляется факториал, а затем числитель делится на знаменатель, имеет два недостатка, которые делают этот способ непригодным. Первый недостаток — большая погрешность вычислений. При возведении в степень и вычислении факториала можно получить очень большие числа, при делении которых друг на друга произойдет потеря точности, поскольку количество значащих цифр, хранимых в ячейке памяти, ограничено. Кроме того, большие числа могут переполнить разрядную сетку. Второй недостаток — неэффективность вычислений: легко заметить, что в момент вычисления очередного члена ряда уже известен предыдущий, поэтому вычислять каждый член ряда «от печки» нерационально.

Для уменьшения количества выполняемых действий следует воспользоваться рекуррентной формулой получения последующего члена ряда через предыдущий

$C_{n+1} = C_n \times T$, где T — некоторый множитель. Подставив в эту формулу C_n и C_{n+1} , получим выражение для вычисления T :

$$T = \frac{C_{n+1}}{C_n} = \frac{x^2}{(2n+2)(2n+3)}.$$

Текст программы с комментариями приведен в листинге 2.6.

Листинг 2.6. Вычисление суммы бесконечного ряда

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    const int MaxIter = 500;           // максимально допустимое количество итераций
    double x, eps;
    cout << "\nВведите аргумент и точность:\n";
    cin >> x >> eps;
    bool done = true;                  // признак достижения точности
    double ch = x, y = ch;             // первый член ряда и начальное значение суммы
    for ( int n = 0; fabs( ch ) > eps; n++ ) {
        ch *= x * x / ( ( 2 * n + 2 ) / ( 2 * n + 3 ) ); // очередной член ряда
        y += ch;                          // добавление члена ряда к сумме
        if ( n > MaxIter ) {
            cout << "\nРяд расходится!";
            done = false; break; }
    }
    if ( done ) cout << "\nЗначение функции: " << y << " для x = " << x << endl;
}
```

Первый член ряда равен 1, поэтому, чтобы при первом проходе цикла значение второго члена вычислялось правильно, n должно быть равно 0. Максимально допустимое количество итераций удобно задать с помощью именованной константы. Для аварийного выхода из цикла применяется оператор `break` (см. Учебник, с. 50), который выполняет выход на первый после цикла оператор.

Поскольку выход и в случае аварийного, и в случае нормального завершения цикла происходит на один и тот же оператор, вводится булева переменная `done`, которая предотвращает печать значения функции, если точность вычислений не достигнута. Создание подобных переменных-«флагов», принимающих значение «истина» в случае успешного окончания вычислений и «ложь» в противном случае, является распространенным приемом программирования.

Измените программу так, чтобы она печатала не только значения аргумента и функции, но и количество просуммированных членов ряда, и выполните программу несколько раз для различных значений аргумента и точности. Выявите зависимость между этими величинами. В библиотеке есть функция `cosh(x)`, вычисляющая гиперболический косинус. Ее прототип находится в файле `<cmath>`. Измените программу так, чтобы она рядом с вычисленным с помощью ряда значением печатала результат применения стандартной функции. Сравните результаты вычислений.

Итоги

1. Чтобы получить максимальную читабельность и простоту структуры программы, надо правильно выбрать способ реализации ветвлений (с помощью `if`, `switch` или условной операции), а также наиболее подходящий оператор цикла.
2. Проверка вещественных величин на равенство опасна.
3. Выражение, стоящее в скобках после ключевого слова `switch`, и константные выражения в `case` должны быть целочисленного типа. После каждой ветви для передачи управления на конец оператора `switch` используется оператор `break`.
4. Рекомендуется всегда описывать в операторе `switch` ветвь `default`.
5. Чтобы избежать ошибок при программировании циклов, рекомендуется:
 - заключать в блок (фигурные скобки) тело циклов, если в них требуется выполнить более одного оператора;
 - проверять, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого начальные значения, а также возможно ли выполнение других операторов;
 - проверять, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;
 - если количество повторений цикла заранее не известно, предусматривать аварийный выход из цикла по достижении некоторого достаточно большого количества итераций.
6. Операторы цикла в языке C++ взаимозаменяемы, но можно привести некоторые рекомендации по выбору наилучшего в каждом конкретном случае:
 - Оператор `do while` используют, когда цикл требуется обязательно выполнить хотя бы один раз — например, при вводе данных.
 - Оператор `while` удобнее в тех случаях, когда либо число итераций заранее неизвестно, либо очевидных параметров цикла нет, либо модификацию параметров удобнее записывать где-то в произвольном месте тела цикла.
 - Оператор `for` предпочтительнее в большинстве остальных случаев. Однозначно — для организации циклов со счетчиками, то есть с целочисленными переменными, которые изменяют свое значение при каждом проходе цикла регулярным образом (например, увеличиваются на 1).

Задания

Вычислить и вывести на экран в виде таблицы значения функции, заданной с помощью ряда Тейлора, на интервале от $x_{\text{нач}}$ до $x_{\text{кон}}$ с шагом dx с точностью ϵ . Таблицу снабдить заголовком и шапкой. Каждая строка должна содержать значение аргумента, значение функции и количество просуммированных членов ряда.

1.
$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \dots$$

2. $x - \frac{x^2}{\sqrt{2}} + \frac{x^3}{\sqrt{3}} - \frac{x^4}{2} + \frac{x^5}{\sqrt{5}} - \dots$
3. $1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \frac{x^5}{5!} + \dots$
4. $1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$
5. $1 - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$
6. $x - \frac{1}{2} \cdot \frac{x^3}{3} + \frac{1 \cdot 3}{2 \cdot 4} \cdot \frac{x^5}{5} - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \cdot \frac{x^7}{7} + \dots$
7. $1 - 2x^2 + 3x^3 - 4x^4 + 5x^5 - 6x^6 + \dots$
8. $x - \frac{x^2}{2^2} + \frac{x^3}{3^2} - \frac{x^4}{4^2} + \frac{x^5}{5^2} - \frac{x^6}{6^2} + \dots$
9. $1 - 2x + \frac{3x^2}{2!} - \frac{4x^3}{3!} + \frac{5x^4}{4!} - \frac{6x^5}{5!} + \dots$
10. $\frac{x}{1 \cdot 2^2} - \frac{x^2}{2 \cdot 3^2} + \frac{x^3}{3 \cdot 4^2} - \frac{x^4}{4 \cdot 5^2} + \frac{x^5}{5 \cdot 6^2} - \dots$
11. $1 - \frac{1}{2} \cdot x^2 + \frac{1 \cdot 3}{2 \cdot 4} \cdot x^4 - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \cdot x^6 + \dots$
12. $x - \frac{2^2 x^2}{5} + \frac{2^3 x^3}{10} - \frac{2^4 x^4}{17} + \frac{2^5 x^5}{26} - \dots$
13. $\frac{x}{2 \cdot 1} - \frac{x^2}{2^2 \cdot 2} + \frac{x^3}{2^3 \cdot 3} - \frac{x^4}{2^4 \cdot 4} + \frac{x^5}{2^5 \cdot 5} - \dots$
14. $\frac{1}{3} - \frac{2x}{2 \cdot 3^2} + \frac{3x^2}{2^2 \cdot 3^3} - \frac{4x^3}{2^3 \cdot 3^4} + \frac{5x^4}{2^4 \cdot 3^5} - \dots$
15. $\frac{x}{1 \cdot 2} - \frac{2x^2}{2 \cdot 2 \cdot 3} + \frac{3x^3}{2^2 \cdot 3 \cdot 4} - \frac{4x^4}{2^3 \cdot 4 \cdot 5} + \dots$
16. $1 - 4x + 9x^2 - 16x^3 + 25x^4 - 36x^5 + \dots$
17. $\frac{2x^2}{3} - \frac{4x^3}{8} + \frac{8x^4}{15} - \frac{16x^5}{24} + \dots$
18. $\frac{1}{3^{\frac{1}{2}}} - \frac{x}{3^{\frac{1}{3}}} + \frac{x^2}{3^{\frac{1}{4}}} - \frac{x^3}{3^{\frac{1}{5}}} + \dots$

$$19. \quad x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \dots$$

$$20. \quad 1 - \frac{x^2}{2} + \frac{x^4}{4} - \frac{x^6}{6} + \frac{x^8}{8} - \dots$$

Семинар 3. Одномерные массивы и указатели

Теоретический материал: с. 51–61.

В случае простых переменных каждой области памяти для хранения одной величины соответствует свое имя. Если требуется работать с группой величин одного типа, их располагают в памяти последовательно и дают им общее имя, а различают по порядковому номеру. Такая последовательность однотипных величин называется *массивом*. Массивы, как и любые другие объекты, можно размещать либо с помощью операторов описания, либо динамически с помощью операций выделения памяти.

Оператор описания массива задает его тип, имя и *размерность* (количество элементов) в квадратных скобках, например, `int a[9]`. Все инструкции выделения памяти формирует компилятор до выполнения программы, поэтому размерность массива может быть только константой или константным выражением. При описании массива можно *инициализировать* (присвоить его элементам начальные значения):

```
int a[9] = { 1, 1, 2, 2, 5, 100 };
```

Если инициализирующих значений меньше, чем элементов в массиве, остаток массива обнуляется, если больше — лишние значения не используются.

Элементы массива нумеруются с нуля, поэтому максимальный *индекс* (номер) элемента всегда на единицу меньше размерности. *Автоматический контроль выхода индекса за границы массива не выполняется*, поэтому программист должен следить за этим самостоятельно. Для описанного выше массива `a` элементы имеют номера от 0 до 8. Номер элемента задается после его имени в квадратных скобках: `a[0]`, `a[3]`.

Если до начала работы программы неизвестно, сколько в массиве элементов, следует использовать *динамические массивы*. Память под них выделяют с помощью операции `new` или функции `malloc` в хипе (куче) во время выполнения программы. Адрес начала массива хранится в переменной, называемой *указателем*. Например:

```
int n = 9; // 1
int *a = new int[n]; // 2
double *b = ( double * ) malloc( n * sizeof ( double ) ); // 3
```

В *строке 2* описан указатель на целую величину, которому присваивается адрес начала непрерывной области динамической памяти, выделенной с помощью операции `new`. Выделяется столько памяти, сколько необходимо для хранения `n` величин

типа `int`. Величина `n` может быть переменной. В строке 3 для выделения памяти под `n` элементов типа `double` используется функция `malloc`, унаследованная из библиотеки `C`. Этот способ устарел, мы им пользоваться не будем.

ВНИМАНИЕ

Обнуления памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.

Обращение к элементу динамического массива выполняется так же, как и к элементу обычного — например, `a[3]`. Можно (но не нужно) использовать и другой способ, более экзотичный — `*(a + 3)`. Здесь мы явно задаем те же действия, что выполняются при обычном обращении к элементу массива. Рассмотрим их подробнее. В переменной-указателе `a` хранится адрес начала массива¹. Для получения адреса третьего элемента к этому адресу прибавляется смещение 3. Операция сложения с константой для указателей учитывает размер адресуемых элементов, то есть на самом деле индекс умножается на длину элемента массива: `a + 3 * sizeof(int)`. Затем с помощью операции `*` (разадресации) выполняется выборка значения из указанной области памяти.

Если динамический массив в какой-то момент становится ненужным и мы планируем использовать эту память повторно, необходимо освободить ее с помощью операции `delete[]`, при этом размерность массива не указывается, например:

```
delete [] a;
```

ВНИМАНИЕ

Квадратные скобки в операции `delete []` при освобождении памяти из-под массива обязательны. Их отсутствие может привести к неопределенному поведению программы. Память, выделенную с помощью `malloc`, следует освобождать посредством функции `free` (см. Учебник, с. 55, с. 422).

Таким образом, время жизни динамического массива, как и любой динамической переменной, — с момента выделения памяти до момента ее освобождения. Область действия зависит от места описания указателя, через который выполняется работа с массивом. Область действия и время жизни указателей подчиняются общим правилам, рассмотренным на первом семинаре. Как вы помните, локальная переменная при выходе из блока, в котором она описана, «теряется». Если эта переменная является указателем и в ней хранится адрес выделенной динамической памяти, при выходе из блока эта память перестает быть доступной, однако не помечается как свободная, поэтому не может быть использована в дальнейшем. Это называется *утечкой памяти* и является распространенной ошибкой:

```
{   int n; cin >> n;
    int *pmas = new int[n];    ...           // пример утечки памяти:
}                               // после выхода из блока указатель pmas недоступен
```

¹ Имя «обычного» массива также является указателем на его первый элемент, только константным (то есть ему нельзя присвоить новое значение).

Задача 3.1. Количество элементов между минимумом и максимумом

Написать программу, которая для целочисленного массива из 100 элементов определяет, сколько положительных элементов располагается между его максимальным и минимальными элементами.

Запишем алгоритм в общем виде.

- 1. Определить, где в массиве расположены его максимальный и минимальный элементы, то есть найти их индексы.
- 2. Завести счетчик положительных элементов.
- 3. Просмотреть все элементы, расположенные между найденными индексами. Если элемент массива больше нуля, увеличить счетчик на единицу.

Перед написанием программы полезно составить тестовые примеры, чтобы более наглядно представить себе алгоритм решения задачи. Ниже представлен пример массива из 10 чисел и обозначены искомые величины:

6	−8	15	9	−1	3	5	−10	12	2
		макс	+		+	+	мин		

Ясно, что порядок расположения элементов в массиве заранее не известен, и сначала может следовать как максимальный, так и минимальный элемент, кроме того, они могут совпадать. Поэтому, прежде чем просматривать массив в поисках количества положительных элементов, требуется определить, какой из этих индексов больше. Запишем уточненный алгоритм:

- 1. Определить, где в массиве расположены его максимум и минимум:
 - Задать начальные значения для индексов максимального и минимального элементов (например, равные нулю, но можно использовать любые другие значения индекса, не выходящие за границу массива).
 - Просмотреть массив, поочередно сравнивая каждый его элемент с ранее найденными максимумом и минимумом. Если очередной элемент больше ранее найденного максимума, принять этот элемент за новый максимум (то есть запомнить его индекс). Если очередной элемент меньше ранее найденного минимума, принять этот элемент за новый минимум.
 - 2. Определить границы просмотра массива для поиска положительных элементов, находящихся между его максимальным и минимальными элементами:
 - Если максимум расположен в массиве раньше, чем минимум, принять левую границу просмотра равной индексу максимума, иначе — индексу минимума.
 - Если максимум расположен в массиве раньше, чем минимум, принять правую границу просмотра равной индексу минимума, иначе — индексу максимума.
 - 3. Обнулить счетчик положительных элементов. Просмотреть массив в указанном диапазоне. Если очередной элемент больше нуля, увеличить счетчик на единицу.
- Для простоты элементы массива задаются в листинге 3.1 путем инициализации.

Листинг 3.1. Количество элементов между минимумом и максимумом – вариант 1

```
#include <iostream>
using namespace std;
int main() {
    const int n = 10;
    int a[n] = { 1, 3, -5, 1, -2, 1, -1, 3, 8, 4 };
    int imax = 0, imin = 0, count = 0;
    for ( int i = imax = imin = 0; i < n; i++ ) {
        if ( a[i] > a[imax] ) imax = i;
        if ( a[i] < a[imin] ) imin = i;
    }
    cout << "\n\t max= " << a[imax] << " min= " << a[imin];           // отладка
    int ibeg = imax < imin ? imax : imin;
    int iend = imax < imin ? imin : imax;
    cout << "\n\t ibeg= " << ibeg << " iend= " << iend;               // отладка
    for ( int i = ibeg + 1; i < iend; i++ )
        if ( a[i] > 0 ) count++;
    cout << " Количество положительных: " << count << endl;
}
```

В программе использована управляющая последовательность `\t`, которая задает отступ при выводе на следующую позицию табуляции.

СОВЕТ

После нахождения каждой величины вставлена отладочная печать. Мы рекомендуем не пренебрегать таким способом отладки и стараться сделать эту печать содержащей необходимые пояснения и хорошо форматированной.

Массив просматривается, начиная с элемента, следующего за максимальным (минимальным), заканчивая элементом, предшествующим минимальному (максимальному). Индексы границ просмотра хранятся в переменных `ibeg` и `iend`. Для определения их значений используется тернарная условная операция (см. с. 31).

Можно поступить по-другому: просматривать массив всегда от максимума к минимуму, а индекс увеличивать или уменьшать в зависимости от их взаимного расположения. В листинге 3.2 направление просмотра (приращение индекса) хранится в переменной `d`. Если массив просматривается «слева направо», она равна 1, иначе — -1. Обратите внимание на условие продолжения этого цикла.

Листинг 3.2. Количество элементов между минимумом и максимумом – вариант 2

```
#include <iostream>
using namespace std;
int main() {
    const int n = 10;
    int a[n];
    cout << " Введите " << n << " целых чисел: " << endl;
    for ( int i = 0; i < n; i++ ) cin >> a[i];
    for ( int i = 0; i < n; i++ ) cout << a[i] << " ";
}
```

продолжение ➤

Листинг 3.2 (продолжение)

```

int imax = 0, imin = 0, count = 0;
for ( int i = imax = imin = 0; i < n; i++ ) {
    if ( a[i] > a[imax] ) imax = i;
    if ( a[i] < a[imin] ) imin = i;
}
int d = imax < imin ? 1 : imax > imin ? -1 : 0;
for ( int i = imax + d; i != imin; i += d )
    if ( a[i] > 0 ) count++;
cout << " Количество положительных: " << count << endl;
}

```

Ввод массива выполняется с клавиатуры. Напоминаем, что в этом случае желательно для проверки вывести введенные значения на печать. Разница между приведенными способами решения несущественна, но первый вариант более «прозрачен», поэтому он, на наш взгляд, предпочтительнее.

Тестовых примеров для этой задачи должно быть по крайней мере три: когда минимум расположен левее максимума, правее максимума и когда они совпадают (это имеет место, если в массиве все элементы имеют одно и то же значение. Кстати, в листинге 3.2 этот случай корректно обрабатывается благодаря значению $d = 0$). Желательно также проверить, как работает программа, если $a[\text{imin}]$ и $a[\text{imax}]$ расположены рядом, а также в начале и в конце массива (граничные случаи). Элементы массива нужно задать и положительные и отрицательные.

Часто бывает, что точное количество элементов в исходном массиве не задано, но известно, что оно не может превышать некоторое конкретное значение. В этом случае память под массив выделяется «по максимуму», а затем заполняется только ее часть. Фактическое количество введенных элементов запоминается в переменной, которая затем участвует в организации циклов по массиву, задавая его верхнюю границу. Этот подход является весьма распространенным, поэтому ниже приводится фрагмент программы для его иллюстрации:

```

const int n = 1000;
int a[n]; // Выделение памяти "с запасом"
int kol_a;
cout << "Введите количество элементов:\n"; cin >> kol_a;
if ( kol_a > n ) { cout << " Превышение размеров массива!\n"; return 1; }
cout << "Введите элементы:\n";
for ( int i = 0; i < kol_a; i++ ) cin >> a[i];
...

```

Несмотря на то что значение n определяется «с запасом», надо обязательно проверять, не запрашивается ли большее количество элементов, чем возможно. Привычка к проверке подобных, казалось бы, маловероятных случаев позволит вам создавать более надежные программы, а нет ничего более важного для программы, чем надежность. Впрочем, и для человека это качество совсем не лишнее.

Задача 3.2. Сумма элементов правее последнего отрицательного

Написать программу, которая для вещественного массива из n элементов определяет сумму его элементов, расположенных правее последнего отрицательного элемента.

Здесь размерность массива задана переменной величиной. Предполагается, что она станет известна при выполнении программы до того, как мы будем вводить сами элементы. Тогда мы сможем выделить в хипе непрерывный участок нужного размера, а потом заполнить его вводимыми значениями. Если же стоит задача вводить заранее неизвестное количество чисел до тех пор, пока не будет введен какой-либо признак окончания ввода, то заранее выделить память не удастся и придется воспользоваться *динамическими структурами данных*, например списком. Мы рассмотрим их на девятом семинаре, а пока остановимся на предположении, что количество элементов вводится с клавиатуры до начала ввода самих элементов.

По аналогии с предыдущей задачей приходит в голову такой алгоритм решения задачи: просмотрев массив с начала до конца, найти номер последнего отрицательного элемента, а затем организовать цикл суммирования всех элементов, расположенных правее него. В листинге 3.3 приведена построенная по этому алгоритму программа (сразу же признаемся, что она далека от совершенства).

Листинг 3.3. Сумма элементов правее последнего отрицательного (не работает)

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout << " Введите количество элементов "; cin >> n;
    double *a = new double [n]; // 1
    cout << " Введите элементы массива ";
    for ( int i = 0; i < n; i++ ) cin >> a[i];
    for ( int i = 0; i < n; i++ ) cout << a[i] << " "; // 2
    int ineg;
    for ( int i = 0; i < n; i++ ) if ( a[i] < 0 ) ineg = i; // 3
    double sum = 0;
    for ( int i = ineg + 1; i < n; i++ ) sum += a[i]; // 4
    cout << " Сумма " << sum;
}
```

Поскольку количество элементов заранее не задано, память под массив выделяется в *операторе 1* на этапе выполнения программы с помощью операции `new`. Выделяется столько памяти, сколько необходимо для хранения n элементов вещественного типа, и адрес начала этого участка заносится в указатель `a`.

Номер последнего отрицательного элемента массива формируется в переменной `ineg`. При просмотре массива с помощью *оператора 3* в эту переменную последо-

вательно записываются номера всех отрицательных элементов массива, таким образом, после выхода из цикла в ней остается номер самого последнего.

С целью оптимизации программы может возникнуть мысль объединить цикл нахождения этого номера с циклами ввода и контрольного вывода элементов, но мы советуем так не делать, потому что ввод данных, их вывод и анализ — разные по смыслу действия, и объединение их в одном цикле не прибавит программе ясности. После отладки программы контрольный вывод (*оператор 2*) можно удалить. В последующих примерах мы для экономии места будем позволять себе его опускать, но это не значит, что вы должны поступать так же!

Теперь перейдем к критическому анализу листинга 3.3. Для массивов, содержащих отрицательные элементы, программа работает верно, но при их отсутствии, как правило, завершается аварийно. Это связано с тем, что, если в массиве нет ни одного отрицательного элемента, в *операторе 3* переменной `ineg` значение явным образом не присваивается. *Неинициализированные локальные переменные* до первого присваивания имеют неопределенное значение, зависящее от среды выполнения. Например, в Visual Studio 2005 неинициализированная переменная типа `int` имеет значение `0xCCCCCCC`. Поэтому в операторе `for` (*оператор 4*) произойдет аварийное завершение программы.

Если задуматься еще глубже, обнаружится, что требуется специальная обработка и в том случае, когда последний элемент массива отрицательный. В этом случае сумма также не существует.

Для выполнения необходимых проверок переменной `ineg` следует присвоить начальное значение, не входящее в множество допустимых индексов массива (например, `-1`). В листинге 3.4 представлен исправленный вариант программы.

Листинг 3.4. Сумма элементов правее последнего отрицательного

```
#include <iostream>
using namespace std;
int main() {
    int n;
    cout << " Введите количество элементов: "; cin >> n;
    double *a = new double [n]; // 1
    cout << " Введите элементы массива: ";
    for ( int i = 0; i < n; i++ ) cin >> a[i];
    for ( int i = 0; i < n; i++ ) cout << a[i] << " "; // 2
    int ineg = -1;
    for ( int i = 0; i < n; i++ ) if ( a[i] < 0 ) ineg = i; // 3
    if ( ineg == -1 ) cout << "\n0трицательных элементов нет!\n";
    else if ( ineg == n - 1 ) cout << "\n0трицательный элемент - последний!\n";
    else {
        double sum = 0;
        for ( int i = ineg + 1; i < n; i++ ) sum += a[i]; // 4
        cout << " Сумма отрицательных: " << sum;
    }
}
```

Можно предложить более рациональное на первый взгляд решение задачи: просматривать массив в обратном порядке, суммируя элементы, и завершить цикл, как только встретится отрицательный элемент:

```
...
bool flag_neg = false;
double sum = 0;
for ( int i = n - 1; i >= 0; i-- ) {
    if ( a[i] < 0 ) { flag_neg = true; break; }
    sum += a[i];
}
if ( flag_neg ) cout << "\nСумма " << sum;
else          cout << "\nОтрицательных элементов нет";
```

В этой программе каждый элемент массива анализируется не более одного раза, а ненужные элементы не просматриваются вообще. Казалось бы, мы позаботились об эффективности, но на самом деле результат может оказаться даже хуже, чем в предыдущем варианте! Это связано с тем, что в современных процессорах для оптимизации применяется опережающее чтение в быстросействующую память (*кэш*), поэтому при «естественном» порядке просмотра массива следующие элементы окажутся в кэше с большей вероятностью.

Для тестирования этой программы необходимо ввести несколько вариантов исходных данных — когда массив содержит один, несколько или не содержит ни одного отрицательного элемента, а также когда он заканчивается отрицательным элементом.

Мы рассмотрели примеры задач поиска и вычислений в массиве. Другой распространенной задачей является *сортировка массива*, то есть упорядочивание его элементов в соответствии с каким-либо критерием — чаще всего по возрастанию или убыванию элементов. Существует множество методов сортировки, различающихся по поведению, быстросействию, объему памяти и ограничениям, накладываемым на исходные данные. В Учебнике (с. 59) рассмотрен один из наиболее простых методов — сортировка выбором. Он характеризуется квадратичной зависимостью времени сортировки t от количества элементов: $t = a \cdot N^2 + b \cdot N \cdot \lg N$, где a и b — константы, зависящие от программной реализации алгоритма. Иными словами, для сортировки массив требуется просмотреть примерно N раз.

Существуют алгоритмы и с лучшими характеристиками, самый известный из которых называется *быстрой сортировкой*. Для него зависимость имеет вид $t = a \cdot N \cdot \lg N + b \cdot N$. Давайте посмотрим, за счет чего достигается экономия. Кстати, наименее эффективным является любимый студентами метод пузырька.

Задача 3.3. Быстрая сортировка массива

Написать программу, которая упорядочивает вещественный массив по возрастанию значений его элементов методом быстрой сортировки.

Идея алгоритма состоит в следующем. Применим к массиву так называемую *процедуру разделения* относительно среднего элемента. В качестве «среднего» можно

выбрать любой элемент массива, но для наглядности мы будем выбирать, по возможности, средний по своему номеру элемент.

Процедура разделения делит массив на две части. В левую помещаются элементы, меньшие элемента, выбранного в качестве среднего, а в правой — большие. Это достигается путем просмотра массива попеременно с обоих концов, при этом каждый элемент сравнивается с выбранным средним, и элементы, находящиеся в «неподходящей» части, меняются местами. После завершения процедуры разделения средний элемент оказывается на своем окончательном месте, то есть его «судьба» определена, и мы можем про него забыть. Далее процедуру разделения необходимо повторить отдельно для левой и правой части: в каждой части выбирается среднее, относительно которого она делится на две, и так далее.

Понятно, что одновременно процедура не может заниматься и левой, и правой частями, поэтому необходимо каким-то образом запомнить запрос на обработку одной из двух частей (например, правой) и заняться оставшейся частью (например, левой). Так продолжается до тех пор, пока не окажется, что очередная обрабатываемая часть содержит ровно один элемент. Тогда нужно вернуться к последнему из необработанных запросов, применить к нему все ту же процедуру разделения, и так далее. В конце концов массив окажется полностью упорядочен.

Для хранения границ еще не упорядоченных частей массива более всего подходит структура данных, называемая *стеком*. Мы будем рассматривать «настоящие» стеки на девятом семинаре, а пока просто уловите идею. Представьте себе узкий коридор-тупик. Человек, который имел несчастье туда войти первым, сможет покинуть его только самым последним. Это и есть стек — «первым пришел, последним ушел».

В приведенной ниже программе стек реализуется в виде двух массивов, `stackr` и `stackl`, и одной переменной `sp`, используемой как «указатель» на вершину стека (она хранит номер последнего заполненного элемента массива). Для этого алгоритма количество элементов в стеке не может превышать `n`, поэтому размер массивов задан равным именно этой величине. При занесении в стек переменная `sp` увеличивается на единицу, а при выборке — уменьшается. Программа приведена в листинге 3.5¹.

Листинг 3.5. Быстрая сортировка массива

```
#include <iostream>
#include <cmath>
using namespace std;
int main() {
    const int n = 20;
    float arr[n];
    int *stackl = new int [n], *stackr = new int [n];
    cout << "Введите элементы массива: ";
    for ( int i = 0; i < n; i++ ) cin >> arr[i];
    ////////////////////////////////////////////////// Сортировка //////////////////////////////////////
    int sp = 1; stackl[sp] = 0; stackr[sp] = n - 1; // 1
```

¹ Существует более красивая реализация метода быстрой сортировки, основанная на рекурсии. Мы рассмотрим ее на седьмом семинаре.


```

while ( sp > 0 ) {           // Выборка из стека последнего запроса      // 2
    int left = stackl[sp];    // 3
    int right = stackr[sp];   // 4
    sp--;                    // 5
    while ( left < right ) {   // 6
        // Разделение { arr[left] .. arr[right] }
        int i = left, j = right;    // 7
        float middle = arr[( left + right ) / 2]; // 8
        while ( i < j ) {           // 9
            while ( arr[i] < middle ) i++; // 10
            while ( middle < arr[j] ) j--; // 11
            if ( i <= j ) {
                float temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
                i++; j--;
            }
        }
        if ( i < right ) { // Запись в стек запроса из правой части // 12
            sp++;
            stackl[sp] = i;
            stackr[sp] = right;
        }
        right = j; // Теперь left и right ограничивают левую часть // 13
    }
}
for ( int i = 0; i < n; i++ ) cout << arr[i] << ' ';
}

```

На каждом шаге сортируется один фрагмент массива. Левая и правая границы фрагмента хранятся в переменных `left` и `right`. Сначала фрагмент устанавливается размером с весь массив (*строка 1*). В *операторе 8* выбирается «средний» элемент.

Для продвижения по массиву слева направо используется переменная `i` (*цикл 10*), справа налево — переменная `j` (*цикл 11*). Их начальные значения устанавливаются в *операторе 7*. После того, как оба счетчика «сойдутся» где-то в средней части массива, происходит выход из *цикла 9* на *оператор 12*, в котором в стек заносятся границы правой части фрагмента. В *операторе 13* устанавливаются новые границы левой части для сортировки на следующем шаге.

Если фрагмент уже настолько мал, что сортировать его не требуется, происходит выход из *цикла 6*, после чего выполняется выборка из стека границ еще не отсортированного фрагмента (*операторы 3, 4*). Если стек пуст, происходит выход из главного *цикла 2*. Массив отсортирован.

Добавьте в программу подсчет количества итераций основного цикла. Прогоните программу несколько раз для массивов с большим количеством элементов¹ и сравните с аналогичной программой, реализующей метод выбора. Сделайте выводы.

¹ Для удобства можно заменить ввод с клавиатуры на генерацию случайной последовательности с помощью функций `srand` и `rand` (см. Учебник, с. 433, 435).

Метод быстрой сортировки был предложен Ч. Хоаром. Впоследствии дотошный исследователь методов сортировки Д. Кнут (D. Knuth) установил, что размер стека может быть уменьшен до величины $\log_2 n$, если после каждого появления двух частей, подлежащих дальнейшей обработке, более длинную часть откладывать на потом (помещать в стек), а более короткую обрабатывать немедленно.

Быстрая сортировка является одним из лучших методов упорядочивания, однако существует целый ряд алгоритмов, которые предпочтительнее применять для данных, отвечающих определенным критериям. Советуем вам на досуге ознакомиться с этими алгоритмами. Выбор наиболее подходящего для каждого случая метода сортировки данных — показатель класса программиста.

Итоги

1. Размерность не-динамического массива может быть только константой или константным выражением. Рекомендуется задавать размерность с помощью именованной константы.
2. Элементы массивов нумеруются с нуля, поэтому максимальный номер элемента всегда на единицу меньше размерности.
3. Автоматический контроль выхода индекса за границы массива не производится, поэтому программист должен следить за этим самостоятельно.
4. Указатель — это переменная, в которой хранится адрес области памяти. Имя массива является указателем на его нулевой элемент.
5. Обнуления динамической памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.
6. Освобождение памяти, выделенной посредством `new[]`, выполняется с помощью операции `delete[]`.
7. Перед выходом локального указателя из области его действия необходимо освободить связанную с ним динамическую память.
8. Если количество элементов, которые должны быть введены в программу, известно до ее выполнения, определяйте массив в операторе описания переменных (причем лучше как локальную переменную, чем как глобальную); если количество можно задать во время выполнения программы, но до ввода элементов, создавайте динамический массив; если нет, используйте линейный список или другую динамическую структуру.
9. Алгоритмы сортировки массивов различаются по быстродействию, занимаемой памяти и области применения.

Задания

Исходные данные для всех вариантов — n вещественных величин. При написании программ можно использовать как динамические, так и не-динамические массивы. Размерность последних задавать именованной константой.

Вариант 1

1. Найти сумму отрицательных элементов массива.
2. Найти произведение элементов массива, расположенных между максимальным и минимальным элементами.
3. Упорядочить элементы массива по возрастанию.

Вариант 2

1. Найти сумму положительных элементов массива.
2. Найти произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.
3. Упорядочить элементы массива по убыванию.

Вариант 3

1. Найти произведение элементов массива с четными номерами.
2. Найти сумму элементов массива, расположенных между первым и последним нулевыми элементами.
3. Преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом — все отрицательные (элементы, равные 0, считать положительными).

Вариант 4

1. Найти сумму элементов массива с нечетными номерами.
2. Найти сумму элементов массива, расположенных между первым и последним отрицательными элементами.
3. Сжать массив, удалив из него все элементы, модуль которых не превышает 1. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 5

1. Найти максимальный элемент массива.
2. Найти сумму элементов массива, расположенных до последнего положительного элемента.
3. Сжать массив, удалив из него все элементы, модуль которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 6

1. Найти минимальный элемент массива.
2. Найти сумму элементов массива, расположенных между первым и последним положительными элементами.
3. Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.

Вариант 7

1. Найти номер максимального элемента массива.
2. Найти произведение элементов массива, расположенных между первым и вторым нулевыми элементами.
3. Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в нечетных позициях, а во второй половине — элементы, стоявшие в четных позициях.

Вариант 8

1. Найти номер минимального элемента массива.
2. Найти сумму элементов массива, расположенных между первым и вторым отрицательными элементами.
3. Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает 1, а потом — все остальные.

Вариант 9

1. Найти максимальный по модулю элемент массива.
2. Найти сумму элементов массива, расположенных между первым и вторым положительными элементами.
3. Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.

Вариант 10

1. Найти минимальный по модулю элемент массива.
2. Найти сумму модулей элементов массива, расположенных после первого элемента, равного нулю.
3. Преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в четных позициях, а во второй половине — элементы, стоявшие в нечетных позициях.

Вариант 11

1. Найти номер минимального по модулю элемента массива.
2. Найти сумму модулей элементов массива, расположенных после первого отрицательного элемента.
3. Сжать массив, удалив из него все элементы, величина которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 12

1. Найти номер максимального по модулю элемента массива.
2. Найти сумму элементов массива, расположенных после первого положительного элемента.
3. Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых лежит в интервале $[a, b]$, а потом — все остальные.

Вариант 13

1. Найти количество элементов массива, лежащих в диапазоне от A до B .
2. Найти сумму элементов массива, расположенных после максимального элемента.
3. Упорядочить элементы массива по убыванию модулей элементов.

Вариант 14

1. Найти количество элементов массива, равных 0.
2. Найти сумму элементов массива, расположенных после минимального элемента.
3. Упорядочить элементы массива по возрастанию модулей элементов.

Вариант 15

1. Найти количество элементов массива, больших C .
2. Найти произведение элементов массива, расположенных после максимального по модулю элемента.
3. Преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом — все положительные (элементы, равные 0, считать положительными).

Вариант 16

1. Найти количество отрицательных элементов массива.
2. Найти сумму модулей элементов массива, расположенных после минимального по модулю элемента.
3. Заменить все отрицательные элементы массива их квадратами и упорядочить элементы массива по возрастанию.

Вариант 17

1. Найти количество положительных элементов массива.
2. Найти сумму элементов массива, расположенных после последнего элемента, равного нулю.
3. Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых не превышает 1, а потом — все остальные.

Вариант 18

1. Найти количество элементов массива, меньших C .
2. Найти сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.
3. Преобразовать массив таким образом, чтобы сначала располагались все элементы, отличающиеся от максимального не более чем на 20%, а потом — все остальные.

Вариант 19

1. Найти произведение отрицательных элементов массива.
2. Найти сумму положительных элементов массива, расположенных до максимального элемента.
3. Изменить порядок следования элементов в массиве на обратный.

Вариант 20

1. Найти произведение положительных элементов массива.
2. Найти сумму элементов массива, расположенных до минимального элемента.
3. Упорядочить по возрастанию отдельно элементы, стоящие на четных местах, и элементы, стоящие на нечетных местах.

Семинар 4. Двумерные массивы

Теоретический материал: с. 61–63.

Двумерный массив представляется в C++ как массив, состоящий из массивов. Вторая размерность задается в отдельных квадратных скобках. Если массив определяется с помощью *операторов описания*, обе его размерности должны быть константами или константными выражениями, поскольку инструкции по выделению памяти формируются компилятором до выполнения программы:

```
int a[3][5];    // Целочисленная матрица из 3 строк и 5 столбцов
```

Массив хранится по строкам в непрерывной области памяти:

```
a00 a01 a02 a03 a04 a10 a11 a12 a13 a14 a20 a21 a22 a23 a24  
| --- 0-я строка -- | --- 1-я строка -- | --- 2-я строка -- |
```

Строки двумерного массива ничем не разделены, то есть прямоугольной матрицей он является только в нашем воображении. В памяти сначала располагается одномерный массив `a[0]` — нулевая строка массива `a`, затем — массив `a[1]`, являющийся первой строкой массива `a`, и т.д. Количество элементов в каждом из этих массивов равно длине строки, то есть количеству столбцов в матрице. При просмотре массива от начала в первую очередь изменяется правый индекс (номер столбца).

Для доступа к отдельному элементу массива применяется конструкция вида `a[i][j]`, где `i` (номер строки) и `j` (номер столбца) — выражения целочисленного типа. *Каждый индекс может изменяться от 0 до значения соответствующей размерности, уменьшенной на единицу.*

ВНИМАНИЕ

Первый индекс всегда воспринимается как номер строки, второй — как номер столбца, независимо от имени переменной.

Можно (но не нужно) обратиться к элементу массива и другими способами: `*(a + i) + j` или `*(a[i] + j)`. Они приведены для лучшего понимания механизма индексации, поскольку здесь в явном виде записаны те же действия, которые генерируются компилятором при обычном обращении к массиву.

Рассмотрим их подробнее. Допустим, требуется обратиться к элементу, расположенному на пересечении второй строки и третьего столбца — `a[2][3]`. Как и для одномерных массивов, имя массива `a` является константным указателем на начало массива. Здесь это массив, состоящий из трех массивов. Сначала требуется обратиться ко второй строке массива, то есть одномерному массиву `a[2]`. Для этого надо прибавить к адресу начала массива смещение, равное номеру строки, и выполнить разадресацию: `*(a + 2)`. Напомним, что при сложении указателя с константой учитывается длина адресуемого элемента, поэтому на самом деле число 2 умножается на длину элемента, то есть `2 * (5 * sizeof(int))`, поскольку элементом является строка, состоящая из 5 элементов типа `int`.

Далее требуется обратиться к третьему элементу полученного массива. Для получения его адреса опять применяется сложение указателя с константой 3 (на самом деле прибавляется `3 * sizeof(int)`), а затем применяется операция разыменования для получения значения элемента: `*(*(a + 2) + 3)`.

При описании массива можно задать в фигурных скобках начальные значения его элементов. Элементы массива инициализируются в порядке их расположения в памяти. Например, оператор

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

определяет матрицу со следующими значениями элементов:

```
1 2 1 3 5
2 3 4 5 1
```

Если количество значений в фигурных скобках превышает количество элементов в массиве, возникает ошибка компиляции. Если значений меньше, оставшиеся элементы массива инициализируются значением по умолчанию (для основных типов это 0). Можно задавать начальные значения не для всех элементов массива. Для этого список значений констант для каждой строки заключается в дополнительные фигурные скобки. Вот, например, как заполнить единицами нулевой и первый столбцы приведенного выше массива (остальные элементы массива обнуляются):

```
int a[3][5] = { { 1, 1 }, { 1, 1 }, { 1, 1 } };
```

При явном задании хотя бы одного инициализирующего значения для каждой строки количество строк массива можно не задавать, память будет выделена под столько строк, сколько серий значений в фигурных скобках указано в списке, например:

```
int a[][5] = { { 1, 1, 7, 7, 7 }, { 1, 1, 0 }, { 1, 1, 2, 2, 2 } }; // три строки
```

В динамической области памяти можно создавать двумерные массивы с помощью операции `new` или функции `malloc`. Остановимся на первом варианте, поскольку он более безопасен и прост в использовании.

При выделении памяти под массив целиком количество строк (левую размерность) можно задавать с помощью переменной или выражения, а количество

столбцов должно быть константным выражением, то есть явно определено до выполнения программы. После слова `new` записывается тип создаваемого массива, а затем — его размерности в квадратных скобках (аналогично описанию «обычных» массивов):

```
const int m = 5;
int n; cin >> n;
int (*a)[m] = new int [n][m];           // 1
int ** b = (int **) new int [n][m];     // 2
```

Здесь показано два способа создания динамического массива. В *операторе 1* адрес начала выделенного с помощью `new` участка памяти присваивается переменной `a`, определенной как указатель на массив из `m` элементов типа `int`. Именно такой тип значения возвращает в данном случае операция `new`. Скобки необходимы, поскольку без них конструкция интерпретировалась бы как массив указателей. Всего выделяется `n` элементов.

В *операторе 2* адрес начала выделенного участка памяти присваивается переменной `b`, которая описана как «указатель на указатель на `int`», поэтому перед присваиванием требуется выполнить преобразование типа. Строго говоря, по стандарту в этом случае рекомендуется применять другую операцию преобразования типа (см. Учебник, с. 238), которую могут не поддерживать старые компиляторы:

```
int ** b = reinterpret_cast<int **> (new int [n][m]);
```

Обращение к элементам динамических массивов производится точно так же, как к элементам «обычных» — с помощью конструкции вида `a[i][j]`.

Чтобы понять, отчего динамические массивы описываются именно так, нужно разобраться в механизме индексации элемента массива, рассмотренном выше (см. с. 56). Поскольку для доступа к элементу массива применяется две операции разадресации, то переменная, в которой хранится адрес начала массива, должна быть указателем на указатель.

Более универсальный и безопасный способ выделения памяти, когда обе размерности массива задаются на этапе выполнения программы, приведен ниже. Каждая строка массива состоит из `ncol` элементов типа `int` (см. рис. 4.1):

```
cout << " Введите количество строк и столбцов :";
int nrow, ncol; cin >> nrow >> ncol;
int **a = new int *[nrow];           // выделяется память под массив указателей на строки
for ( int i = 0; i < nrow; i++ )      // цикл для выделения памяти под каждую строку
    a[i] = new int [ncol];           // выделение памяти под строку массива
```

ВНИМАНИЕ

Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции `delete []`.

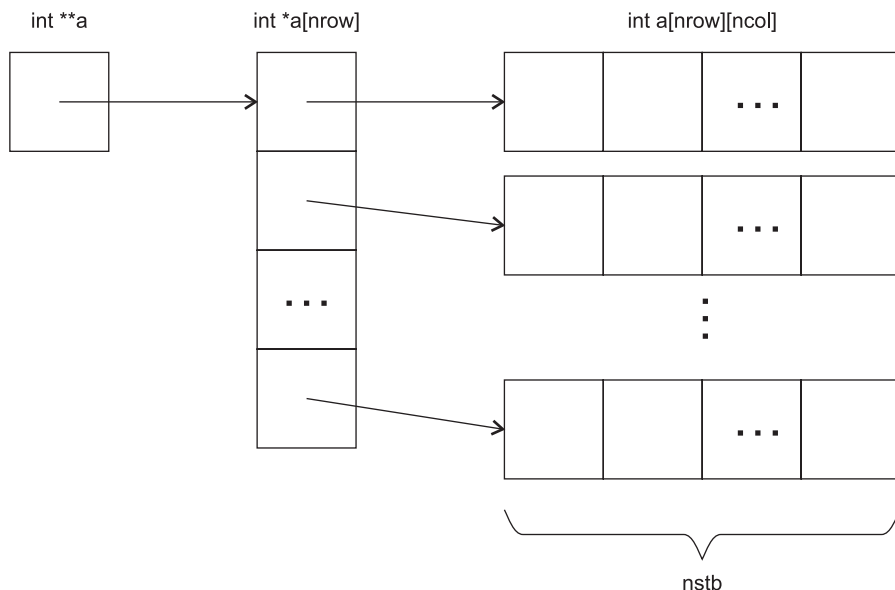


Рис. 4.1. Динамический массив с двумя переменными размерностями

Задача 4.1. Среднее арифметическое и количество положительных элементов массива

Написать программу, которая для целочисленной матрицы 10×20 определяет среднее арифметическое ее элементов и количество положительных элементов в каждой строке.

Для вычисления среднего арифметического элементов требуется найти их общую сумму, после чего разделить ее на количество элементов. Порядок просмотра массива (по строкам или по столбцам) роли не играет. Определение количества положительных элементов каждой строки требует просмотра матрицы по строкам. Обе величины вычисляются при одном просмотре матрицы (листинг 4.1).

Листинг 4.1. Среднее арифметическое и количество положительных элементов

```
#include <iostream>
#include <iomanip>
using namespace std;
void main() {
    const int nrow = 10, ncol = 20;           // размерности массива
    int a[nrow][ncol];                       // описание массива
    cout << "Введите элементы массива:" << endl;
    for ( int i = 0; i < nrow; i++ )          // ввод массива
        for ( int j = 0; j < ncol; j++ ) cin >> a[i][j];
    for ( int i = 0; i < nrow; i++ ) {        // контрольный вывод
```

```

        for ( int j = 0; j < ncol; j++ ) cout << setw(4) << a[i][j] << " ";
        cout << endl;
    }
    int n_pos_el;
    float s = 0;
    for ( int i = 0; i < nrow; i++ ) {
        n_pos_el = 0;
        for ( int j = 0; j < ncol; j++ ) {
            s += a[i][j];
            if ( a[i][j] > 0 ) n_pos_el++;
        }
        cout << " Строка: " << i << " кол-во: " << n_pos_el << endl;
    }
    s /= nrow * ncol;
    cout << "Среднее арифметическое: " << s << endl;
}

```

Размерности массива заданы именованными константами, что позволяет легко их изменять. Для упрощения отладки рекомендуется задать небольшие значения этих величин или приготовить тестовый массив в текстовом файле и изменить программу так, чтобы она читала значения из файла (см. задачу 4.3, семинар 5 и Учебник, с. 276, 280). Составляйте тестовые примеры таким образом, чтобы строки массива содержали разное количество отрицательных элементов (ни одного элемента, один и более элементов, все элементы).

Обратите внимание на два момента. Во-первых, требуется до написания алгоритма решить, каким образом будут храниться результаты. Количество положительных элементов для каждой строки свое, и в результате должно получиться столько значений, сколько строк в матрице. В данной задаче мы можем отвести для хранения этих значений одну переменную целого типа, поскольку они вычисляются последовательно, после чего выводятся на экран. Однако если эти значения впоследствии потребуются одновременно, для их хранения придется описать целочисленный массив с количеством элементов, равным количеству строк матрицы.

Второй момент — место инициализации суммы и количества. Сумма обнуляется перед циклом просмотра всей матрицы, а количество положительных элементов — перед просмотром очередной строки, поскольку для каждой строки его вычисление начинается заново. Начинающие часто забывают про инициализацию накапливаемых в цикле величин или неверно определяют ее место в программе.

СОВЕТ

Записывайте операторы инициализации накапливаемых в цикле величин непосредственно перед циклом, в котором они вычисляются.

После ввода значений предусмотрен их контрольный вывод. Чтобы элементы матрицы располагались один под другим, используется манипулятор `setw`, устанавливающий для очередного выводимого значения ширину поля в четыре символа. Для использования манипулятора подключен заголовок `<iomanip>`.

Следите за отступами при написании циклов. Все операторы одного уровня вложенности должны начинаться в одной и той же колонке. Это облегчает чтение программы и, следовательно, поиск ошибок. По вопросу расстановки фигурных скобок существуют разные мнения. На сегодня наиболее распространенными являются *стиль 1TBS* (One True Bracing Style), которого придерживались основоположники языка C (Б. Керниган и Д. Ритчи), и *стиль Алмена*:

```
for (j = 0; j < MAX_LEN; j++) {                                // стиль 1TBS
    foo();
}
for (j = 0; j < MAX_LEN; j++)                                  // стиль Алмена
{
    foo();
}
```

Наш выбор, естественно, остановился на *единственно верном (1T) стиле*.

Задача 4.2. Номер столбца из положительных элементов

Написать программу, которая для прямоугольной целочисленной матрицы определяет номер самого левого столбца, содержащего только положительные элементы. Если такого столбца нет, вывести сообщение.

Для решения этой задачи матрицу необходимо просматривать по столбцам. При этом быстрее меняется первый индекс (номер строки). Сделать вывод о том, что какой-либо столбец содержит только положительные элементы, можно только после просмотра столбца целиком; зато если в процессе просмотра встретился отрицательный элемент, можно сразу переходить к следующему столбцу.

Эта логика реализуется с помощью переменной-флага `all_posit`, которая перед началом просмотра каждого столбца устанавливается в значение `true`, а при нахождении отрицательного элемента «опрокидывается» в `false`. Если все элементы столбца положительны, флаг не опрокинется и останется истинным, что будет являться признаком присутствия в матрице искомого столбца. Если столбец найден, просматривать матрицу дальше не имеет смысла, поэтому выполняется выход из цикла и вывод результата.

Листинг 4.2. Номер столбца из положительных элементов

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    cout << "Введите количество строк и столбцов: ";
    int nrow, ncol;
    cin >> nrow >> ncol;                                // ввод размерности массива
    int **a = new int *[nrow];                          // выделение памяти под указатели на строки
    for( int i = 0; i < nrow; i++ ) a[i] = new int [ncol]; // память под строки
    cout << "Введите элементы массива:" << endl;
    for ( int i = 0; i < nrow; i++ )
```

```

        for ( int j = 0; j < ncol; j++ ) cin >> a[i][j];           // ввод массива
    int num = -1;
    bool all_posit;
    for ( int j = 0; j < ncol; j++ ) {                             // просмотр по столбцам
        all_posit = true;
        for ( int i = 0; i < nrow; i++ )                          // анализ элементов столбца
            if ( a[i][j] < 0 ) { all_posit = false; break; }
        if ( all_posit ) { num = j; break; }
    }
    if ( -1 == num ) cout << " Столбцов нет " << endl;
    else             cout << " Номер столбца: " << num << endl;
}

```

В программе необходимо предусмотреть случай, когда ни один столбец не удовлетворяет условию. Для этого переменной `num`, в которой будет храниться номер искомого столбца, присваивается начальное значение, не входящее в множество значений, допустимых для индекса, например, `-1`. Перед выводом результата его значение анализируется¹. Если после просмотра матрицы оно сохранилось неизменным, то есть осталось равным `-1`, то столбцов, удовлетворяющих заданному условию, в матрице нет.

Можно обойтись без переменной `num`, если вывести номер столбца сразу после его определения, после чего завершить программу:

```

bool all_posit;
for ( int j = 0; j < ncol; j++ ) {
    all_posit = true;
    for ( int i = 0; i < nrow; i++ )
        if ( a[i][j] < 0 ) { all_posit = false; break; }
    if ( all_posit ) { cout << " Номер столбца: " << j; return 0; }
}
cout << " Столбцов нет";

```

Задача 4.3. Упорядочивание строк матрицы

Написать программу, которая упорядочивает строки прямоугольной целочисленной матрицы по возрастанию сумм их элементов.

На этом примере мы формализуем общий порядок создания структурной программы, которому ранее следовали интуитивно. Этого порядка полезно придерживаться при решении даже простейших задач. Более подробно он изложен в Учебнике на с. 109.

I. Исходные данные, результаты и промежуточные величины. Как уже упоминалось, начинать решение задачи необходимо с описания того, что является ее

¹ Распространенной ошибкой при проверке на равенство является использование вместо него операции присваивания (`=`). Есть прием, повышающий надежность кодирования сравнения с константой: записывать константу слева от знака `"=="`; в этом случае потерю одного символа `"="` обнаружит компилятор.

исходными данными и результатами и каким образом они будут представлены в программе.

Исходные данные. Поскольку размерность матрицы неизвестна, будем использовать динамический массив элементов целого типа. Ограничимся типом `int`, хотя для общности можно было бы использовать максимально длинный целый тип.

Результаты. Результатом является та же матрица, но упорядоченная. Это значит, что нам не следует заводить для результата новую область памяти, а необходимо упорядочить матрицу на том же месте. Это важно при командной работе: представьте, что коллега полагает, что получил от вас упорядоченную матрицу, а на самом деле вы сформировали ее в совершенно другой области памяти.

Промежуточные величины. Кроме результатов, в любой программе есть промежуточные переменные. Следует выбрать их тип и способ хранения.

Очевидно, что, если требуется упорядочить матрицу по возрастанию сумм элементов ее строк, эти суммы надо вычислить и где-то хранить. Поскольку все они одновременно потребуются при упорядочивании, их надо записать в массив, количество элементов которого соответствует количеству строк матрицы, а i -й элемент содержит сумму элементов i -й строки. Количество строк заранее неизвестно, поэтому этот массив также должен быть динамическим. Сумма элементов строки может превысить диапазон значений, допустимых для отдельного элемента строки, поэтому для элемента этого массива надо выбрать тип `long`.

После того, как выбраны структуры для хранения данных, можно подумать и об алгоритме (именно в таком порядке, а не наоборот — ведь алгоритм зависит от того, каким образом представлены данные).

II. Алгоритм работы программы. Для сортировки строк воспользуемся одним из самых простых методов — *методом выбора*. Он состоит в том, что из массива выбирается наименьший элемент и меняется местами с первым, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым, и так далее $n - 1$ раз (при последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива). Одновременно с обменом элементов массива выполняется и обмен значений двух соответствующих строк матрицы.

Алгоритм сначала записывается в самом общем виде (например, так, как это сделано выше). Разветвленные алгоритмы и алгоритмы с циклами полезно представить в виде обобщенной блок-схемы. При этом надо стремиться представить алгоритм в виде простой последовательности шагов. Например, первоначально можно выделить этапы ввода исходных данных, вычислений и вывода результата. В данном случае вычисление состоит из двух шагов: формирование сумм элементов каждой строки и упорядочивание матрицы.

III. Когда алгоритм полностью прояснился, можно переходить к *кодированию* программы. Одновременно с этим (а еще лучше — гораздо раньше!) продумываются и подготавливаются *тестовые примеры*. Не ленитесь придумать переменным понятные имена и сразу же при написании аккуратно форматировать текст программы.

Функционально завершенные части алгоритма отделяются пустой строкой¹, комментарием или хотя бы комментарием вида

```
// -----
```

Рекомендуется при написании программы включать в нее промежуточную печать вычисляемых величин в удобном для восприятия формате. Это простой и надежный способ контроля хода выполнения программы. Не нужно стремиться написать сразу всю программу. Сначала пишется и отлаживается фрагмент, содержащий ввод исходных данных. Затем промежуточную печать можно убрать и переходить к следующему функционально законченному фрагменту алгоритма. Для отладки полезно выполнять программу по шагам с наблюдением значений изменяемых величин. Все популярные оболочки предоставляют такую возможность (см. Приложение). Текст программы приведен в листинге 4.3.

Листинг 4.3. Упорядочивание строк матрицы

```
#include <fstream>
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    ifstream fin( "input.txt" );
    if ( !fin ) { cout << " Файл input.txt не найден " << endl; return 1; }
    int nrow, ncol;
    fin >> nrow >> ncol;                                // ввод размерности массива
    int **a = new int *[nrow];                          // выделение памяти
    for ( int i = 0; i < nrow; i++ ) a[i] = new int [ncol]; // выделение памяти
    for ( int i = 0; i < nrow; i++ ) // ввод массива
        for ( int j = 0; j < ncol; j++ ) fin >> a[i][j];
    long *sum = new long [nrow];                        // массив сумм элементов строк
    for ( int i = 0; i < nrow; i++ ) {
        sum[i] = 0;
        for ( int j = 0; j < ncol; j++ ) sum[i] += a[i][j];
    }
    for ( int i = 0; i < nrow; i++ ) { // контрольный вывод
        for ( int j = 0; j < ncol; j++ ) cout << setw(4) << a[i][j] << " ";
        cout << "| " << sum[i] << endl;
    }
    cout << endl;
    for ( int i = 0; i < nrow - 1; i++ ) { // упорядочивание матрицы
        int nmin = i;
        for ( int j = i + 1; j < nrow; j++ )
            if ( sum[j] < sum[nmin] ) nmin = j;
        long buf_sum = sum[i]; sum[i] = sum[nmin]; sum[nmin] = buf_sum;
        for ( int j = 0; j < ncol; j++ ) {
            int buf_a = a[i][j]; a[i][j] = a[nmin][j]; a[nmin][j] = buf_a;
        }
    }
}
```

продолжение ➤

¹ К сожалению, из-за ограниченного объема книги мы не имеем такой возможности!

Листинг 4.3 (продолжение)

```
for ( int i = 0; i < nrow; i++ ) {                               // вывод упорядоченной матрицы
    for ( int j = 0; j < ncol; j++ ) cout << setw(4) << a[i][j] <<" ";
    cout << endl;
}
```

Здесь использованы две буферные переменные: `buf_sum` для обмена двух значений сумм имеет тот же тип, что и сумма, а переменная `buf_a` для обмена значений элементов массива имеет такой же тип, как и элементы массива.

Данные читаются из файла, который можно создать в интегрированной среде и сохранить там же, где исходный текст программы. В этом случае легче продумать, какие значения лучше взять для тщательного тестирования программы. Первый тестовый набор должен быть простым — например, массив из 3–4 строк с небольшими значениями элементов для того, чтобы можно было в уме проверить, правильно ли вычисляются суммы (сумма элементов строки для контроля выводится рядом со строкой, отделенная вертикальной чертой). Для второго тестового примера рекомендуется ввести значения элементов, близкие к максимальным для типа `int`. Дополнительно следует проверить, правильно ли упорядочивается массив из одной и двух строк и столбцов, поскольку многие ошибки при написании циклов связаны с неверным указанием их граничных значений.

Итоги

1. В массивах, определенных с помощью операторов описания, обе размерности должны быть константами или константными выражениями. Массив хранится по строкам в непрерывной области памяти. При описании массива можно задать начальные значения его элементов.
2. Первый индекс всегда представляет собой номер строки, второй — номер столбца. Каждый индекс может изменяться от 0 до значения соответствующей размерности, уменьшенной на единицу.
3. При выделении динамической памяти под массив целиком самую левую размерность можно задать с помощью переменной, все остальные размерности должны быть константами. Для двумерного массива это значит, что левая размерность может быть переменной, а вторая — только константой.
4. Для выделения динамической памяти под массив, в котором все размерности переменные, используются циклы.
5. Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции `delete []`.

Задания

Задания этого семинара соответствуют приведенным в Учебнике на с. 139. Рекомендуется выполнять каждое задание в двух вариантах: используя локальные и динамические массивы. Размерности локальных массивов задавать именованными константами, значения элементов массива — в списке инициализации. Ввод

данных в динамический массив выполнять из файла. Более сложные задания на массивы приведены в Учебнике на с. 142.

Вариант 1

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество строк, не содержащих ни одного нулевого элемента;
- 2) максимальное из чисел, встречающихся в заданной матрице более одного раза.

Вариант 2

Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

Характеристикой строки целочисленной матрицы назовем сумму ее положительных четных элементов. Переставляя строки заданной матрицы, располагать их в соответствии с ростом характеристик.

Вариант 3

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество столбцов, содержащих хотя бы один нулевой элемент;
- 2) номер строки, в которой находится самая длинная серия одинаковых элементов.

Вариант 4

Дана целочисленная квадратная матрица. Определить:

- 1) произведение элементов в тех строках, которые не содержат отрицательных элементов;
- 2) максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

Вариант 5

Дана целочисленная квадратная матрица. Определить:

- 1) сумму элементов в тех столбцах, которые не содержат отрицательных элементов;
- 2) минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.

Вариант 6

Дана целочисленная прямоугольная матрица. Определить:

- 1) сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент;
- 2) номера строк и столбцов всех седловых точек матрицы.

Примечание. Матрица A имеет седловую точку A_{ij} , если A_{ij} является минимальным элементом в i -й строке и максимальным в j -м столбце.

Вариант 7

Для заданной матрицы размером 8×8 найти такие k , что k -я строка матрицы совпадает с k -м столбцом.

Найти сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент.

Вариант 8

Характеристикой столбца целочисленной матрицы назовем сумму модулей его отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, располагать их в соответствии с ростом характеристик.

Найти сумму элементов в тех столбцах, которые содержат хотя бы один отрицательный элемент.

Вариант 9

Соседями элемента A_{ij} в матрице назовем элементы A_{kl} с $i-1 \leq k \leq i+1, j-1 \leq l \leq j+1, (k, l) \neq (i, j)$. Операция сглаживания матрицы дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Построить результат сглаживания заданной вещественной матрицы размером 10×10 .

В сглаженной матрице найти сумму модулей элементов, расположенных ниже главной диагонали.

Вариант 10

Элемент матрицы называется локальным минимумом, если он строго меньше всех имеющихся у него соседей. Подсчитать количество локальных минимумов заданной матрицы размером 10×10 .

Найти сумму модулей элементов, расположенных выше главной диагонали.

Вариант 11

Коэффициенты системы линейных уравнений заданы в виде прямоугольной матрицы. С помощью допустимых преобразований привести систему к треугольному виду.

Найти количество строк, среднее арифметическое элементов которых меньше заданной величины.

Вариант 12

Уплотнить заданную матрицу, удаляя из нее строки и столбцы, заполненные нулями. Найти номер первой из строк, содержащих хотя бы один положительный элемент.

Вариант 13

Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо или вниз (в зависимости от введенного режима). n может быть больше количества элементов в строке или столбце.

Вариант 14

Осуществить циклический сдвиг элементов квадратной матрицы размерности $M \times N$ вправо на k элементов таким образом: элементы 1-й строки сдвигаются в последний столбец сверху вниз, из него — в последнюю строку справа налево, из нее — в первый столбец снизу вверх, из него — в первую строку; для остальных элементов — аналогично.

Вариант 15

Дана целочисленная прямоугольная матрица. Определить номер первого из столбцов, содержащих хотя бы один нулевой элемент.

Характеристикой строки целочисленной матрицы назовем сумму ее отрицательных четных элементов. Переставляя строки заданной матрицы, располагать их в соответствии с убыванием характеристик.

Вариант 16

Упорядочить строки целочисленной прямоугольной матрицы по возрастанию количества одинаковых элементов в каждой строке.

Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

Вариант 17

Путем перестановки элементов квадратной вещественной матрицы добиться того, чтобы ее максимальный элемент находился в левом верхнем углу, следующий по величине — в позиции (2,2), следующий по величине — в позиции (3,3) и т. д., заполнив таким образом всю главную диагональ.

Найти номер первой из строк, не содержащих ни одного положительного элемента.

Вариант 18

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество строк, содержащих хотя бы один нулевой элемент;
- 2) номер столбца, в котором находится самая длинная серия одинаковых элементов.

Вариант 19

Дана целочисленная квадратная матрица. Определить:

- 1) сумму элементов в тех строках, которые не содержат отрицательных элементов;
- 2) минимум среди сумм элементов диагоналей, параллельных главной диагонали.

Вариант 20

Дана целочисленная прямоугольная матрица. Определить:

- 1) количество отрицательных элементов в тех строках, которые содержат хотя бы один нулевой элемент;
- 2) номера строк и столбцов всех седловых точек матрицы.

Примечание. Матрица A имеет седловую точку A_{ij} , если A_{ij} является минимальным элементом в i -й строке и максимальным в j -м столбце.

Семинар 5. Строки и файлы

Теоретический материал: с. 63–65, 89–93, 411–414.

В C++ есть два вида строк: C-строки, унаследованные из языка C, и класс `string` стандартной библиотеки C++. C-строка представляет собой массив символов, завершающийся символом с кодом 0. Класс `string` более безопасен, чем C-строки, но и более ресурсоемок. Для его грамотного использования требуется знание основ объектно-ориентированного программирования, поэтому мы рассмотрим его во второй части практикума, а сейчас ограничимся C-строками.

Описание строк

Память под строки, как и под другие массивы, может выделяться как компилятором, так и непосредственно в программе (динамически). Длина динамической строки может задаваться выражением, длина не-динамической строки — только константным выражением. Чаще всего используют частный случай константного выражения — именованную константу. Это удобно, поскольку при возможном изменении длины строки потребуется изменить программу только в одном месте:

```
const int len_str = 80;           // именованная константа (длина строки)
char str[len_str];               // описание строки (память выделит компилятор)
```

При задании длины необходимо учитывать завершающий нуль-символ. Например, в приведенной выше строке можно хранить не 80 символов, а только 79. При описании можно *инициализировать строку* строковой константой (нуль-символ в позиции, следующей за последним символом, формируется автоматически):

```
char a[100] = "Never trouble trouble";
```

Если строка инициализируется, ее размерность можно опускать — компилятор сам выделит память, достаточную для размещения всех символов строки и завершающего нуля:

```
char a[] = "Never trouble trouble";           // 22 символа
```

Для размещения строки *в динамической памяти* надо описать указатель на `char`, а затем выделить память с помощью `new` или `malloc`:

```
char *p = new char [m];           // этот способ предпочтительнее
char *q = (char *)malloc( m * sizeof (char)); // унаследовано из языка C
```

В этом случае длина строки может быть переменной и задаваться на этапе выполнения программы. Динамические строки, как и другие динамические массивы, нельзя инициализировать при создании. Следующий оператор создает не строковую переменную, а *указатель на строковую константу*, изменить которую невозможно:

```
char *str = "Never trouble trouble"
```

Присваивания вида `str[2]='w'` для такого указателя, хотя и компилируются, могут вызвать ошибки времени выполнения. Для *ввода-вывода строк* используются как уже известные нам объекты `cin` и `cout`, так и функции, унаследованные из библиотеки C. Рассмотрим первый способ:

```
#include <iostream>
using namespace std;
int main() {
    const int n = 80;
    char s[n];
    cin >> s; cout << s << endl;
}
```

Строка вводится так же, как и переменные известных вам типов. Ввод выполняется до первого пробельного символа (пробела, знака табуляции или символа перевода строки): если ввести строку из нескольких слов, воспринимается только первое. Если во вводимой строке больше символов, чем может вместить выделенная для ее хранения область, поведение программы не определено (она может завершиться аварийно). Можно ввести слова входной строки в отдельные строковые переменные:

```
const int n = 80;
char s[n], t[n], r[n];
cin >> s >> t >> r;
cout << s << endl << t << endl << r << endl;
```

Если требуется ввести строку, состоящую из нескольких слов, в одну строковую переменную, используются *методы (функции)* `getline` или `get` класса `istream`, объектом которого является `cin`. Во второй части практикума вы узнаете, что такое методы класса, а пока можно пользоваться ими как волшебным заклинанием, не вдумываясь в смысл. Единственное, что пока нужно знать, — синтаксис вызова метода: после имени объекта ставится точка, а затем пишется имя метода:

```
const int n = 80;
char s[n];
cin.getline( s, n ); cout << s << endl;
cin.get( s, n );     cout << s << endl;
```

Метод `getline` считывает из входного потока $n - 1$ символов или менее (если символ перевода строки встретится раньше) и записывает их в строковую переменную `s`. Символ перевода строки¹ `\n` также считывается (удаляется) из входного потока, но не записывается в строковую переменную, вместо него размещается

¹ Этот символ появляется во входном потоке, когда вы нажимаете клавишу **Enter**.

завершающий 0. Если в исходных данных более $n - 1$ символов, следующий ввод будет выполняться из той же строки, начиная с первого несчитанного символа.

Метод `get` работает аналогично, но оставляет в потоке символ перевода строки. В строковую переменную добавляется завершающий 0.

Никогда не обращайтесь к разновидности метода `get` с двумя аргументами два раза подряд, не удалив `\n` из входного потока. Пример:

```
cin.get( s, n );           // 1 - считывание строки
cout << s << endl;        // 2 - вывод строки
cin.get( s, n );           // 3 - считывание строки
cout << s << endl;        // 4 - вывод строки
cin.get( s, n );           // 5 - считывание строки
cout << s << endl;        // 6 - вывод строки
cout << "Конец - делу венец" << endl; // 7
```

При выполнении этого фрагмента вы увидите на экране первую строку, выведенную *оператором 2*, а затем завершающее сообщение, выведенное *оператором 7*. Какие бы прекрасные строки вы ни ввели с клавиатуры в надежде, что они будут прочитаны *операторами 3 и 5*, метод `get` «уткнется» в символ `\n`, оставленный во входном потоке от первого вызова этого метода (*оператор 1*). В результате будут считаны и, соответственно, выведены на экран пустые строки, а символ `\n` так и останется «торчать» во входном потоке.

Возможное решение этой проблемы — удалить символ `\n` из входного потока путем вызова метода `get` без параметров, то есть вставить `cin.get()` после *операторов 1 и 3*. Более простое решение — использовать метод `getline`, который после прочтения строки не оставляет во входном потоке символ `\n`. Если требуется ввести несколько строк, метод `getline` удобно использовать в заголовке цикла, например:

```
const int n = 80;
char s[n];
while ( cin.getline( s, n ) ) {
    cout << s << endl;
    ... // обработка строки
}
```

Рассмотрим теперь способы ввода-вывода строк, переключавшиеся в C++ из языка C. Во-первых, можно использовать для ввода строки известную нам функцию `scanf`, а для вывода — `printf`, задав спецификацию формата `%s`:

```
#include <cstdio>
using namespace std;
int main() {
    const int n = 10;
    char s[n];
    scanf( "%s", s ); printf( "%s", s );
}
```

Имя строки, как и любого массива, является указателем на его начало, поэтому использовавшаяся в предыдущих примерах применения функции `scanf` операция

взятия адреса (&) опущена. Ввод будет выполняться так же, как и для классов ввода-вывода — до первого пробельного символа. Чтобы ввести строку, состоящую из нескольких слов, используется спецификация %s (символы) с указанием максимального количества вводимых символов, например `scanf("%10c", s)`.

Количество символов может быть только целой константой. При выводе можно задать перед спецификацией %s количество позиций, отводимых под строку: `printf("%15s", s)`. Строка при этом выравнивается по правому краю отведенного поля. Если заданное количество позиций недостаточно для размещения строки, оно игнорируется, и строка выводится целиком. Спецификации формата описаны в Учебнике на с. 387, а сами функции семейства `printf` — на с. 411.

Библиотека содержит функции `gets` и `puts`, специально предназначенные для ввода-вывода строк. Вот как выглядят ввод и вывод строки из предыдущего примера:

```
gets(s); puts(s);
```

Функция `gets(s)` читает символы с клавиатуры до появления символа новой строки и помещает их в строку `s` (сам символ новой строки в строку не включается, вместо него в строку заносится нуль-символ). Функция возвращает указатель на строку `s`, а в случае возникновения ошибки или конца файла — `NULL`.

Функция `puts(s)` выводит строку `s` на стандартное устройство вывода, заменяя завершающий 0 символом новой строки. Возвращает неотрицательное значение при успехе или EOF при ошибке.

Функциями семейства `printf` удобнее пользоваться в том случае, если в одном операторе требуется ввести или вывести данные различных типов. Если же работа выполняется только со строками, проще применять специальные функции для ввода-вывода строк `gets` и `puts`.

Операции со строками

Для строк не определена операция присваивания, поскольку они не являются основным типом данных. Присваивание делается с помощью функций библиотеки или посимвольно «вручную» (что чревато ошибками). Например, чтобы присвоить строке `p` строку `a`, можно воспользоваться функцией `strcpy` или `strncpy`, описанными в заголовочном файле `<string>`:

```
char a[100] = "Never trouble trouble";  
char *p = new char [m];  
strcpy( p, a );  
strncpy( p, a, strlen(a) + 1 );
```

Функция `strcpy(p, a)` копирует все символы строки `a`, включая завершающий 0, в строку `p`. Функция `strncpy(p, a, n)` выполняет то же самое, но не более `n` символов. Если нуль-символ в исходной строке встретится раньше, копирование прекращается, а оставшиеся до `n` символы строки `p` заполняются нуль-символами. В противном случае (если `n` ≤ длине строки `a`), нуль-символ в `p` не добавляется. Обе функции возвращают указатель на результирующую строку. Если области

памяти, занимаемые исходной и результирующей строками, перекрываются, поведение программы не определено.

Программист должен сам заботиться о том, чтобы в строке-приемнике хватило места для строки-источника (в данном случае при выделении памяти значение переменной *m* должно быть больше или равно 100), и о том, чтобы строка всегда имела завершающий нуль-символ.

ВНИМАНИЕ

Выход за границы строки и отсутствие нуль-символа являются распространенными причинами ошибок в программах обработки строк.

Функция `strlen(a)` возвращает фактическую длину строки *a*, не включая нуль-символ. Для преобразования строки в целое число используется функция `atoi(str)`. Она преобразует строку, содержащую символьное представление целого числа, в соответствующее целое число. Признаком конца числа служит первый символ, который не может быть интерпретирован как принадлежащий числу. Если преобразование не удалось, возвращает 0. Аналогичные функции преобразования строки в длинное целое число (`long`) и в вещественное число с двойной точностью (`double`) называются `atol` и `atof` соответственно. Пример применения функций преобразования:

```
char   a[] = "10) Рост - 162 см, вес - 63.5 кг";
int    num   = atoi( a );
long   height = atol( &a[11] );
double weight = atof( &a[25] );
cout << num << ' ' << height << ' ' << weight;
```

Библиотека предоставляет также функции для сравнения строк и подстрок, объединения строк, поиска в строке символа и подстроки и выделения из строки лексем. Эти функции описаны в Учебнике на с. 414 – 446. В процессе разбора задач мы рассмотрим некоторые из них.

Работа с символами

Для хранения отдельных символов используются переменные типа `char`. Их ввод-вывод также может выполняться как с помощью классов ввода-вывода, так и с помощью функций библиотеки. При использовании классов ввод-вывод символов выполняется с помощью *операций* помещения в поток `<<` и извлечения из потока `>>`, а также *методов* `get()` и `get(char)`. Пример применения операций:

```
#include <iostream>
using namespace std;
int main() {
    char c, d, e;
    cin >> c; cin >> d >> e;
    cout << c << d << e << endl;
}
```


Вводимые символы могут разделяться или не разделяться пробельными символами, поэтому таким способом ввести символ пробела нельзя. Для ввода любого символа, включая пробельные, можно воспользоваться методом `get()` или `get(c)`:

```
char c, d, e;
c = cin.get(); cin.get(d); cin.get(e);
cout << c << d << e << endl;
```

Метод `get()` возвращает код извлеченного из потока символа или EOF, а метод `get(c)` записывает извлеченный символ в переменную, переданную ему в качестве аргумента, а возвращает ссылку на поток.

В заголовочном файле `<cstdio>` определены функции `getchar` для ввода символа со стандартного ввода и `putchar` для вывода:

```
#include <cstdio>
using namespace std;
int main() {
    char c, d;
    c = getchar(); putchar(c);
    d = getchar(); putchar(d);
}
```

В библиотеке определен целый ряд функций, проверяющих принадлежность символа какому-либо множеству, например, множеству букв (`isalpha`), разделителей (`isspace`), знаков пунктуации (`ispunct`), цифр (`isdigit`) и т.д. Описание этих функций приведено в Учебнике на с. 92 и с. 409–446.

Задача 5.1. Поиск подстроки

Написать программу, которая определяет, встречается ли в заданном текстовом файле заданная последовательность символов. Длина строки текста не превышает 80 символов, текст не содержит переносов слов, последовательность не содержит пробельных символов.

На предыдущем семинаре на примере задачи 4.3 мы рассмотрели общий порядок действий при создании программы. Будем придерживаться его и впредь.

I. Исходные данные, результаты и промежуточные величины. Исходные данные:

1. Текстовый файл неизвестного размера, состоящий из строк длиной не более 80 символов. Поскольку по условию переносы отсутствуют, можно ограничиться поиском заданной последовательности в каждой строке отдельно. Следовательно, необходимо помнить только одну текущую строку файла. Для ее хранения выделим строковую переменную длиной 81 символ (дополнительный символ требуется для завершающего нуля).
2. Последовательность символов для поиска, вводимая с клавиатуры. Поскольку по условию задачи она не содержит пробельных символов, ее длина также не должна быть более 80 символов, иначе поиск завершится неудачей. Для ее хранения также выделим строковую переменную длиной 81 символ.

Результатом работы программы является сообщение либо о наличии заданной последовательности, либо о ее отсутствии. Представим варианты сообщений в программе в виде строковых констант. Для хранения длины строки будем использовать именованную константу. Для работы с файлом потребуется служебная переменная соответствующего типа.

II. Алгоритм решения задачи.

1. Построчно считывать текст из файла.
2. Для каждой строки проверять, содержится ли в ней заданная последовательность.
3. Если да, напечатать сообщение о наличии заданной последовательности и завершить программу.
4. При нормальном выходе из цикла напечатать сообщение об отсутствии заданной последовательности и завершить программу.

III. Программа и тестовые примеры. Программа приведена в листинге 5.1.

Листинг 5.1. Поиск подстроки

```
#include <fstream>
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    const int len = 81; // 1
    char word[len], line[len]; // 2
    cout << "Введите слово для поиска: "; cin >> word;
    ifstream fin( "text1.txt" ); // 3
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; return 1; } // 4
    while ( fin.getline( line, len ) ) { // 5
        if ( strstr( line, word ) ) { cout << "Присутствует!" << endl; return 0; }
    }
    cout << "Отсутствует!" << endl;}
```

В *операторе 1* описывается константа, определяющая длину строки файла и длину последовательности. В *операторе 2* описывается переменная `line` для размещения очередной строки файла и переменная `word` для хранения искомой последовательности символов. В *операторе 3* определяется объект `fin` класса входных потоков `ifstream`. С этим объектом можно работать так же, как со стандартными объектами `cin` и `cout`, то есть использовать операции помещения в поток `<<` и извлечения из потока `>>`, а также рассмотренные выше функции `get`, `getline` и другие. Предполагается, что файл с именем `text.txt` находится в том же каталоге, что и текст программы, иначе следует указать полный путь, дублируя символ обратной косой черты, так как иначе он будет иметь специальное значение:

```
ifstream fin( "c:\\prim\\cpp\\text.txt" ); // 3
```

В *операторе 4* проверяется успешность создания объекта `fin`. *Файлы, открываемые для чтения, нужно обязательно проверять!* В *цикле 5* выполняется чтение из файла

в переменную `line`. Метод `getline`, описанный выше, при достижении конца файла вернет значение, завершающее цикл.

Для анализа строки применяется функция `strstr(line, word)`. Она выполняет поиск подстроки `word` в строке `line`. Обе строки должны завершаться нуль-символами. В случае успешного поиска функция возвращает указатель на найденную подстроку, в случае неудачи — `NULL` (*пустой указатель*). Если искомая подстрока пуста, функция возвращает указатель на начало строки `line`.

Для тестирования создайте текстовый файл, состоящий из нескольких строк. Длина хотя бы одной из строк должна быть равна 80 символам. Для тестирования программы следует запустить ее по крайней мере два раза: введя с клавиатуры слово, содержащееся в файле, и слово, которого в нем нет.

Для успешного поиска последовательностей, состоящих из *русских букв*, файл надо создать в текстовом редакторе с кодировкой ASCII — например, во встроенном редакторе оболочки `Far`. Если же текстовый файл уже существует и создан в кодировке ANSI, используемой в Windows, для преобразования в эту же кодировку строки, вводимой с клавиатуры в переменную `word`, придется воспользоваться нестандартной функцией `OemToChar`, описанной в заголовочном файле `<windows.h>`. У нее два параметра: исходная строка и результирующая. Существует и функция `CharToOem`, выполняющая обратную перекодировку. Напомним, что вопрос отображения русского текста в консольном окне рассматривался на первом семинаре (см. с. 18).

Задача 5.2. Подсчет количества вхождений слова в текст

Написать программу, которая определяет, сколько раз встретилось заданное слово в текстовом файле, длина строки в котором не превышает 80 символов. Текст не содержит переносов слов.

На первый взгляд эта задача не сильно отличается от предыдущей: вместо факта наличия искомой последовательности в файле требуется подсчитать количество вхождений слова, то есть после первого удачного поиска не выходить из цикла просмотра, а увеличить счетчик и продолжать просмотр. В целом это верно, однако здесь требуется найти не просто последовательность символов, а законченное слово.

Определим *слово* как последовательность алфавитно-цифровых символов, после которых следует знак пунктуации, разделитель или признак конца строки. Слово может находиться либо в начале строки, либо после разделителя или знака пунктуации. Это можно записать следующим образом (фигурные скобки и вертикальная черта означают выбор из альтернатив):

```
слово =  
{ начало строки | знак пунктуации | разделитель }  
символы, составляющие слово  
{ конец строки | знак пунктуации | разделитель }
```

I. Исходные данные и результаты. Исходные данные:

1. Текстовый файл неизвестного размера, состоящий из строк длиной не более 80 символов. Поскольку по условию переносы отсутствуют, ограничимся поиском слова в каждой строке отдельно, поэтому выделим строку длиной 81 символ.
 2. Слово для поиска, вводимое с клавиатуры (также выделим строку в 81 символ).
- Результатом работы программы является количество вхождений слова в текст. Представим его в программе в виде целой переменной. Для хранения длины строки будем использовать именованную константу, а для хранения фактического количества символов в слове — переменную целого типа. Для работы с файлом потребуются служебная переменная соответствующего типа.

II. Алгоритм решения задачи.

1. Построчно считывать текст из файла.
2. Просматривая каждую строку, искать в ней заданное слово. При каждом нахождении слова увеличивать счетчик.

Детализируем второй пункт алгоритма. Очевидно, что слово может встречаться в строке многократно, поэтому для поиска следует организовать цикл просмотра строки, который будет работать, пока происходит обнаружение в строке последовательности символов, составляющих слово.

При обнаружении совпадения с символами, составляющими слово, требуется определить, что оно является отдельным словом, а не частью другого (кроме этого, слово может быть написано в разных регистрах, но мы для простоты будем искать точное совпадение). Например, задано слово «кот». Эта последовательность символов содержится в словах «котенок», «трикотаж», «трескотня», «апперкот». Следовательно, требуется проверить символ, стоящий после слова, а в случае, когда слово не находится в начале строки, — еще и символ перед словом. Эти символы проверяются на принадлежность множеству знаков пунктуации и разделителей.

III. Программа и тестовые примеры. Разобьем написание программы (листинг 5.2) на последовательность шагов. Они помечены соответствующими комментариями.

Листинг 5.2. Подсчет количества вхождений слова в текст

```
#include <fstream> // 1
#include <iostream> // 1
#include <cstring> // 1
#include <cctype> // 1
using namespace std; // 1
int main() { // 1
    const int len = 81; // 1
    char word[len], line[len]; // 1
    cout << " Введите слово для поиска: "; cin >> word; // 1
    int l_word = strlen( word ); // 1
    ifstream fin( "text.txt" ); // 1
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; return 1; } // 1
    int count = 0; // 3
    while ( fin.getline( line, len ) ) { // 2
        cout << line << endl; // контрольный вывод (убрать перед шагом 3) // 2
```

```

char *p = line;                                     // 3
while( p = strstr( p, word ) ) {                     // 3
    cout << " совпадение: " << p << endl; // контрольный вывод // 3
    char *c = p;                                     // 4
    p += 1 word;                                     // 3
    if ( c != line ) // слово не в начале строки? // 4
        // символ перед словом не разделитель?
        if ( !ispunct(*(c - 1) ) && !isspace(*(c - 1) ) // 4
            continue; // 4
        // символ после слова разделитель?
        if ( ispunct(*p) || isspace(*p) || (*p == '\0') ) // 4
            count++; // 3
    } // 3
} // 2
cout << "Количество вхождений слова: " << count << endl; // 3
} // 1

```

Шаг 1. Ввести «скелет» программы. Добавить контрольный вывод введенного слова. Запустив программу, проверить ввод слова и успешность открытия файла. Для проверки вывода сообщения об ошибке задать имя несуществующего файла.

Шаг 2. Добавить цикл чтения из файла с контрольным выводом считанной строки.

Шаг 3. Добавить цикл поиска последовательности символов, составляющих слово, с контрольным выводом. Для многократного поиска вхождения подстроки в заголовке цикла используется функция `strstr`. Очередной поиск должен выполняться с позиции, следующей за найденной на предыдущем проходе подстрокой. Для хранения этой позиции определяется вспомогательный указатель `p`, который на каждой итерации цикла наращивается на длину подстроки. Также вводится счетчик количества совпадений. На данном этапе он считает не количество слов, а количество вхождений последовательности символов, составляющих слово.

Шаг 4. Добавить в программу анализ принадлежности символов, находящихся перед словом и после него, множеству знаков пунктуации и разделителей. Здесь вводится служебная переменная `s` для хранения адреса начала вхождения подстроки. Символы, ограничивающие слово, проверяются с помощью функций `ispunct` и `isspace`, прототипы которых хранятся в заголовочном файле `<cctype>`. Символ, стоящий после слова, проверяется также на признак конца строки (для случая, когда искомое слово находится в конце строки).

Для *тестирования программы* требуется создать файл с текстом, в котором заданное слово встречается: в начале строки; в конце строки; в середине строки; несколько раз в одной строке; как часть других слов, находящаяся в начале, середине и конце этих слов; в скобках, кавычках и других разделителях. Длина хотя бы одной из строк должна быть равна 80 символам. Программу следует выполнить, введя с клавиатуры слово, содержащееся в файле, и слово, которого в нем нет.

Рассмотрим теперь другой вариант решения этой задачи (листинг 5.3). В библиотеке есть «устаревшая» функция `strtok`, которая разбивает переданную ей строку на лексемы в соответствии с заданным набором разделителей. Если воспользоваться этой функцией, нам не придется «вручную» выделять и проверять начало и конец слова. Правда, придется задать список разделителей.

Листинг 5.3. Подсчет количества вхождений слова в текст с помощью strtok

```

#include <fstream>
#include <iostream>
#include <cstring>
using namespace std;
int main() {
    const int len = 81;
    char word[len], line[len];
    char delims[] = ",.!? /<>|)(*;\\"";
    cout << "Введите слово для поиска: "; cin >> word;
    ifstream fin( "text.txt" );
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; return 1; }
    int count = 0;
    while ( fin.getline( line, len ) ) {
        char *token = strtok( line, delims );           // 1
        while( token != NULL ) {
            if ( !strcmp ( token, word ) )count++;      // 2
            token = strtok( NULL, delims );             // 3
        }
    }
    cout << "Количество вхождений слова: " << count << endl;
}

```

Первый вызов функции strtok в *операторе 1* формирует адрес первой лексемы (слова) строки line. Он сохраняется в переменной token. Функция strtok заменяет на NULL разделитель, находящийся после найденного слова, поэтому в *операторе 2* можно сравнить на равенство искомое и выделенное слово. В *операторе 3* выполняется поиск следующей лексемы в той же строке. Для этого следует задать в функции strtok в качестве первого параметра NULL.

Программа стала короче и яснее. Как видите, выбор средств, предоставляемых языком и библиотеками, влияет на алгоритм, поэтому перед тем, как продумывать алгоритм, необходимо эти средства изучить. Представьте, как раздулась бы эта программа вообще без использования функций работы со строками и символами!

Задача 5.3. Вывод вопросительных предложений

Написать программу, которая считывает текст из файла и выводит на экран только вопросительные предложения из этого текста.

I. Исходные данные и результаты. *Исходные данные:* текстовый файл неизвестного размера, состоящий из неизвестного количества предложений. Предложение может занимать несколько строк, поэтому ограничиться буфером на одну строку в данной задаче нельзя. Примем решение выделить буфер, в который поместится весь файл.

Результаты являются частью исходных данных, поэтому дополнительное пространство под них не выделяется. Будем хранить длину файла в переменной

длинного целого типа. Для организации вывода предложений понадобятся переменные того же типа, хранящие позиции начала и конца предложения.

II. Алгоритм решения задачи.

1. Открыть файл, определить его длину в байтах.
2. Выделить в динамической памяти буфер соответствующего размера.
3. Считать файл с диска в буфер.
4. Анализируя буфер посимвольно, выделять предложения. Если предложение оканчивается вопросительным знаком, вывести его на экран.

Детализируем последний пункт алгоритма. Для вывода предложения необходимо хранить позиции его начала и конца. Предложение может оканчиваться точкой, восклицательным или вопросительным знаком. В первых двух случаях предложение пропускается. Это выражается в том, что значение позиции начала предложения обновляется: оно устанавливается равным символу, следующему за текущим, и просмотр продолжается. При обнаружении знака вопроса предложение выводится, после чего позиция начала предложения также обновляется.

III. Программа и тестовые примеры. Текст программы приведен в листинге 5.4. Рекомендуем вам самостоятельно разбить его для отладки на последовательность шагов аналогично предыдущим примерам, вставляя и удаляя отладочную печать. Файл с тестовым примером должен содержать предложения различной длины (от нескольких символов до нескольких строк), в том числе и вопросительные.

Листинг 5.4. Вывод вопросительных предложений

```
#include <fstream>
#include <iostream>
#include <cstdio>
using namespace std;
int main() {
    ifstream fin( "text.txt" );
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; return 1; }
    fin.seekg( 0, ios::end );
    long len = fin.tellg();
    char *buf = new char [len + 1];
    fin.seekg( 0, ios::beg );
    fin.read( buf, len );
    buf[len] = '\0'; long n = 0, i = 0;
    while( buf[i] ) {
        if( buf[i] == '?' ) {
            for ( int j = n; j <= i; j++ ) cout << buf[j];
            n = i + 1;
        }
        if ( buf[i] == '.' || buf[i] == '!' ) n = i + 1;
        i++;
    }
    fin.close();
}
```

Для определения длины файла используются методы `seekg` и `tellg` класса `ifstream`. С любым файлом при его открытии связывается так называемая текущая позиция чтения или записи. Когда файл открывается для чтения, эта позиция устанавливается на начало файла. Для определения длины файла мы смещаем ее на конец файла с помощью метода `seekg` (*оператор 1*), а затем с помощью `tellg` получаем ее значение, запомнив его в переменной `len` (*оператор 2*).

Метод `seekg (offset, org)` смещает текущую позицию чтения из файла на `offset` байтов относительно параметра `org`, который может принимать значения: `ios::beg` — от начала файла, `ios::cur` — от текущей позиции, `ios::end` — от конца файла. Константы `beg`, `cur` и `end` определены в классе `ios`, предке `ifstream`, а символы `::` означают операцию доступа к этому классу.

В *операторе 3* выделяется `len + 1` байтов под символьную строку `buf`, в которой будет храниться текст из файла. Мы выделяем на один байт больше, чем длина файла, чтобы после считывания файла записать в этот байт нуль-символ.

Для чтения информации требуется снова переместить текущую позицию на начало файла (*оператор 4*). Собственно чтение выполняется в *операторе 5* с помощью метода `read(buf, len)`, который считывает из файла `len` символов (или менее, если конец файла встретится раньше) в символьный массив `buf`. В *операторе 6* определяются служебные переменные. В переменной `n` будет храниться позиция начала текущего предложения, переменная `i` используется для просмотра массива.

Цикл просмотра массива `buf` (*оператор 7*) завершается, когда встретился нуль-символ. Если очередным символом оказался вопросительный знак (*оператор 8*), выполняется вывод символов, начиная с позиции `n` и заканчивая текущей, после чего в переменную `n` заносится позиция начала нового предложения. *Оператор 9* (закрытие потока) в данном случае не является обязательным, так как он не обязателен, только когда требуется закрыть поток раньше окончания действия его области видимости.

Если требуется вывести результаты выполнения программы не на экран, а в файл, в программе следует описать объект класса выходных потоков `ofstream`, а затем использовать его аналогично другим потоковым объектам, например:

```
ofstream fout( "textout.txt" );
if ( !fout ) { cout << "Ошибка открытия файла вывода" << endl; return 1; }
...
fout << buf[j];
```

При выполнении некоторых заданий этого семинара может потребоваться гораздо менее эффективное посимвольное чтение из файла. При использовании потоков оно выполняется с помощью метода `get`. Например, для приведенной выше программы посимвольный ввод выглядит следующим образом:

```
while( ( buf[i] = fin.get() ) != EOF ) {
    // ... тело цикла
    i++;
}
```


Функции вывода в файл описаны в Учебнике на с. 90 и 411. Смешивать в одной программе ввод-вывод с помощью потоковых классов и функций библиотеки не рекомендуется. В целом программа, написанная с использованием функций библиотеки, может получиться более быстродействующей, но менее безопасной, поскольку программист должен заботиться о большем количестве деталей.

Итоги

1. При задании длины строки необходимо учитывать завершающий нуль-символ. Длина не-динамической строки должна быть константным выражением, длина динамической строки может быть переменной. Динамические строки нельзя инициализировать при создании.
2. Присваивание строк выполняется с помощью функций библиотеки.
3. Для консольного ввода-вывода строк используются либо объекты `cin` и `cout`, либо функции библиотеки `gets`, `scanf` и `puts`, `printf`. Ввод-вывод из файла может выполняться с помощью либо объектов классов `ifstream` и `ofstream`, либо функций библиотеки `fgets`, `fscanf` и `fputs`, `fprintf`.
4. Ввод строки с помощью операции `>>` выполняется до первого пробельного символа. Для ввода строки, содержащей пробелы, можно использовать либо методы `getline` или `get` класса `istream`, либо функции `gets` и `scanf`.
5. Смешивать в одной программе ввод-вывод с помощью потоковых классов и с помощью функций библиотеки не рекомендуется. Программа, написанная с использованием функций, а не классов ввода-вывода, может получиться более быстродействующей, но менее безопасной.
6. Недостатком C-строк по сравнению с классом `string` является отсутствие проверки выхода строки за пределы отведенной ей памяти. Выход за границы строки и отсутствие нуль-символа являются распространенными причинами ошибок в программах.
7. Средства, предоставляемые языком и библиотеками, влияют на алгоритм решения задачи, поэтому перед разработкой алгоритма необходимо их изучить.
8. Разбивайте написание и отладку программы на последовательность шагов.

Задания

1. Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.
2. Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие введенное с клавиатуры слово.
3. Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.
4. Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.
5. Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.

6. Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.
7. Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих из не более чем четырех буквами.
8. Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.
9. Написать программу, которая считывает текст из файла и выводит на экран только предложения, состоящие из заданного количества слов.
10. Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся и оканчивающиеся гласными букв.
11. Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначных чисел.
12. Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут находиться только пробельные символы.
13. Написать программу, которая считывает английский текст из файла и выводит его на экран, заменив каждую первую букву слов, начинающихся с гласной буквы, на прописную.
14. Написать программу, которая считывает текст из файла и выводит его на экран, заменив цифры от 0 до 9 на слова «ноль», «один», ..., «девять», начиная каждое предложение с новой строки.
15. Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.
16. Написать программу, которая считывает текст из файла и выводит на экран сначала вопросительные, а затем восклицательные предложения.
17. Написать программу, которая считывает текст из файла и выводит его на экран, добавляя после каждого предложения, сколько раз встретилось в нем введенное с клавиатуры слово.
18. Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке.
19. Написать программу, которая считывает текст из файла и выводит на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.
20. Написать программу, которая считывает текст из файла и выводит на экран предложения, содержащие максимальное количество знаков пунктуации.

Семинар 6. Структуры

Теоретический материал: с. 67–69.

Структуры в C++ обладают практически теми же возможностями, что и классы, но чаще их применяют исключительно для группирования (объединения) данных. В отличие от массива, в структуру можно объединять данные различных типов.

Например, требуется обрабатывать информацию о расписании работы конференц-зала, и для каждого мероприятия надо знать время, тему, фамилию организатора и количество участников. Поскольку все это относится к одному событию, логично дать ему общее имя. Для этого описывается новый тип данных с именем `Event`. Переменные этого типа можно описать так же, как переменные встроенных типов:

```
struct Event {
    int  hour, min, num;
    char theme[100], name[100];
};
Event e1, e2[10];
```

// обратите внимание: точка с запятой после описания обязательна!
// описание структуры и массива структур

Если структура используется только в одном месте программы, можно совместить описание типа с описанием переменных, при этом имя типа можно не указывать:

```
struct {
    int  hour, min, num;
    char theme[100], name[100];
} e1, e2[10];
```

Переменные структурного типа можно размещать и в динамической области памяти, для этого надо описать указатель на структуру и выделить под нее место:

```
Event *pe = new Event;           // структура
Event *pm = new Event[m];        // массив структур
```

Элементы структуры называются *полями*. Поля могут быть любого основного типа, массивом, указателем, объединением или структурой. Для обращения к полю используется *операция выбора* («точка» для переменной и `->` для указателя):

```
e1.hour = 12; e1.min = 30;
strncpy( e2[0].theme, "Выращивание кактусов в условиях Крайнего Севера", 99 );
pe->num = 30;           // или (*pe).num = 30;
pm[2].hour = 14;        // или (*(pm + 2)).hour = 14;
```

Структуры (не-динамические) можно *инициализировать*:

```
Event e3 = { 12, 30, "Выращивание кактусов в условиях Крайнего Севера", 25 };
```

Структуры одного типа можно *присваивать* друг другу:

```
*pe = e1; pm[1] = e1; pm[4] = e2[0];
```

Присваивание — это все, что можно делать со структурами целиком. Другие операции, например, сравнение на равенство и вывод, не определены. Впрочем, можно задать их самостоятельно, поскольку в структурах, как и в классах, разрешается определять свои операции. Мы рассмотрим эту тему во второй части.

Ввод-вывод структур выполняется поэлементно. Вот, например, как выглядит ввод и вывод описанной выше структуры e1:

```
cin >> e1.hour >> e1.min;           // с использованием классов: подключить <iostream>
cin.getline( e1.theme, 100 );
cout << e1.hour << ' ' << e1.min << ' ' << e1.theme << endl;
                                   // в стиле C: подключить <cstdio>:
scanf( "%d%d", &e1.hour, &e1.min ); gets( e1.theme );
printf( "%d %d %s", e1.hour, e1.min, e1.theme );
```

Задача 6.1. Поиск в простой базе (массив структур)

В текстовом файле хранится база отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия и инициалы (30 поз., фамилия должна начинаться с первой позиции), год рождения (5 поз.), оклад (10 поз.). Написать программу, которая по заданной фамилии выводит на экран сведения о сотруднике, подсчитывая средний оклад всех запрошенных сотрудников.

I. Исходные данные и результаты. *Исходные данные.* База сотрудников находится в текстовом файле. Поиск по базе будет выполняться многократно, поэтому всю информацию желательно хранить в оперативной памяти, поскольку многократное чтение из файла крайне нерационально. Количество строк файла по условию задачи ограничено, поэтому можно выделить для их хранения массив из 100 элементов. Каждый элемент массива будет содержать сведения об одном сотруднике. Поскольку эти сведения разнородные, удобно организовать их в виде структуры.

ПРИМЕЧАНИЕ

Строго говоря, для решения этой конкретной задачи запись о сотруднике может быть просто строкой символов, из которой при необходимости выделяется подстрока с окладом, преобразуемая затем в число, но мы для общности и удобства дальнейшей модификации программы будем использовать структуру.

Результаты. В результате работы программы необходимо вывести на экран требуемые элементы исходного массива. Поскольку эти результаты являются выборкой из исходных данных, дополнительная память для них не отводится. Для подсчета среднего оклада опишем переменную вещественного типа.

II. Алгоритм решения задачи очевиден:

1. Ввести из файла в массив сведения о сотрудниках.
2. Организовать цикл вывода сведений о сотруднике:
 - ввести с клавиатуры фамилию;
 - выполнить поиск сотрудника в массиве;
 - увеличить суммарный оклад и счетчик количества сотрудников;
 - вывести сведения о сотруднике или сообщение об их отсутствии.
3. Вывести средний оклад.

Для простоты условимся, что для выхода из цикла вывода сведений о сотрудниках вместо фамилии следует ввести слово “end”.

III. Программа и тестовые примеры. Программа приведена в листинге 6.1.**Листинг 6.1. Поиск в массиве структур**

```
#include <windows.h> // содержит прототип OemToChar
#include <fstream>
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
int main() {
    const int l_name = 30, l_year = 5, l_pay = 10, // длины полей строки файла
            l_buf = l_name + l_year + l_pay; // длина буфера как сумма длин полей
    struct Man { // структура для хранения сведений об одном сотруднике
        int birth_year;
        char name[l_name + 1]; // длина поля задана с учетом нуля-символа
        float pay;
    };
    const int l_dbase = 100;
    Man dbase[l_dbase]; // массив структур для хранения всей базы
    char buf[l_buf + 1]; // буфер для ввода строки из файла
    char name[l_name + 1]; // строка для фамилии запрашиваемого сотрудника
    ifstream fin( "dbase.txt" ); // 1
    if ( !fin ) { cout << " Ошибка открытия файла "; return 1; }
    int i = 0;
    while ( fin.getline( buf, l_buf ) ) { // 2
        if ( i >= l_dbase ) { cout << " Слишком длинный файл "; return 1; }
        strncpy( dbase[i].name, buf, l_name );
        dbase[i].name[l_name] = '\0';
        dbase[i].birth_year = atoi(&buf[l_name]);
        dbase[i].pay = atof(&buf[l_name + l_year]);
        i++;
    }
    int n_record = i, n_man = 0; // 3
    float mean_pay = 0;
    while ( true ) { // 4
        cout << " Введите фамилию или слово end: ";
```

продолжение ➤

Листинг 6.1 (продолжение)

```

    cin >> name;
    OemToChar(name, name); // для ввода кириллицы
    if ( strcmp( name, "end" ) == 0 ) break; // 5
    bool not_found = true; // 6
    for ( int i = 0; i < n_record; i++ ) { // 7
        if ( strstr( dbase[i].name, name ) ) // 8
            if ( dbase[i].name[strlen( name )] == ' ' ) { // 9
                cout << dbase[i].name << dbase[i].birth_year << ' ' <<
                    dbase[i].pay << endl;
                n_man++; mean_pay += dbase[i].pay;
                not_found = false;
            }
        }
        if ( not_found ) cout << " Такого сотрудника нет" << endl; // 10
    }
    if ( n_man > 0 ) cout << " Средний оклад: " << mean_pay / n_man << endl; // 11
}

```

В *операторе 1* файл `dbase.txt` открывается для чтения. Предполагается, что он находится в том же каталоге, что и текст программы, иначе следует указать полный путь. Файл из нескольких строк следует создать в любом текстовом редакторе в соответствии с форматом, заданным в условии задачи (для правильного поиска русского текста он должен иметь кодировку ASCII, см. задачу 5.1). Необходимо предусмотреть случай, когда одна фамилия является частью другой (например, Иванов и Ивановский). В конце файла не должно быть пустых строк. Не забудьте проверить, выдается ли диагностическое сообщение, если файл не найден.

Цикл 2 выполняет построчное считывание из файла в строку `buf` и заполнение очередного элемента массива `dbase`. Счетчик `i` хранит индекс первого свободного элемента массива. Для формирования полей структуры используются функции `strncpy`, `atoi` и `atof`, которые изучались на семинаре 5 (см. с. 72). Обратите внимание, что завершающий нуль-символ в поле фамилии заносится «вручную», поскольку функция `strncpy` делает это только в случае, если строка-источник короче строки-приемника. В функцию `atoi` передается адрес начала подстроки, в которой находится год рождения.

При каждом проходе цикла выполняется проверка, не превышает ли считанное количество строк размерность массива (не забудьте проверить выдачу диагностического сообщения). При тестировании добавьте в цикл контрольный вывод на экран считанной строки и сформированных полей структуры.

ВНИМАНИЕ

При заполнении массива из файла обязательно контролируйте выход за границы массива и при необходимости выдавайте предупреждающее сообщение.

В *операторе 3* задаются переменные `n_record` для хранения фактического количества записей о сотрудниках и `n_man` для подсчета количества сотрудников, о которых

будут выдаваться сведения. Следует не забыть обнулить переменную `mean_pay`, в которой будет накапливаться сумма окладов.

Цикл поиска сотрудников по фамилии организован как бесконечный (*оператор 4*) с принудительным выходом (*оператор 5*). Некоторые специалисты считают, что такой способ является плохим стилем и для выхода из цикла следует определить переменную-флаг, но нам кажется иначе.

Впрочем, в данном случае выход из цикла действительно организован не лучшим образом. Более удобным был бы выход по нажатию, например, клавиши `Esc`. К сожалению, в рамках стандарта это сделать невозможно, однако в состав библиотек Visual Studio.NET входит библиотека консольного ввода-вывода, описанная в заголовочном файле `<conio.h>`, и в ней есть функция `_getch`, позволяющая анализировать код нажатой клавиши. В общем случае, несомненно, следует выбирать интерфейс, наиболее удобный для пользователя, поскольку именно для него и пишутся все без исключения программы.

В *операторе 6* определяется переменная-флаг `not_found` для того, чтобы после окончания цикла поиска было известно, завершился ли он успешно. Обратите внимание на имя переменной: его следует выбирать таким образом, чтобы по нему было ясно, какое значение является истинным. Как видите, в этом случае *оператор 10* хорошо понятен без дополнительных комментариев.

В *операторе 7* организуется цикл просмотра массива структур (просматриваются только заполненные при вводе элементы). Проверка совпадения фамилии выполняется в два этапа. В *операторе 8* с помощью функции `strstr` определяется, содержится ли в поле базы `name` искомая последовательность букв, а в *операторе 9* проверяется, есть ли пробел непосредственно после фамилии (если пробела нет, то искомая фамилия является частью другой и эта строка нам не подходит). Такая простая проверка возможна из-за условия задачи, по которому фамилия должна начинаться с первой позиции строки.

Проверка переменной `n_map` в *операторе 11* необходима для того, чтобы в случае, если пользователь не введет ни одной фамилии, совпадающей с фамилией в базе, не выполнялось деление на 0. Если в базе есть несколько сотрудников с одной и той же фамилией, программа выдаст сведения обо всех.

Крупным недостатком нашей программы является то, что вводить фамилию сотрудника требуется именно в том регистре, в котором она присутствует в базе. Для преодоления этого недостатка необходимо перед сравнением фамилий переводить все символы в один регистр. Для символов латинского алфавита в библиотеке есть функции `tolower` и `toupper`, переводящие переданный им символ в нижний и верхний регистры соответственно, аналогичные функции для символов русского алфавита придется написать самостоятельно.

А теперь давайте рассмотрим вариант записи этой же программы с помощью библиотечных функций ввода-вывода (листинг 6.2).

Листинг 6.2. Поиск в массиве структур с помощью функций в стиле C

```
#include <stdio>
#include <cstring>
```

продолжение ⇨

Листинг 6.2 (продолжение)

```

using namespace std;
int main() {
    const int l_name = 30;                                     // 1
    struct Man {
        int birth_year;
        char name[l_name + 1];
        float pay;
    };
    const int l_dbase = 100;
    Man dbase[l_dbase];
    char name[l_name + 1];
    FILE *fin;
    if ( ( fin = fopen( "dbase.txt", "r" ) ) == NULL ) {
        puts( "Ошибка открытия файла\n" ); return 1; }
    int i = 0;
    while ( !feof ( fin ) ) {
        fgets( dbase[i].name, l_name, fin );
        fscanf( fin, "%i%f\n", &dbase[i].birth_year, &dbase[i].pay ); // 2
        i++;
        if ( i > l_dbase ) { puts( "Слишком длинный файл\n" ); return 1; }
    }
    int n_record = i, n_man = 0;
    float mean_pay = 0;
    while ( true ) {
        puts( "Введите фамилию или нажмите Enter для окончания: " );
        gets( name );
        if ( strlen(name) == 0 ) break; // 3
        bool not_found = true;
        for ( int i = 0; i < n_record; i++ ) {
            if ( strstr( dbase[i].name, name ) )
                if ( dbase[i].name[strlen(name)] == ' ' ) {
                    printf( "%30s%5i%10.2f\n", dbase[i].name, dbase[i].birth_year,
                        dbase[i].pay ); // 4
                    n_man++; mean_pay += dbase[i].pay;
                    not_found = false; }
        }
        if ( not_found ) puts( "Такого сотрудника нет\n" );
    }
    if ( n_man > 0 ) printf( " Средний оклад: %10.2f\n", mean_pay / n_man );
}

```

Из всех именованных констант осталась одна, задающая длину поля фамилии (*оператор 1*). Все остальные определять нет смысла, потому что ввод выполняется непосредственно в поля структуры с помощью функции чтения строки *fgets* и функции форматного ввода *fscanf* (*оператор 2*), которая сама выполняет преобразования подстроки в число, явным образом заданные в предыдущей программе. Мы упростили выход из цикла ввода запросов, теперь для его завершения достаточно нажать клавишу *Enter* (*оператор 3*). Для вывода сведений о сотруднике

использовалась функция `printf` (*оператор 4*). Как видите, когда требуется форматирование разнотипных данных, старые добрые функции могут сделать программу более лаконичной!

Задача 6.2. Сортировка массива структур

Написать программу, которая упорядочивает описанный в предыдущей задаче файл по году рождения сотрудников.

Изменим предыдущую программу таким образом, чтобы она вместо поиска упорядочивала массив, а затем записывала его в исходный файл. Для сортировки применим метод выбора. При всей своей простоте он достаточно эффективен (помните, мы использовали его в задаче 4.3?). Для экономии места в листинге 6.3 опущено считывание базы, совпадающее с листингом 6.1 (до *оператора 3* включительно), и приводится только фрагмент, выполняющий сортировку и вывод.

Листинг 6.3. Сортировка массива структур

```
...
fin.close();
ofstream fout( "dbase.txt" );
if ( !fout ) { cout << " Ошибка открытия файла " << endl; return 1; }
for ( int i = 0; i < n_record - 1; i++) {
    int imin = i;
    for ( int j = i + 1; j < n_record; j++ )
        if ( dbase[j].birth_year < dbase[imin].birth_year ) imin = j;
    Man a      = dbase[i];           // обмен двух элементов массива структур
    dbase[i]    = dbase[imin];
    dbase[imin] = a;
}
for ( int i = 0; i < n_record ; i++ ) {
    fout << dbase[i].name << dbase[i].birth_year << ' ' << dbase[i].pay << endl;
}
fout.close(); cout << "Сортировка базы данных завершена" << endl;
```

Элементами массива в данной задаче являются структуры. Для структур одного типа определена операция присваивания, поэтому обмен двух элементов массива структур выглядит точно так же, как для основных типов данных.

Чтобы записать результаты в тот же файл, файл, открытый для чтения, закрывается, и открывается файл с тем же именем для записи (точнее, создается объект выходного потока `ostream` с именем `fout`). При этом старый файл на диске уничтожается и создается новый пустой файл, куда и производится запись массива.

СОВЕТ

Будьте аккуратны при отладке программ, изменяющих входные файлы: следует либо перед запуском программы создать копию исходного файла, либо открывать выходной файл с другим именем, а заменять его на имя, требуемое по заданию, только после того, как удалось убедиться в полной работоспособности программы.

Задача 6.3. Структуры и бинарные файлы

Написать две программы. Первая считывает информацию из файла, формат которого описан в задаче 6.1, и записывает ее в бинарный файл. Количество записей в файле не ограничено. Вторая программа по номеру записи корректирует оклад сотрудника в этом файле.

Бинарные файлы, то есть файлы, в которых информация хранится во внутренней форме представления, применяются для последующего использования программными средствами. Смотреть на них в текстовом редакторе — занятие неблагоприятное. Преимущества бинарных файлов состоят в том, что при чтении (записи) не тратится время на *преобразование данных* из символьной формы представления во внутреннюю и обратно, и при этом не возникает *потери точности* вещественных чисел. Кроме того, для бинарных файлов применяется *прямой доступ* к информации путем установки текущей позиции указателя. Это дает возможность быстрого получения и изменения отдельных данных файла.

Бинарный файл открывается в двоичном режиме (то есть символы перевода строки воспринимаются не как специальные), а чтение (запись) в него выполняются с помощью функций `fread` и `fwrite`. Обе программы не представляют алгоритмических сложностей, поэтому мы сразу приведем тексты (листинги 6.4 и 6.5). Для краткости работа с файлами выполнена «в стиле C»¹. Хранить в памяти весь файл нет необходимости, вполне достаточно одной переменной структурного типа, в которой будет содержаться в каждый момент времени запись об одном сотруднике.

Листинг 6.4. Создание бинарного файла из текстового

```
#include <stdio>
#include <string>
using namespace std;
int main() {
    const int l_name = 30;
    struct { char name[l_name + 1]; int birth_year; float pay; } man;
    FILE *fin;
    if ( ( fin = fopen( "dbase.txt", "r" ) ) == NULL ) {
        puts( "Ошибка открытия вх. файла\n" ); return 1; }
    FILE *fout;
    if ( ( fout = fopen( "dbase.bin", "wb" ) ) == NULL ) { // b - бинарный режим
        puts( "Ошибка открытия вых. файла\n" ); return 1; }
    while ( !feof( fin ) ) {
        fgets( man.name, l_name, fin );
        fscanf( fin, "%i%f\n", &man.birth_year, &man.pay );
        printf( "%s%i%10.2f\n", man.name, man.birth_year, man.pay ); // отладка
        fwrite( &man, sizeof(man), 1, fout ); // вывод в бинарный файл
    }
    fclose( fout ); puts( "Бинарный файл записан\n" );
}
```

¹ Другие примеры работы с бинарными файлами приведены в задачах 6.5 и 7.4.

Для формирования записей в бинарном файле здесь применяется функция

```
size_t fwrite( const void *p, size_t size, size_t n, FILE *f )
```

Она записывает n элементов длиной $size$ байт из буфера, заданного указателем p , в поток f . Возвращает число записанных элементов. Для чтения из бинарного файла будем применять функцию `fread`:

```
size_t fread( void *p, size_t size, size_t n, FILE *f );
```

Она считывает n элементов длиной $size$ байт в буфер, заданный указателем p , из потока f . Возвращает число фактически считанных элементов.

Листинг 6.5. Корректировка бинарного файла

```
#include <stdio>
#include <cstring>
using namespace std;
int main() {
    const int l_name = 30;
    struct { char name[l_name + 1]; int birth_year; float pay; } man;
    FILE *fout;
    if ( ( fout = fopen( "dbase.bin", "r+b" ) ) == NULL ) {                // 1
        puts( "Ошибка открытия файла\n" ); return 1; }
    fseek( fout, 0, SEEK_END );      // установка текущей позиции файла на его конец
    int n_record = ftell( fout ) / sizeof( man );    // получение длины файла // 2
    int num;
    while ( true ) {                                                        // 3
        puts( "Введите номер записи или -1: " );
        scanf( "%i", &num );
        if ( num < 0 || num >= n_record ) break;
        fseek( fout, num * sizeof( man ), SEEK_SET );
        fread( &man, sizeof( man ), 1, fout );
        printf( "%s%i%10.2f\n", man.name, man.birth_year, man.pay );
        puts( "Введите новый оклад: " );
        scanf( "%f", &man.pay );
        fseek( fout, num * sizeof( man ), SEEK_SET );
        fwrite( &man, sizeof( man ), 1, fout );
        printf( "%s%i%10.2f\n", man.name, man.birth_year, man.pay );
    }
    fclose( fout );  puts( "Корректировка завершена.\n" );
}
```

В *операторе 1* открывается сформированный в предыдущей задаче бинарный файл. Обратите внимание на режим открытия: `r+` обозначает возможность чтения и записи, `b` — двоичный режим. Чтобы проконтролировать введенный номер записи, в переменную `n_record` заносится длина файла в записях (*оператор 2*).

В цикле корректировки оклада (*оператор 3*) текущая позиция в файле устанавливается дважды, поскольку после первого чтения она смещается на размер считанной записи. Выход из цикла выполняется, если будет задан неверный номер записи: меньший нуля или больший, чем их количество.

Задача 6.4. Структуры в динамической памяти

Вывести на экран содержимое бинарного файла, сформированного в предыдущей задаче, упорядочив фамилии сотрудников по алфавиту.

Приведем программу (листинг 6.6) без предварительного обсуждения¹.

Листинг 6.6. Структуры в динамической памяти

```
#include <stdio>
#include <cstring>
#include <stdlib>
const int l_name = 30;
struct Man {
    char name[l_name + 1];
    int birth_year;
    float pay;
};
int compare( const void *man1, const void *man2 );    // 1
int main() {
    FILE *fbin;
    if ( ( fbin = fopen( "dbase.bin", "rb" ) ) == NULL ) {
        puts( "Ошибка открытия файла\n" ); return 1; }
    fseek( fbin, 0, SEEK_END );
    int n_record = ftell( fbin ) / sizeof( Man );    // размер файла в записях
    Man *man = new Man [n_record];    // выделение памяти под весь массив структур
    fseek( fbin, 0, SEEK_SET );    // установка текущей позиции на начало файла
    fread( man, sizeof( Man ), n_record, fbin );    // считывание файла целиком
    fclose( fbin );    // файл больше не потребуется
    qsort( man, n_record, sizeof( Man ), compare );    // сортировка массива структур
    for ( int i = 0; i < n_record; i++ )
        printf( "%s %5i %10.2f\n", man[i].name, man[i].birth_year, man[i].pay );
}
int compare( const void *man1, const void *man2 ) {
    return strcmp( ((Man *)man1)->name, ((Man *)man2)->name );
}
```

Для сортировки мы использовали стандартную функцию `qsort`, прототип которой находится в заголовочном файле `<stdlib>`. Она может выполнять сортировку массивов любых размеров и типов. У нее четыре параметра: указатель на начало области, в которой размещается сортируемая информация; количество сортируемых элементов; размер каждого элемента в байтах; имя функции, которая выполняет сравнение двух элементов.

На следующем семинаре мы подробно рассмотрим правила оформления функций, а сейчас необходимо уяснить следующее: раз функция `qsort` универсальна, мы должны снабдить ее информацией о том, как сравнивать сортируемые элементы и какие выводы делать из сравнения. Значит, мы должны сами написать функцию, которая сравнивает два любых элемента, и передать ее в `qsort`.

¹ Ведь вы все равно его пропустите, правда?

Имя функции может быть любым. Мы назвали ее `compare`. *Оператор 1* представляет собой заголовок (прототип) функции, он необходим компилятору для проверки правильности ее вызова. Для работы `qsort` требуется, чтобы наша функция имела два параметра — указатели на сравниваемые элементы типа `void`. Функция должна возвращать значение, меньшее нуля, если первый элемент меньше второго, равное нулю, если они равны, и большее нуля, если первый элемент больше. При этом массив будет упорядочен по возрастанию. Если нужно упорядочить массив по убыванию, следует изменить возвращаемые значения так: если первый элемент меньше второго, возвращать значение, большее нуля, а если больше — меньшее.

Внутри функции надо привести указатели на `void` к типу указателя на структуру `Man`. Для этого мы использовали операцию приведения типа в стиле C (`Man *`). Более грамотно применять для этого операцию `reinterpret_cast`, описанную в стандарте C++. Функция `compare` с использованием `reinterpret_cast` выглядит вот таким устрашающим образом:

```
int compare( const void *man1, const void *man2 ) {
    return strcmp( ( reinterpret_cast <const Man *> (man1))->name,
                   ( reinterpret_cast <const Man *> (man2))->name );
}
```

Чтобы описание структуры было известно в функции `compare`, описание структуры и необходимой ей константы `l_name` перенесено в глобальную область.

Для упорядочивания массива по другому полю надо изменить функцию сравнения. Вот, например, как она выглядит при сортировке по возрастанию года рождения:

```
int compare( const void *man1, const void *man2 ) {
    int p;
    if      ( ((Man *) man1)->birth_year < ((Man *) man2)->birth_year ) p = -1;
    else if ( ((Man *) man1)->birth_year == ((Man *) man2)->birth_year ) p = 0;
    else                                         p = 1;
    return p;
}
```

Можно записать то же самое более компактно с помощью тернарной условной операции, которую мы рассматривали на втором семинаре (см. с. 31). Для разнообразия приведем функцию для сортировки по убыванию оклада:

```
int compare( const void *man1, const void *man2 ) {
    return ((Man *) man1)->pay > ((Man *) man2)->pay ? -1:
           ((Man *) man1)->pay == ((Man *) man2)->pay ? 0 : 1;
}
```

Как видите, использование стандартных функций сокращает размер программы, но требует некоторых усилий по изучению интерфейса функций.

Итоги

1. Структуры применяются для логического объединения связанных между собой данных различных типов. Элементы структуры называются полями. Поля

могут быть любого основного типа, массивом, указателем, объединением или структурой.

2. Для обращения к полю используется операция выбора: «точка» при обращении через имя структуры и «->» при обращении через указатель. После описания структурного типа ставится точка с запятой.
3. Структуры одного типа можно присваивать друг другу. Ввод-вывод структур выполняется поэлементно.
4. Структуры, память под которые выделяет компилятор, можно инициализировать перечислением значений их элементов.

Задания

Вариант 1

Описать структуру с именем STUDENT, содержащую следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT; записи должны быть упорядочены по возрастанию номера группы;
- ☐ вывод на дисплей фамилий и номеров групп для всех студентов, включенных в массив, если средний балл студента больше 4.0;
- ☐ если таких студентов нет, вывести соответствующее сообщение.

Вариант 2

Описать структуру с именем STUDENT, содержащую следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT; записи должны быть упорядочены по возрастанию среднего балла;
- ☐ вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5;
- ☐ если таких студентов нет, вывести соответствующее сообщение.

Вариант 3

Описать структуру с именем STUDENT, содержащую следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT; записи должны быть упорядочены по алфавиту;
- ☐ вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
- ☐ если таких студентов нет, вывести соответствующее сообщение.

Вариант 4

Описать структуру с именем AEROFLOT, содержащую следующие поля: название пункта назначения рейса; номер рейса; тип самолета. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из семи элементов типа AEROFLOT; записи должны быть упорядочены по возрастанию номера рейса;
- ☐ вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры;
- ☐ если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 5

Описать структуру с именем AEROFLOT, содержащую следующие поля: название пункта назначения рейса; номер рейса; тип самолета. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из семи элементов типа AEROFLOT; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
- ☐ вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
- ☐ если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 6

Описать структуру с именем WORKER, содержащую следующие поля: фамилия и инициалы работника; название занимаемой должности; год поступления на работу. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из десяти структур типа WORKER; записи должны быть размещены по алфавиту;
- ☐ вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
- ☐ если таких работников нет, вывести на дисплей соответствующее сообщение.

Вариант 7

Описать структуру с именем TRAIN, содержащую следующие поля: название пункта назначения; номер поезда; время отправления. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа TRAIN; записи упорядочить по алфавиту по названиям пунктов назначения;
- ☐ вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
- ☐ если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 8

Описать структуру с именем TRAIN, содержащую следующие поля: название пункта назначения; номер поезда; время отправления. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из шести элементов типа TRAIN; записи должны быть упорядочены по времени отправления поезда;
- ☐ вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры;
- ☐ если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 9

Описать структуру с именем TRAIN, содержащую следующие поля: название пункта назначения; номер поезда; время отправления. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа TRAIN; записи должны быть упорядочены по номерам поездов;
- ☐ вывод на экран информации о поезде, номер которого введен с клавиатуры;
- ☐ если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 10

Описать структуру с именем MARSH, содержащую следующие поля: название начального пункта маршрута; название конечного пункта маршрута; номер маршрута. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
- ☐ вывод на экран информации о маршруте, номер которого введен с клавиатуры;
- ☐ если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 11

Описать структуру с именем MARSH, содержащую следующие поля: название начального пункта маршрута; название конечного пункта маршрута; номер маршрута. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
- ☐ вывод на экран информации о маршрутах, которые начинаются или оканчиваются в пункте, название которого введено с клавиатуры;
- ☐ если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 12

Описать структуру с именем NOTE, содержащую следующие поля: фамилия, имя; номер телефона; дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE; записи должны быть упорядочены по датам рождения;

- ☐ вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
- ☐ если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 13

Описать структуру с именем NOTE, содержащую следующие поля: фамилия, имя; номер телефона; дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE; записи должны быть размещены по алфавиту;
- ☐ вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
- ☐ если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 14

Описать структуру с именем NOTE, содержащую следующие поля: фамилия, имя; номер телефона; дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE; записи должны быть упорядочены по трем первым цифрам номера телефона;
- ☐ вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- ☐ если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 15

Описать структуру с именем ZNAK, содержащую следующие поля: фамилия, имя; знак Зодиака; дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK; записи должны быть упорядочены по датам рождения;
- ☐ вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- ☐ если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 16

Описать структуру с именем ZNAK, содержащую следующие поля: фамилия, имя; знак Зодиака; дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK; записи должны быть упорядочены по датам рождения;
- ☐ вывод на экран информации о людях, родившихся под знаком, название которого введено с клавиатуры;
- ☐ если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 17

Описать структуру с именем ZNAK, содержащую следующие поля: фамилия, имя; знак Зодиака; дата рождения (массив из трех чисел). Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ZNAK; записи должны быть упорядочены по знакам Зодиака;
- ☐ вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры;
- ☐ если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 18

Описать структуру с именем PRICE, содержащую следующие поля: название товара; название магазина, в котором продается товар; стоимость товара в руб. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа PRICE; записи разместить в алфавитном порядке по названиям товаров;
- ☐ вывод на экран информации о товаре, название которого введено с клавиатуры;
- ☐ если таких товаров нет, выдать на дисплей соответствующее сообщение.

Вариант 19

Описать структуру с именем PRICE, содержащую следующие поля: название товара; название магазина, в котором продается товар; стоимость товара в руб. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа PRICE; записи разместить в алфавитном порядке по названиям магазинов;
- ☐ вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры;
- ☐ если такого магазина нет, выдать на дисплей соответствующее сообщение.

Вариант 20

Описать структуру с именем ORDER, содержащую следующие поля: расчетный счет плательщика; расчетный счет получателя; перечисляемая сумма в руб. Написать программу, выполняющую следующие действия:

- ☐ ввод с клавиатуры данных в массив, состоящий из восьми элементов типа ORDER; записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков;
- ☐ вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры; если такого расчетного счета нет, выдать на дисплей соответствующее сообщение.

Семинар 7. Функции

Теоретический материал: с. 72–87, 93–96.

Функция — именованная группа операторов, выполняющая законченное действие. К ней можно обратиться по имени, передать ей значения и получить из нее результат.

Функции нужны в первую очередь для упрощения структуры программы. Разбив задачу на части и оформив их в виде функций, мы поступаем по принципу, известному еще с древних времен: «Разделяй и властвуй». Описав функцию, можно использовать ее многократно, передавая ей различные данные.

Чтобы использовать функцию, не нужно знать алгоритм ее работы — достаточно уметь ее вызвать. Для вызова требуется знать ее *интерфейс*, который определяется ее *заголовком*. В заголовке указывается: *имя* функции, *тип результата*, который она возвращает, сколько *аргументов* и какого типа ей нужно передать.

Формат простейшего *заголовка (прототипа)* функции:

тип имя ([список_параметров]);

В квадратных скобках записано то, что может быть опущено. Например, заголовок функции `main` обычно имеет вид `int main()`. Это означает, что никаких параметров этой функции извне не передается, а возвращает она одно значение типа `int` (код завершения). Функция может не возвращать никакого значения, в этом случае должен быть задан тип `void`. Вот, к примеру, заголовок стандартной библиотечной функции, вычисляющей синус угла:

```
double sin( double );
```

Здесь записано, что функция по имени `sin` вычисляет значение синуса типа `double`, и нужно передать ей аргумент типа `double`. А вот заголовок функции `memcpy`, копирующей блок памяти длиной `n` байтов, начиная с адреса `src`, по адресу `dest`:

```
void *memcpy( void *dest, const void *src, size_t n );
```

Эта функция возвращает указатель неопределенного типа на начало области памяти, в которую выполнялось копирование. Какой именно смысл имеет каждый из параметров функции, описывается в документации на функцию. Имена параметров при записи прототипа функции имеют чисто декоративное значение, то есть они могут понадобиться нам, а не компилятору, поэтому их можно опускать:

```
void *memcpy( void *, const void *, size_t );
```

Неграмотно написанная функция наряду с аргументами использует и глобальные переменные, которые, как вам известно, доступны из любого блока текущего файла¹. Поскольку это никак не отражается на заголовке, для использования такой функции требуется исследовать ее текст. Представьте, что, прежде чем позвонить по телефону, вам нужно было бы рассмотреть все его внутренности, чтобы убедиться, что красный провод не идет к взрывному устройству! Надеемся, что этот устрашающий пример сразу убедит вас не использовать в функциях глобальные переменные.

ВНИМАНИЕ

Все, что передается в функцию и обратно, должно отражаться в ее заголовке. Это требование не синтаксиса, а хорошего стиля.

Заголовок задает *объявление* функции. В программе может содержаться произвольное количество *объявлений* одной и той же функции и только одно *определение* (в этом функции не отличаются от других программных объектов). *Определение* функции, кроме заголовка, включает ее *тело*, то есть те операторы, которые выполняются при вызове функции:

```
int sum( int a, int b ) {                               // функция находит сумму двух значений
    return a + b;                                       // тело функции
}
```

Тело функции представляет собой *блок*, заключенный в фигурные скобки. Для *возврата результата*, вычисленного в функции, служит оператор `return`. После него указывается выражение, результат вычисления которого передается в точку вызова функции. Результат при необходимости преобразуется по общим правилам к типу, указанному в заголовке. Функция может иметь несколько операторов возврата, это определяется алгоритмом.

Для *вызова функции* надо указать ее *имя* (все как в жизни, например, «Ихтиандр!» или «Леопольд!»), а также передать ей *набор аргументов* в соответствии с указанным в ее заголовке. Соответствие должно соблюдаться строго, и это естественно: ведь если в заголовке функции указано, сколько величин и какого типа ей требуется для успешной работы, значит, их надо ей передать.

Вызов функции, имеющей тип `void`, записывается отдельным оператором, а функции, возвращающей значение определенного типа, может быть записан в любом месте, где по синтаксису допустимо выражение, например:

```
double y, x1 = 0.34, x2 = 2;
y = sin( x1 );                                           // вызов в правой части оператора присваивания
cout << y << ' ' << sin( x2 ) << endl;                 // вызов в цепочке вывода
y = sin( x1 + 0.5 ) - sin( x1 - 0.5 );                  // вызовы в составе выражения
char *cite = "Never say never";
char b[100];
memset( b, cite, strlen( cite ) + 1 );                 // вызов в отдельном операторе
```

¹ Кроме блоков, в которых описаны локальные переменные с такими же именами.

```
int a = 2;
int summa = sum( a, 4 );
```

// вызов в выражении инициализации

ВНИМАНИЕ

В определении, в объявлении и при вызове функции типы и порядок следования параметров должны совпадать. Для имен параметров соответствия не требуется.

Задача 7.1. Передача в функцию параметров стандартных типов

Написать программу вывода таблицы значений функции $\cosh x$ (гиперболический косинус) для аргумента, изменяющегося в заданных пределах с заданным шагом. Значения функции вычислять с помощью разложения в ряд Тейлора с точностью ϵ .

На втором семинаре мы уже рассматривали подобные задачи (см. задачи 2.4 и 2.5), поэтому принцип вычисления суммы ряда вам знаком. Алгоритм работы программы также приводился ранее: для каждого из серии значений аргумента вычисляется и затем выводится на экран значение функции. Очевидно, что подсчет суммы ряда для одного значения аргумента логично оформить в виде отдельной функции.

ВНИМАНИЕ

Разработка любой функции ведется в том же порядке, что и разработка программы в целом. Сначала определяется интерфейс функции, то есть какие значения подаются ей на вход и что должно получиться в результате. Затем продумываются структуры данных, в которых будут храниться эти и промежуточные значения; затем составляется алгоритм, программа и тестовые примеры.

Нашей функции подсчета суммы ряда требуется получить извне значение аргумента и точность. Пусть эти величины, а также результат имеют тип `double`. Следовательно, заголовок функции может выглядеть так:

```
double cosh( double x, double eps );
```

Для вычисления суммы ряда (см. задачу 2.5) необходимы промежуточные переменные для хранения очередного члена ряда и его номера. Они описываются внутри функции, поскольку вне ее они не нужны (листинг 7.1).

Листинг 7.1. Передача в функцию параметров стандартных типов

```
#include <stdio>
#include <cmath>
using namespace std;
double cosh( double x, double eps );    // прототип функции
int main() {
    double Xn, Xk, dX, eps;
    printf( "Enter Xn, Xk, dX, eps \n" );
    scanf( "%lf%lf%lf%lf", &Xn, &Xk, &dX, &eps );
    printf( "|      X      |      Y      |\n" );
```

продолжение ➤

Листинг 7.1 (продолжение)

```

    for ( double x = Xn; x <= Xk; x += dX )
        printf( "|%9.2lf    |%14.6g    |\n", x, cosh( x, eps ) );
}
double cosh( double x, double eps ) {
    const int MaxIter = 500;                // максимальное количество итераций
    double ch = 1, y = ch;                  // первый член ряда и нач. значение суммы
    for ( int n = 0; fabs(ch) > eps; n++ ) {
        ch *= x * x / ( ( 2 * n + 1 ) * ( 2 * n + 2 ) ); // член ряда
        y += ch;                                     // добавление члена ряда к сумме
        if ( n > MaxIter ) { puts( "Ряд расходится!\n" ); return 0; }
    }
    return y;
}

```

При использовании функции программа получилась более ясной и компактной, так как задача была разделена на две: вычисление функции и печать таблицы. Кроме того, функцию можно перенести в другую программу или поместить в библиотеку.

Если определение функции размещается после ее вызова, то перед функцией, в которой он выполняется, размещают прототип (заголовок). Обычно заголовки всех функций размещают в самом начале файла или в отдельном заголовочном файле¹. Заголовок нужен для того, чтобы компилятор мог проверить правильность вызова функции. Стандартные заголовочные файлы, которые мы подключаем к программам, содержат прототипы функций библиотеки именно с этой целью.

При написании нашей функции возникает проблема, как сигнализировать о том, что ряд расходится. Давайте отвлечемся от конкретной функции и рассмотрим существующие способы решения проблемы *получения из подпрограммы признака ее аварийного завершения*. Каждый из них имеет свои плюсы и минусы.

Во-первых, можно поступить так, как сделано в листинге 7.1: *вывести сообщение*, сформировать какое-либо значение функции (чаще всего это 0) и выйти из функции. Недостаток этого способа — печать сообщения внутри функции. Это нежелательно, а порой (например, когда функция входит в состав библиотеки) и вовсе недопустимо.

Более грамотное решение — сформировать в функции и передать наружу *признак успешного завершения* подсчета суммы, который должен анализироваться в вызывающей программе. Такой подход часто применяется в стандартных функциях. В качестве признака используется либо возвращаемое значение, которое не входит в множество допустимых (например, отрицательное число при поиске номера элемента в массиве или ноль для указателя), либо отдельный параметр ошибки.

Обычно параметр ошибки — это целое, ненулевые значения которого сигнализируют о различных ошибках в функции. Если ошибка может произойти всего одна, параметру можно назначить тип `bool`. Параметр передается в вызывающую программу и там анализируется. Для нашей задачи подобное решение выглядит так:

¹ В задаче 7.7 мы расскажем о том, как оформлять заголовочные файлы.

```

...
double cosh( double x, double eps, int &err );
int main() {
    double Xn, Xk, dX, eps, y;
    int err;
    ... // здесь ввод данных и вывод шапки таблицы
    for ( double x = Xn; x <= Xk; x += dX ) {
        y = cosh( x, eps, err );
        ...
    }
    double cosh( double x, double eps, int &err ) {
        err = 0;
        ...
        if ( n > MaxIter ) { err = 1; return 0; }
    }
    return y;
}

```

Недостатком этого метода является увеличение количества параметров функции. Да и программа, использующая функцию, тоже несколько усложнилась! Обратите внимание на знак & перед параметром `err`. Это признак передачи параметра по ссылке, что позволяет передавать значения из функции в вызывающую программу.

Сейчас самое время рассмотреть *механизм передачи параметров в функцию*. Он весьма прост. Когда мы пишем в списке параметров функции выражение вида `double x`, это значит, что в функцию при ее вызове должно быть передано значение соответствующего аргумента. Для этого в стеке создается его копия, с которой и работает функция. Естественно, что изменение копии не может оказать никакого влияния на ячейку памяти, в которой хранится сам параметр. Кстати, именно поэтому на месте такого параметра можно при вызове задавать и выражение:

```
y = cosh( x + 0.2, eps / 100, err );
```

Выражение вычисляется, и его результат записывается в стек на место, выделенное для соответствующего параметра.

Ссылка, синтаксически являясь синонимом имени некоторого объекта, в то же время содержит его адрес. Поэтому ссылку, в отличие от указателя, не требуется разадресовывать для получения значения объекта. Если мы передаем в функцию ссылку, то есть пишем в списке параметров выражение вида `double &eps`, а при вызове подставляем на его место аргумент, например, `eps_fact`, мы тем самым передаем в функцию адрес переменной `eps_fact`. Этот адрес обрабатывается так же, как и остальные параметры: в стеке создается его копия. Функция, работая с копией адреса, имеет доступ к ячейке памяти, в которой хранится значение переменной `eps_fact`, и, тем самым, может его изменить. Вот и все!

Можно передать в функцию и указатель, в этом случае придется применять операции разадресации и взятия адреса явным образом. Для нашей функции применение указателя для передачи третьего параметра будет выглядеть так:

```
double cosh( double x, double eps, int *err );           // прототип функции
...
```

продолжение ➤

```

y = cosh(x, eps, &err);           // вызов функции.   & - взятие адреса
...
*err = 0;                         // обращение к err внутри функции.  * - разадресация

```

Как видите, в прототипе (и, конечно, в определении функции) явным образом указывается, что третьим параметром будет указатель на целое. При вызове на его место передается адрес переменной `err`. Чтобы внутри функции изменить значение этой переменной, применяется операция получения значения по адресу.

Итак, мы видим, что для *входных данных* функции используется передача параметров по значению, для *передачи результатов* ее работы — возвращаемое значение и (или) передача параметров по ссылке или указателю. На самом деле у передачи по значению есть недостаток: для размещения в стеке копии данных большого размера (например, структур, состоящих из многих полей) тратится и время на копирование, и место. Кроме того, стек может просто переполниться. Поэтому более безопасный, эффективный и грамотный способ — передавать входные данные по ссылке, да не по простой, а по константной, чтобы исключить возможность непреднамеренного изменения параметра в функции. Для нашего примера *передача входных данных по константной ссылке* выглядит так:

```

double cosh( const double &x, const double &eps, int &err );   // прототип функции
...
y = cosh( x, eps, err );                                       // вызов функции
// способ обращения к x и eps внутри функции не изменяется

```

Указание перед параметром ключевого слова `const` при передаче по значению применяется только для того, чтобы четко указать, какие из параметров являются входными. В случае передачи по ссылке это, кроме того, дает возможность передавать на место этого параметра константу. Поскольку вопрос о передаче параметров очень важен, не поленимся повторить изложенное еще раз.

ВНИМАНИЕ

Входные данные передают в функцию по значению или по константной ссылке, *результаты* работы — через возвращаемое значение, а при необходимости передачи более одной величины — через параметры по ссылке или указателю.

Вернемся к обсуждению способов сообщения об ошибках внутри функции. Еще один способ — *передавать параметр ошибки через возвращаемое значение*. Например, функция стандартной библиотеки `int fputc(int ch, FILE *f)` записывает символ `ch` в поток `f`. При ошибке она возвращает значение EOF, иначе — записанный символ. При необходимости передать в точку вызова какие-либо другие результаты работы функции их передают через список параметров.

Часто в функциях библиотеки в случае возникновения ошибки применяется и более простое решение: при ошибке возвращается значение, равное 0, хотя ноль может входить в множество допустимых значений результата. В этом случае у нас нет средств отличить ошибочное значение от правильного. Например, таким образом реализованы известные вам функции `atoi`, `atol` и `atof`. При невозможности

преобразовать строку в число соответствующего типа они возвращают ноль, и то же самое значение будет выдано в случае, если в строке содержался символ 0.

Во второй части практикума мы рассмотрим еще один механизм уведомления о возникновении ошибки — *генерацию исключения*, а сейчас подведем итоги.

При написании функции нужно предусмотреть все возможные ошибки и обеспечить пользователя функции средствами их диагностики. Печать диагностических сообщений внутри функции нежелательна.

Теперь давайте немножко упростим себе жизнь, воспользовавшись средством C++, называемым *значениями параметров по умолчанию*. Может оказаться неудобным каждый раз при вызове функции `cosh` задавать требуемую точность вычисления суммы ряда. Конечно, можно определить точность в виде константы внутри функции, задав максимальное допустимое значение, но иногда это может оказаться излишним, поэтому желательно сохранить возможность задания точности через параметры. Для этого либо в определении (если оно находится выше по тексту, чем любой вызов функции), либо в прототипе функции после имени параметра указывается его значение по умолчанию, например:

```
double cosh( double x, double eps = DBL_EPSILON );
```

Константа `DBL_EPSILON` определена в файле `<float>`. Ее значение равно минимальному числу, которое, будучи прибавлено к единице, даст не равный единице результат. Теперь нашу функцию можно вызывать с одним параметром:

```
y = cosh(x);
```

Функция может иметь несколько параметров со значениями по умолчанию. Они должны находиться в конце списка параметров.

Давайте посмотрим на нашу программу с другой стороны (с рассмотренных она нам уже порядком надоела). Мы оформили в виде функции вычисление суммы ряда, однако и задача вывода таблицы значений функции сама по себе типична. Поэтому логично оформить и ее решение в виде функции.

Задача 7.1-а. Передача в функцию имени функции

Оформить в виде функции вывод на экран таблицы значений произвольной функции, применить разработанную функцию к предыдущей программе.

Назовем функцию вывода таблицы значений `print_tabl`. Прежде всего определим ее интерфейс. Чтобы вывести таблицу, нашей функции потребуется знать диапазон и шаг изменения значений аргумента, а также какую, собственно, функцию мы собираемся вычислять. Кроме того, поскольку в функцию вычисления суммы ряда надо передавать точность, ее следует включить в список параметров вызывающей функции `print_tabl`. Никакого значения эта функция не возвращает (перед ее именем надо указать `void`).

Имя функции передается в функцию точно так же, как любая другая величина: в списке параметров перед именем параметра указывается его тип. До этого момента мы передавали в функцию величины стандартных типов, а теперь нам потребуется

определить собственный. *Тип функции* определяется типами ее возвращаемого значения и параметров. Для нашей функции это выглядит так:

```
double ( *fun )( double, double );
```

Здесь описывается указатель по имени fun на функцию, получающую два аргумента типа double и возвращающую значение того же типа (от параметров по умолчанию нам, к сожалению, придется отказаться). Часто, если описание типа сложное, с целью улучшения читабельности программы для него задают синоним с помощью ключевого слова typedef:

```
typedef double ( *Pfun )( double, double );
```

В этом операторе задается тип Pfun, который можно использовать наряду со стандартными типами при описании переменных. Таким образом, заголовок функции печати таблицы должен иметь примерно такой вид:

```
void print_tabl( Pfun fun, double Xn, double Xk, double dX, double eps );
```

Запишем теперь текст программы (листинг 7.2), для экономии места сведя к минимуму диагностику ошибок: при превышении допустимого количества итераций функция завершается, возвращая 0, а вызывающая программа выводит это значение. Функция print_tabl может выводить таблицу значений любой функции, принимающей два аргумента типа double и возвращающей значение того же типа.

Листинг 7.2. Передача в функцию имени функции

```
#include <stdio>
#include <cmath>
using namespace std;
typedef double ( *Pfun )( const double, const double );
void print_tabl( Pfun fun, const double Xn, const double Xk,
                const double dX, const double eps );
double cosh( const double x, const double eps );
int main() {
    double Xn, Xk, dX, eps;
    printf( "Enter Xn, Xk, dX, eps \n" );
    scanf( "%lf%lf%lf%lf", &Xn, &Xk, &dX, &eps );
    print_tabl( cosh, Xn, Xk, dX, eps );
}
void print_tabl( Pfun fun, const double Xn, const double Xk,
                const double dX, const double eps ) {
    printf( " ----- \n" );
    printf( "|      X      |      Y      | \n" );
    printf( " ----- \n" );
    for ( double x = Xn; x <= Xk; x += dX )
        printf( "|%9.2lf    |%14.6g    | \n", x, fun( x, eps ) );
    printf( " ----- \n" );
}
double cosh( const double x, const double eps ) {
    // ... здесь вычисление суммы ряда
}
```

Как видите, наряду с большей общностью мы добились лучшего структурирования программы, разбив ее на две логически не связанные подзадачи: вычисление функции и вывод таблицы.

Задача 7.2. Передача массивов в функцию

Даны два массива из n целых чисел каждый. Определить, в каком из них больше положительных элементов.

Очевидно, что для решения этой задачи требуется подсчитать одну и ту же величину в двух массивах. Следовательно, эти действия надо поместить в функцию. Ее входные данные — массив и количество его элементов, результат — количество положительных элементов в массиве:

```
int n_posit( const int *a, const int n );
```

Имя массива — это указатель на его нулевой элемент, поэтому в функцию массивы передаются через указатели¹. Количество элементов массива передается отдельным параметром, потому что, в отличие от строк, использующих признак конца строки, для массивов общего вида признака конца не существует. Аналогичные задачи рассматривались на семинаре 3, поэтому сразу приведем программу (листинг 7.3).

Листинг 7.3. Передача в функцию одномерных массивов

```
#include <iostream>
using namespace std;
int n_posit( const int *a, const int n );           // прототип функции
int main() {
    int n;
    cout << "Введите количество элементов: "; cin >> n;
    int *a = new int[n];
    int *b = new int[n];
    cout << "Введите элементы первого массива: ";
    for ( int i = 0; i < n; i++ ) cin >> a[i];
    cout << "Введите элементы второго массива: ";
    for ( int i = 0; i < n; i++ ) cin >> b[i];
    if ( n_posit( a, n ) > n_posit( b, n ) )
        cout << " В первом положительных больше\n";
    else if ( n_posit( a, n ) < n_posit( b, n ) )
        cout << " Во втором положительных больше\n";
    else    cout << " Одинаковое количество\n";
}
int n_posit( const int *a, const int n ) {
    int count = 0;
    for ( int i = 0; i < n; i++ ) if ( a[i] > 0 ) count++;
    return count;
}
```

¹ Это же относится и к именам функций, передаваемых в функции. Все остальные величины могут быть переданы по значению.

Здесь место под массивы выделяется в динамической области памяти, однако функцию можно без изменений применять и для «обычного» массива, потому что его имя тоже является указателем на нулевой элемент, только константным.

Рассмотрим способ анализа результатов работы функции. Как видите, функцию приходится вызывать для каждого массива дважды, что для больших массивов, конечно, нерационально. Чтобы этого избежать, можно завести две переменные для хранения результатов обработки обоих массивов:

```
int n_posit_a = n_posit( a, n );
int n_posit_b = n_posit( b, n );
if      ( n_posit_a > n_posit_b ) cout << " В первом положительных больше\n";
else if ( n_posit_a < n_posit_b ) cout << " Во втором положительных больше\n";
else      cout << " Одинаковое количество\n";
```

Надо заметить, что современные компиляторы обладают широкими возможностями и сами оптимизируют подобные ситуации, но это не означает, что программист вообще не должен обращать внимание на эффективность.

Мы не получили выигрыша в длине программы, используя функцию, но причина лишь простота задания. Обычно рекомендуется, чтобы размер функции не превышал два экрана. Главное, чтобы каждая функция решала только одну задачу.

Двумерные массивы передаются в функции аналогичным образом, например:

```
int fun ( int **a, const int m, const int n );           // прототип функции
int main() {
    int m, n; cin >> m >> n;                             // ввод размерностей массива
    int **a = new int *[m];                               // выделение памяти
    for ( int i = 0; i < m; i++ ) a[i] = new int [n];      // выделение памяти
    ...                                                    // здесь ввод массива
    int x = fun( a, m, n );                                // вызов функции
    ...
}
int fun( int **a, const int m, const int n ) {
    ... // здесь текст функции
}
```

Задача 7.3. Передача строк в функцию

Написать программу, определяющую, сколько чисел содержится в каждой строке текстового файла. Длина каждой строки не превышает 100 символов.

Эту задачу можно разбить на две: ввод данных из файла и их анализ. Для каждой строки проверка выполняется отдельно, поэтому в виде функции логично оформить поиск и подсчет количества чисел в одной строке. На вход функции будем подавать строку, а на выходе получать количество чисел в этой строке.

Конец строки будем определять по нуль-символу внутри функции. Для простоты предположим, что числа могут быть либо целые, либо вещественные с фиксированной точкой и непустой дробной частью (листинг 7.4).

Листинг 7.4. Передача строк в функцию

```

#include <fstream>
#include <iostream>
#include <cctype>
using namespace std;
int num_num( const char *str );
int main() {
    ifstream fin( "test.txt" );
    if ( !fin ) { cout << "Нет файла test.txt" << endl; return 1; }
    const int len = 101;
    char      str[len];
    int       i = 1;
    while ( fin.getline( str, len ) ) {
        cout << "В строке " << i << " содержится " << num_num(str) << " чисел\n ";
        i++; }
}
int num_num( const char *str ) {
    int count = 0;
    while ( *str ) {
        if ( isdigit( *str ) && ! isdigit( *(str + 1) ) && *(str + 1) != '.' )
            count++;
        str++; }
    return count;
}

```

Увеличение счетчика чисел в функции `num_num` происходит каждый раз, когда заканчивается число, то есть если после цифры стоит не цифра и не точка. Цикл заканчивается по достижении нуль-символа.

Задача 7.4. Передача структур в функцию

Написать программу дополнения бинарного файла, сформированного в задаче 6.3, вводимыми с клавиатуры сведениями о сотрудниках.

Эту задачу можно разбить на две части: ввод сведений о сотрудниках в структуру и ее добавление в бинарный файл, поэтому в нашей программе будет две функции. Первая возвращает сформированную структуру, ничего не получая извне. Вторая получает структуру и имя файла и возвращает признак успешности добавления. Для проверки правильности занесения данных в бинарный файл напомним еще одну функцию, которая будет по введенному номеру записи выводить ее на экран. Текст программы приведен в листинге 7.5.

Листинг 7.5. Передача структур в функцию

```

#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
const int l_name = 30;

```

продолжение ➤

Листинг 7.5 (продолжение)

```

    struct Man {                // структура для хранения сведений об одном сотруднике
        char name[l_name + 1];
        int birth_year;
        float pay;
    };
Man read_data();
int append2binfile( const Man &man, const char* filename );
int print_from_bin( const char * filename );
int main() {
    bool contin; char y_n;
    char filename[] = "dbase.bin";
    do{ contin = false;
        append2binfile( read_data(), filename );
        cout << " Продолжить (y/n)? "; cin.get(y_n); cin.get();
        if ( ( y_n == 'y' ) || ( y_n == 'Y' ) ) contin = true;
    } while ( contin );
    print_from_bin( filename );
}
int append2binfile( const Man &man, const char* filename ) {
    ofstream fout (filename, ios::binary|ios::app); // бинарный файл для добавления
    if ( !fout ) { cout << "Ошибка открытия файла." << endl; return 1; }
    fout.write( (char *) &man, sizeof( man ) );
    fout.close();
    return 0;
}
int print_from_bin( const char * filename ) {
    Man man;
    ifstream fin( filename, ios::binary ); // бинарный файл для чтения
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; return 1; }
    fin.seekg( 0, ios::end ); // установка позиции на конец файла
    int n_records = fin.tellg() / sizeof ( man ); // длина файла в записях
    while ( true ) {
        int num = 0;
        cout << "Введите номер записи или -1: "; cin >> num;
        if ( num < 0 || num >= n_records ) break;
        fin.seekg( num * sizeof( man ) ); // абсолютное смещение на нужную запись
        fin.read( (char*) &man, sizeof( man ) );
        cout << man.name << man.birth_year << ' ' << man.pay << endl;
    }
    return 0;
}
Man read_data() {
    char name[l_name + 1];
    cout << "Введите фамилию И.О. "; cin.getline( name, l_name + 1 );
    if ( strlen( name ) < l_name )
        for ( int i = strlen( name ); i < l_name; i++ ) name[i] = ' ';
    name[l_name] = 0;
    Man man;

```

```
strncpy( man.name, name, l_name + 1 );
const int buf_len = 80;
char buf[buf_len];
do { cout << "Введите год рождения "; cin.getline( buf, buf_len ); }
while ( ( man.birth_year = atoi( buf ) ) == 0 );           // 1
do { cout << "Введите оклад "; cin.getline( buf, buf_len ); }
while ( !( man.pay = atof( buf ) ) );                     // 2
return man;
}
```

Работа с бинарным файлом здесь выполняется с помощью классов. Как видите, двоичный файл, созданный с помощью функций библиотеки C (<cstdio>), можно считывать и изменять с помощью объектно-ориентированных средств, и наоборот.

В функции ввода `read_data` выполняется заполнение пробелами оставшейся части строковой переменной `name`, чтобы ее формат соответствовал формату имени в файле. Обратите внимание, как производится проверка правильности ввода числовой информации. Чтение выполняется в буферную строку, которая затем преобразуется с помощью функций `atoi` и `atof` в числа. Если функции возвращают 0, преобразование выполнить не удалось (например, вместо цифр были введены буквы), и информация запрашивается повторно. Условие повторения циклов 1 и 2 записано в двух разных вариантах, чтобы вы могли оценить, какой из них вам более понятен (профессионалы предпочли бы второй, более лаконичный вариант).

Надо заметить, что программа из листинга 7.5 содержит целый ряд не связанных с передачей структур в функции уязвимостей, которые делают ее совершенно непригодной для практического использования. Они связаны с возможным превышением размеров буферов ввода (`y_n` в функции `main`, `name` и `buf` в функции `read_data`). Чтобы «сломать» эту программу, достаточно ввести большее количество символов, чем может вместить буфер. Именно поэтому функции работы со строками «старого стиля» считаются устаревшими, и рекомендуется пользоваться классом `string`, которому посвящен семинар 14.

В этой задаче структура передавалась в функцию по константной ссылке. Можно передавать ее и по значению, что несколько хуже, потому что тогда затрачивается время на копирование и требуется дополнительное место в стеке параметров. В отличие от массива, структура может быть возвращаемым значением функции.

Задача 7.5. Рекурсивные функции

Написать программу упорядочивания массива методом быстрой сортировки, используя рекурсию.

Пришло время выполнить обещание, данное в задаче 3.3, — показать рекурсивную реализацию метода быстрой сортировки. Говоря упрощенно, *рекурсивной* называется функция, в которой имеется обращение к ней самой. При быстрой сортировке используется *процедура разделения*, применяемая к фрагменту массива. На каждом шаге образуются две половинки текущего фрагмента, и к ним снова применяется процедура разделения. То есть по своей сути алгоритм является рекурсивным, и если не здесь применять рекурсивные функции, то где?..

Любая функция в программе на C++ может вызываться рекурсивно. При этом в стеке выделяется новый участок памяти для размещения копий параметров, а также автоматических и регистровых переменных, поэтому предыдущее состояние выполняемой функции сохраняется и к нему впоследствии можно вернуться (так и будет, если программа не зависнет). Одна из возможных версий программы сортировки приведена в листинге 7.6.

Листинг 7.6. Рекурсивная функция

```
#include <iostream>
using namespace std;
void qsort( float* array, int left, int right );
int main() {
    const int n = 10;
    float    arr[n];
    cout << "Введите элементы массива: ";
    for ( int i = 0; i < n; i++ ) cin >> arr[i];
    int l = 0, r = n - 1;           // левая и правая границы начального фрагмента
    qsort( arr, l, r );             // 1
    for ( int i = 0; i < n; i++ ) cout << arr[i] << ' ';
}
void qsort( float* array, int left, int right ) {
    int i = left, j = right;
    float middle = array[( left + right ) / 2];
    float temp;
    while ( i < j ) {
        while ( array[i] < middle ) i++;
        while ( middle < array[j] ) j--;
        if ( i <= j ) {
            temp = array[i]; array[i] = array[j]; array[j] = temp;           // обмен
            i++; j--;
        }
    }
    if ( left < j ) qsort( array, left, j );           // 2
    if ( i < right ) qsort( array, i, right );         // 3
}
```

Процедура разделения реализована здесь в виде рекурсивно вызываемой функции `qsort`, в теле которой есть два обращения к самой себе: в *операторе 2* — для сортировки левой половины текущего фрагмента, и в *операторе 3* — для его правой половины. Надеемся, что сравнение этой программы с предыдущей версией (задача 3.3) доставит вам истинное эстетическое наслаждение.

У рекурсии есть и недостатки: программу *труднее отлаживать* (требуется контролировать глубину рекурсии), при большой глубине *стек может переполниться*, а кроме того, увеличиваются *накладные расходы*. Например, в данном случае в стеке сохраняются не два числа, являющиеся границами фрагмента, а гораздо больше, не говоря уже о затратах, связанных с вызовом функции. Поэтому рекурсию при всей ее красоте следует применять с осторожностью.

Многофайловые проекты

До сих пор мы писали программы, исходный текст которых размещался в одном файле. Однако реальные задачи требуют создания многофайловых проектов с распределением решаемых подзадач по разным модулям (файлам).

ПРИМЕЧАНИЕ

Интегрированная среда Microsoft Visual Studio 2005 поддерживает создание, компиляцию и сборку многофайловых проектов. С технологией их создания вы можете ознакомиться по Приложению, а здесь мы рассмотрим вопросы, связанные с распределением функций по модулям, разделением интерфейса и реализации и особенностями использования глобальных переменных. Теоретический материал по директивам препроцессора см. в Учебнике на с. 93–96.

Напомним, что, пользуясь технологией нисходящего проектирования программ, мы разбиваем исходную задачу на подзадачи, затем при необходимости каждая из них также разбивается на подзадачи, и так далее, пока решение очередной подзадачи не окажется достаточно простым, то есть реализуемым в виде функции обозримого размера (предпочтительным считается размер не более двух экранов редактора).

Исходные тексты совокупности функций для решения какой-либо подзадачи, как правило, размещаются в отдельном модуле (файле). Такой файл называют *исходным (source file)*. Обычно он имеет расширение `.c` или `.cpp`. Прототипы всех функций исходного файла выносят в отдельный *заголовочный файл (header file)*, для него принято использовать расширение `.h` или `.hpp`.

Таким образом, заголовочный файл `xxx.h` содержит *интерфейс* для некоторого набора функций, а исходный файл `xxx.cpp` содержит *реализацию* этого набора. Если некоторая функция из этого набора вызывается из какого-то другого исходного модуля `ууу.cpp`, то необходимо включить в этот модуль заголовочный файл `xxx.h` с помощью директивы `#include`. Негласное правило стиля программирования на C++ требует включения этого же заголовочного файла и в исходный файл `xxx.cpp`.

Теперь о *глобальных переменных*. В многофайловом проекте возможны два «вида глобальности». Если глобальная переменная объявлена в файле `xxx.cpp` с модификатором `static`, она видима от точки определения до конца этого файла, то есть область ее видимости ограничена файлом. Если же глобальная переменная объявлена в файле `xxx.cpp` без модификатора `static`, то она может быть видимой в пределах всего проекта. Правда, для того, чтобы она оказалась видимой в другом файле, необходимо иметь в этом файле ее объявление с модификатором `extern` (рекомендуется поместить это объявление в файл `xxx.h`).

Что и как следует размещать в заголовочном файле

В заголовочном файле принято размещать:

- ☐ определения типов, задаваемых пользователем, констант, шаблонов;
- ☐ объявления (прототипы) функций;

- ❑ объявления внешних глобальных переменных (с модификатором `extern`);
- ❑ пространства имен¹.

Теперь обратим ваше внимание на проблему *повторного включения заголовочных файлов*. Она может возникнуть при иерархическом проектировании структур данных, когда в один заголовочный файл включается другой заголовочный файл (например, для использования типов, определенных в этом файле). Впрочем, лучше рассмотреть эту проблему на конкретном примере.

Ниже приведены тексты простой многофайловой программы, в которой определены типы данных «Точка» (структура `Point` в файле `Point.h`) и «Прямоугольник» (структура `Rect` в файле `Rect.h`). Поскольку второй тип данных определяется через первый, в файле `Rect.h` имеется директива `#include "Point.h"`.

В основном модуле `main.cpp` создается объект типа «Прямоугольник» (под «объектом» здесь понимается переменная типа `struct`) и выводятся координаты его двух углов. В основном модуле используются как функции из модуля `Point.cpp`, так и функции из модуля `Rect.cpp`, поэтому в него включены оба заголовочных файла, `Point.h` и `Rect.h`. Но после обработки этих директив препроцессором окажется, что структура `Point` определена дважды. В результате компилятор выдаст сообщение об ошибке вроде: `«error . . . : 'Point': 'struct' type redefinition»`:

```

//////////////////////////////////// Файл Point.h //////////////////////////////////////
// Объявления типов
struct Point { int x, y; };
// Прототипы функций
void SetXY( Point& point, int x, int y );
int  GetX( Point& point );
int  GetY( Point& point );
//////////////////////////////////// Файл Rect.h //////////////////////////////////////
#include "Point.h"
// Объявления типов
struct Rect { Point leftTop, rightBottom; };
// Прототипы функций
void SetLTRB( Rect& rect, Point lt, Point rb );
void GetLT( Rect& rect, Point& lt );
void GetRB( Rect& rect, Point& rb );
//////////////////////////////////// Файл Point.cpp //////////////////////////////////////
#include "Point.h"
void SetXY( Point& point, int x, int y ) { point.x = x; point.y = y; }
int  GetX( Point& point ) { return point.x; }
int  GetY( Point& point ) { return point.y; }
//////////////////////////////////// Файл Rect.cpp //////////////////////////////////////
#include "Rect.h"
void SetLTRB( Rect& rect, Point lt, Point rb ) {
    rect.leftTop    = lt;
    rect.rightBottom = rb;
}

```

¹ Пространства имен рассматриваются в Учебнике на с. 99.

```

void GetLT( Rect& rect, Point& lt ) { lt = rect.leftTop; }
void GetRB( Rect& rect, Point& rb ) { rb = rect.rightBottom; }
//////////////////// Файл Main.cpp //////////////////////
#include <stdio.h>
#include "Point.h"
#include "Rect.h"
int main() {
    Point pt1, pt2, lt, rb;
    Rect rect1;
    SetXY( pt1, 2, 5 ); SetXY( pt2, 10, 14 );
    SetLTRB( rect1, pt1, pt2 );
    GetLT( rect1, lt ); GetRB( rect1, rb );
    printf( "rect.lt.x = %d, rect.lt.y = %d\n", lt.x, lt.y );
    printf( "rect.rb.x = %d, rect.rb.y = %d\n", rb.x, rb.y );
}

```

Каков же выход из этой ситуации? Бьерн Страуструп рекомендует использовать так называемые *стражи включения*, и этот способ нашел широкое применение. Он состоит в следующем: чтобы предотвратить повторное включение заголовочных файлов, содержимое каждого .h-файла должно находиться между директивами условной компиляции `#ifndef` и `#endif`, как описано ниже:

```

#ifndef FILENAME_H
#define FILENAME_H
/* содержимое заголовочного файла */
#endif /* FILENAME_H */

```

Для нашего примера файл `Point.h` должен содержать следующий текст:

```

//////////////////// Файл Point.h //////////////////////
#ifndef POINT_H
#define POINT_H

// Объявления типов
struct Point { int x, y; };

// Прототипы функций
void SetXY( Point& point, int x, int y );
int  GetX( Point& point );
int  GetY( Point& point );
#endif /* POINT_H */

```

Задача 7.6. Многофайловый проект — форматирование текста

Написать программу форматирования текста, читаемого из файла `unformt.txt` и состоящего из строк ограниченной длины. Слова в строке разделены произвольным количеством пробелов. Программа должна читать входной файл по строкам, форматировать каждую строку и выводить результат в выходной файл `formatd.txt`. Форматирование заключается в выравнивании границ текста слева и справа путем равномерного распределения пробелов между соседними словами, а также в отступе с левой стороны страницы на `margin` позиций (результатирующий текст должен

находиться в позициях `margin + 1 .. margin + maxl_line`). Программа должна также подсчитать общее количество слов в тексте.

На этом примере показана технология разработки многофайловых проектов.

Алгоритм решения задачи не представляет особой сложности:

1. Открыть входной файл.
2. Читать файл построчно в текстовый буфер `line`, попутно удаляя возможные пробелы в начале строки (до первого слова).
3. Для каждой строки `line` выполнить следующие действия:
 - Вычислить величину интервала (количество пробелов), которую необходимо обеспечить между соседними словами для равномерного распределения слов в пределах строки.
 - Вывести каждое слово из строки `line` в выходной файл, вставляя между словами необходимое количество пробелов и одновременно увеличивая счетчик слов на единицу.
4. После обработки последней строки входного файла вывести на экран значение счетчика слов и закрыть выходной файл.

Разбиение на подзадачи. В результате детализации описанного алгоритма определим спецификации функций.

```
void DefInter ( const char* pline, int & base_int, int & add_int, int& inter );
```

определяет для строки, на которую указывает `pline`, количество межсловных промежутков `inter`, требуемую величину основного интервала `base_int` для каждого промежутка (количество пробелов) и величину дополнительного интервала `add_int`, определяемую как остаток от деления общего количества пробелов в строке на количество межсловных промежутков; последняя величина должна быть равномерно распределена путем добавления одного пробела в каждый из первых `add_int` промежутков.

```
void GetLine ( FILE* finp, char* pline );
```

читает очередную строку из входного файла в массив символов с адресом `pline`, ликвидируя при этом пробелы в начале строки.

```
void PutInterval ( FILE* fout, const int k );
```

выводит очередной интервал, состоящий из `k` пробелов.

```
int PutWord ( FILE* fout, const char* pline, const int startpos );
```

выводит очередное слово в выходной файл, начиная с позиции `startpos` текущей строки `pline`; возвращает номер позиции в строке `pline`, следующей за последним переданным символом, или 0, если достигнут конец строки.

```
int SearchNextWord ( const char* pline, const int curpos );
```

возвращает номер позиции, с которой начинается следующее слово в строке `pline`, или 0, если достигнут конец строки (поиск начинается с позиции `curpos`).

Разбиение на модули. Наша программа будет располагаться в двух исходных файлах: task7_7.cpp — с функцией main, edit.cpp — с реализацией перечисленных выше функций, а также заголовочный файл edit.h с интерфейсом этих функций. В листинге 7.7 приводится содержимое этих файлов.

Листинг 7.7. Многофайловый проект — форматирование текста

```

//////////////////// Файл Task7_7.cpp //////////////////////
#include <stdio>
#include <cstring>
#include <stdlib>
#include "edit.h"
using namespace std;
const int maxl_line = 63;
const int margin = 5;
int main() {
    printf("Работает программа Task7_7.\n");
    FILE* finp;
    if ( ! ( finp = fopen( "unformt.txt", "r" ) ) ) {
        puts( "Файл unformt.txt не найден.\n" ); exit(0); }
    puts( "Читается файл unformt.txt.\n" );
    FILE* fout;
    if ( ! ( fout = fopen( "formatd.txt", "w" ) ) ) {
        puts( "Файл formatd.txt не создан.\n" ); exit(0); }
    puts( "Выполняется запись в файл formatd.txt.\n" );
    char line[maxl_line + 1];
    int base_i, add_i, inter;
    int nword = 0;
    while ( GetLine( finp, line ) ) {
        DefInter( line, base_i, add_i, inter );
        PutInterval( fout, margin );
        int next = PutWord( fout, line, 0, nword );
        for ( int i = 0; i < inter; i++ ) {
            int start = SearchNextWord( line, next );
            PutInterval( fout, base_i );
            if ( add_i ) { add_i--; PutInterval( fout, 1 ); }
            next = PutWord( fout, line, start, nword );
            if ( !next ) break;
        }
        fprintf( fout, "\n" );
    }
    printf( "\nКоличество слов - %d\n", nword );
    fclose( fout );
    printf( "Работа завершена.\n" );
    return 0;
}
//////////////////// Файл Edit.h //////////////////////
void DefInter(const char* pline, int& base_int, int& add_int, int& inter);
int GetLine(FILE*, char*);
void PutInterval(FILE*, const int);

```

продолжение ➤

Листинг 7.7 (продолжение)

```

int PutWord(FILE*, const char*, const int, int&);
int SearchNextWord(const char*, const int);
extern const int maxl_line;
////////// Файл Edit.cpp //////////////////////////////////////////
#include <stdio>
#include <cstring>
#include "edit.h"
using namespace std;
int GetLine( FILE* finp, char* pline ) {
    int i = 0;
    char c;
    while ( ( c = fgetc( finp ) ) == ' ' ) i++;
    if( c == EOF ) return 0;
    fseek( finp, -1, SEEK_CUR );
    fgets( pline, maxl_line - i + 1, finp );
    pline[strlen(pline) - 1] = 0;
    return 1;
}
int SearchNextWord( const char* pline, const int curpos ) {
    int i = curpos;
    while( pline[i] != ' ' ) {
        if ( pline[i] == '\n' ) return 0;
        i++; }
    while ( pline[i] == ' ' && pline[i + 1] == ' ' ) i++;
    return i + 1;
}
void DefInter ( const char* pline, int& base_int, int& add_int, int& inter ) {
    int k = 0, end = strlen( pline ) - 1;
    while ( (pline[end] == ' ') || (pline[end] == '\n') || (pline[end] == '\r') )
        end--;
    inter = 0;
    for ( unsigned int i = 0; i < end; i++ ) {
        if ( pline[i] == ' ' ) {
            k++;
            if ( pline[i + 1] != ' ' ) inter++;
        }
    }
    int blank_amount = k + maxl_line - end;
    if ( !k ) { base_int = 0; add_int = 0; }
    else { base_int = blank_amount / inter; add_int = blank_amount % inter; }
    return;
}
int PutWord ( FILE* fout, const char* pline, const int startpos, int& n ) {
    int i = startpos;
    char c;
    n++;
    while ( ( c = pline[i++] ) != ' ' ) {
        fprintf( fout, "%c", c );
    }
}

```

```

        if ( ( c == '\n' ) || ( c == '\0' ) ) { i = 0; break; }
    }
    return i - 1;
}

void PutInterval( FILE* fout, const int k ) {
    for ( int i = 0; i < k; i++ ) fprintf( fout, " " );
    return;
}

```

Константу `maxline` следует задавать большей, чем максимальная длина строки исходного файла. В качестве упражнения измените программу так, чтобы можно было форматировать текст и в более узкую колонку, чем в исходном файле.

Приведем результаты тестирования этой программы. Входной файл `unformt.txt`¹:

23. Не терпеть и малого своего недостатка - вот признак совершенства. От изъянов духовных и телесных редко кто свободен, но часто их лелеют, когда от них легко бы исцелиться. Вчуже досадно видеть

разумному, как ничтожный изъян порой портит великолепное сочетание достоинств, - довольно и облачка, чтобы затмить солнце. Родимые пятна на доброй славе злоба людская сразу подметит - и упорно в них метит. Особенно ценно искусство скрывать свой недостаток, обращая его в преимущество.

Так, Цезарь скрывал свою плешь лавровым венком.

Содержимое выходного файла `formatd.txt`:

23. Не терпеть и малого своего недостатка - вот признак совершенства. От изъянов духовных и телесных редко кто свободен, но часто их лелеют, когда от них легко бы исцелиться. Вчуже досадно видеть разумному, как ничтожный изъян порой портит великолепное сочетание достоинств, - довольно и облачка, чтобы затмить солнце. Родимые пятна на доброй славе злоба людская сразу подметит - и упорно в них метит. Особенно ценно искусство скрывать свой недостаток, обращая его в преимущество. Так, Цезарь скрывал свою плешь лавровым венком.

Протестируйте эту программу на других текстах.

Итоги

1. Функция — это именованная последовательность операторов, выполняющая законченное действие. Функции нужны для упрощения структуры программы. Интерфейс грамотно написанной функции определяется ее заголовком.
2. Для вызова функции надо указать ее имя и набор аргументов. В определении, в объявлении и при вызове функции типы и порядок следования аргументов и параметров должны совпадать.

¹ Текст взят из афоризмов Грасиана (1647 г.)

3. Передача параметров в функцию может выполняться по значению или по адресу. Входные данные функции надо передавать по значению или по константной ссылке, результаты ее работы — через возвращаемое значение, а при необходимости передать более одной величины — через параметры по ссылке или указателю.
4. При написании функции нужно предусмотреть все возможные ошибки и обеспечить пользователя функции средствами их диагностики. Печать диагностических сообщений внутри функции нежелательна.
5. Функция может иметь несколько параметров со значениями по умолчанию. Они должны находиться в конце списка параметров.
6. Массивы всегда передаются в функцию по адресу. Количество элементов в массиве должно передаваться отдельным параметром.
7. Рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь. При использовании рекурсии следует учитывать возникающие при этом проблемы и накладные расходы.
8. В многофайловых проектах важно грамотно разбить задачу на подзадачи и распределить функции по файлам. Для предотвращения ошибок повторного включения заголовочных файлов следует использовать стражи включения.

Задания

Функции и массивы

Выполнить задания третьего семинара («Одномерные массивы») и четвертого семинара («Двумерные массивы»), оформив каждый пункт задания в виде функции. Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Функции, строки и файлы

Выполнить задания пятого семинара («Строки и файлы»), оформив в виде функций законченные последовательности действий. Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается.

Функции, структуры и бинарные файлы

Выполнить задания, приведенные в Учебнике на с. 151 (раздел «Функции и файлы») и на с. 165 (раздел «Модульное программирование»).

Семинар 8. Перегрузка и шаблоны функций

Теоретический материал: с. 83–87.

Перегрузка функций

Перегрузкой функций называется использование нескольких функций с одним и тем же именем, но с различными списками параметров. Перегруженные функции должны отличаться друг от друга либо типом хотя бы одного параметра, либо количеством параметров, либо тем и другим одновременно. Перегрузка является видом полиморфизма и применяется в тех случаях, когда одно и то же по смыслу действие реализуется по-разному для различных типов или структур данных. Компилятор сам определяет, какой именно вариант функции вызвать, по списку аргументов.

Небольшие перегруженные функции удобно применять при отладке программ. Допустим, вам требуется промежуточная печать различного вида: в одном месте требуется выводить на экран структуру, в другом — пару целых величин с пояснениями или вещественный массив. Забота о ясной промежуточной печати — это забота о себе, любимом, при отладке программы, а что может быть важнее? Поэтому проще сразу оформить печать в виде функций, например, таких:

```
void print( char* str, const int i, const int j ) { // 1
    cout << str << '|' << oct << setw(4) << i << '|' << setw(4) << j << '|' << endl;
}
void print( float mas[], const int n ) { // 2
    cout << "Массив:" << endl;
    cout.setf( ios::fixed ); cout.precision( 2 );
    for ( int i = 0; i < n; i++ ) {
        cout << mas[i] << " "; if ( ( i + 1 ) % 4 == 0 ) cout << endl;
    }
    cout << endl;
}
void print( Man m ) { // 3
    cout.setf( ios::fixed ); cout.precision( 2 );
    cout << setw(40) << m.name << ' ' << m.birth_year << ' ' << m.pay << endl;
}
```

В первой из этих функций на экран выводятся строка и два целых числа в восьмеричной форме, разделенные для читабельности вертикальными черточками. Под

каждое число отводится по 4 позиции (манипулятор `setw` устанавливает ширину одного следующего за ним поля).

Во второй функции для вывода вещественных значений по четыре числа на строке задается вывод с фиксированной точкой и точностью два знака после запятой. Для этого используются методы установки флагов `setf`, установки точности `precision` и константа `fixed`, определенная в классе `ios`. Точность касается только вещественных чисел, ее действие продолжается до следующей установки.

Третья функция выводит поля знакомой нам по семинару 6 структуры так, чтобы они не склеивались между собой. Вызов этих функций может выглядеть, например, так:

```
print( "После цикла ", i, n );      print( a, n );      print( m );
```

По имени функции сразу понятно, что она делает, кроме того, при необходимости вызов функции легче закомментировать или перенести в другое место, чем группу операторов печати. Конечно, промежуточная печать — не единственный метод отладки, но зато универсальный, потому что отладчик не всегда доступен.

При написании перегруженных функций основное внимание следует обращать на то, чтобы в процессе поиска нужного варианта функции по ее вызову не возникало неоднозначности. *Неоднозначность* может возникнуть по нескольким причинам. Во-первых, *из-за преобразований типов*, которые компилятор выполняет по умолчанию. Правила преобразования арифметических типов аналогичны описанным в Учебнике на с. 390. Их смысл сводится к тому, что более короткие типы преобразуются в более длинные. Если соответствие между формальными параметрами и аргументами функции на одном и том же этапе может быть получено более чем одним способом, вызов считается неоднозначным и выдается сообщение об ошибке. Неоднозначность может также возникнуть *из-за параметров по умолчанию и ссылок*. Рассмотрим создание перегруженных функций на примере.

Задача 8.1. Перегрузка функций

Написать программу, которая для базы сотрудников, описанной в задаче 6.1, выдает по запросу список сотрудников, либо родившихся раньше заданного года, либо имеющих оклад больше введенного с клавиатуры.

Варианты выборки из базы по различным запросам оформим в виде перегруженных функций. Мы от природной лени и для простоты рассматриваем базу с минимальным количеством полей, в реальных ситуациях их может быть гораздо больше, соответственно, больше будет и вариантов перегруженных функций. Оформим также в виде функции чтение базы из файла — для лучшего структурирования программы и чтобы в случае необходимости было легче заменить эту функцию на другую, например, на чтение из бинарного файла (листинг 8.1). Заметим, что эта программа так же, как и предыдущие, содержит уязвимости.

Листинг 8.1. Перегрузка функций

```
#include <fstream>
#include <iostream>
#include <cstring>
```

```

#include <cstdlib>
#include <iomanip>
using namespace std;
const int l_name = 30, l_year = 5;
struct Man {
    char name[l_name + 1];
    int birth_year;
    float pay;
};
int read_dbase( const char * filename, Man dbase[], const int l_dbase,
               int &n_record );
void print( Man m );
void select( Man dbase[], const int n_record, const int year );
void select( Man dbase[], const int n_record, const float pay );
int main() { // ----- Главная функция -----
    const int l_dbase = 100;
    Man dbase[l_dbase];
    int n_record = 0;
    if ( read_dbase( "dbase.txt", dbase, l_dbase, n_record ) != 0 ) return 1;
    int option, year;
    float pay;
    do {
        cout << "-----" << endl;
        cout << "1 - Сведения по году рождения" << endl;
        cout << "2 - Сведения по окладу" << endl;
        cout << "3 - Выход" << endl;
        cin >> option;
        switch ( option ) {
            case 1: cout << "Введите год "; cin >> year;
                    select( dbase, n_record, year );      break;
            case 2: cout << "Введите оклад "; cin >> pay;
                    select( dbase, n_record, pay );        break;
            case 3: return 0;
            default: cout << "Надо вводить число от 1 до 3" << endl;
        }
    } while ( true );
}
void select( Man dbase[], const int n_record, const int year ) {
    cout << " Вывод сведений по году рождения" << endl;
    bool success = false;
    for ( int i = 0; i < n_record; i++ )
        if ( dbase[i].birth_year >= year ) { print (dbase[i]); success = true; }
    if ( !success ) cout << " Таких сотрудников нет" << endl;
}
void select( Man dbase[], const int n_record, const float pay ){
    cout << " Вывод сведений по окладу" << endl;
    bool success = false;
    for ( int i = 0; i < n_record; i++ )
        if ( dbase[i].pay >= pay ) { print (dbase[i]); success = true; }
}

```

продолжение ➤

Листинг 8.1 (продолжение)

```

    if ( !success ) cout << " Таких сотрудников нет" << endl;
}
void print( Man m ) {
    cout.setf( ios::fixed ); cout.precision( 2 );
    cout << setw(40) << m.name << ' ' << m.birth_year << ' ' << m.pay << endl;
}
int read_dbase( const char * filename, Man dbase[], const int l_dbase,
               int &n_record ) {
    char buf [l_buf + 1];
    ifstream fin( filename );
    if ( !fin ) { cout << "Нет файла " << filename << endl; return 1; }
    int i = 0;
    while ( fin.getline( buf, l_buf ) ) {
        strncpy( dbase[i].name, buf, l_name );
        dbase[i].name[l_name] = '\0';
        dbase[i].birth_year = atoi( &buf[l_name] );
        dbase[i].pay = atof( &buf[l_name + l_year] );
        i++;
        if ( i > l_dbase ) { cout << "Слишком длинный файл"; return 2; }
    }
    n_record = i;
    fin.close();
    return 0;
}

```

Ниже приведены *правила описания перегруженных функций*.

- ❑ Перегруженные функции должны находиться в одной области видимости, иначе произойдет сокрытие аналогично одинаковым именам переменных во вложенных блоках.
- ❑ Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию.

Функции не могут быть перегружены, если описание их параметров отличается только модификатором `const` или использованием ссылки.

Шаблоны функций

Если алгоритм не зависит от типа данных, лучше реализовать его не в виде группы перегруженных функций для различных типов, а в виде шаблона функции. В этом случае компилятор сам сгенерирует текст функции для конкретных типов данных, с которыми выполняется вызов, и программисту не придется поддерживать несколько практически одинаковых функций.

Таким образом, области применения перегрузки функций и шаблонов различаются: перегруженные функции применяют для оформления действий, аналогичных по названию, но различных по реализации, а шаблоны — для идентичных действий над данными различных типов.

Шаблон функции определяется следующим образом:

```
template <class Type> тип имя ([ список_параметров ])
{ /* тело функции */ }
```

Идентификатор Type, задающий так называемый *параметризованный тип*, может использоваться и в остальной части заголовка, и в теле функции. Параметризованный тип — это фиктивное имя, которое компилятор автоматически заменит именем реального типа данных при создании конкретной версии функции. В общем случае шаблон функции может содержать несколько параметризованных типов:

```
<class Type1, class Type2, class Type3, ... >
```

Процесс создания конкретной версии функции называется *инстанцированием шаблона*, или *созданием экземпляра* функции. Возможны два способа инстанцирования: *явный*, когда объявляется заголовок функции, в котором все параметризованные типы заменены на конкретные типы, известные в этот момент в программе, и *неявный*, когда создание экземпляра функции происходит автоматически, если встречается фактический вызов функции.

Чтобы вы прониклись разнообразием возможностей C++, упомянем, что шаблоны тоже можно перегружать, причем как шаблонами, так и обычными функциями.

Задача 8.2. Шаблоны функций

Написать программу, которая определяет максимальные элементы в одномерных массивах различных арифметических типов.

Поиск максимума — весьма распространенная задача, и желание сделать для этого универсальную функцию естественно. Для этого достаточно простейшего шаблона с одним параметром-типом. В функцию будет передаваться два аргумента: указатель на массив и длина этого массива (листинг 8.2).

Листинг 8.2. Шаблоны функций

```
#include <iostream>
#include <cstring>
using namespace std;
template <class T> T Max( T *b, int n );
int main() {
    const int n = 20;
    int b[n];
    cout << "Введите " << n << " целых чисел:" << endl;
    for ( int i = 0; i < n; i++ ) cin >> b[i];
    cout << Max( b, n ) << endl;           // инстанцирование для целого массива
    double a[] = { 0.22, 117.2, -0.08, 0.21, 42.5 };
    cout << Max( a, 5 ) << endl;           // инстанцирование для вещественного массива
    char *str = "Sophisticated fantastic template";
    cout << Max( str, strlen(str) ) << endl; // инстанцирование для строки
}
```

продолжение ➤

Листинг 8.2 (продолжение)

```
template <class T> T Max( T *b, int n ) {  
    int imax = 0;  
    for ( int i = 1; i < n; i++ ) if ( b[i] > b[imax] ) imax = i;  
    return b[imax];  
}
```

Шаблон функции имеет имя Max. После ключевого слова `template` в угловых скобках перечисляются все параметры шаблона. В данном случае параметр один. При инстанцировании шаблона (в данном случае — неявном), то есть когда компилятор будет создавать конкретный вариант функции, этот тип будет заменен конкретным стандартным или пользовательским типом. Соответствие устанавливается при вызове функции либо по типу аргументов, либо по явным образом указанному типу. Например, последний вызов функции можно записать так:

```
cout << Max<char>( str, strlen(str) ); // явное инстанцирование
```

Этот способ применяется тогда, когда тип не определяется по виду вызова функции. Аналогично обычным параметрам функции можно задавать значение параметра шаблона по умолчанию.

ВНИМАНИЕ

При работе с многофайловым проектом нужно не забывать, что если некий шаблон функции имеет инстанцирование в нескольких исходных файлах, то определение этого шаблона должно повторяться в каждом из этих файлов. Поэтому обычно определение шаблона выносят в заголовочный файл и подключают его в нужных местах директивой `#include`.

Итоги

1. Перегрузкой функций называется использование нескольких функций с одним именем и различными типами параметров. Перегрузка применяется, когда одно и то же по смыслу действие реализуется по-разному для различных типов или структур данных.
2. При написании перегруженных функций необходимо, чтобы в процессе поиска нужного варианта функции по ее вызову не возникало неоднозначности. Неоднозначность может возникнуть из-за преобразований типов, параметров по умолчанию и ссылок.
3. Функции не могут быть перегружены, если описания их параметров различаются только модификатором `const` или использованием ссылки.
4. Шаблоны функций применяются для записи идентичных действий над данными различных типов. Инстанцирование шаблона функции — это создание компилятором конкретного варианта функции.
5. Шаблоны можно перегружать как шаблонами, так и обычными функциями.

Задания

Выполнить задания третьего и четвертого семинаров, оформив каждый пункт задания в виде шаблона функции. Все необходимые данные для функций должны передаваться им в качестве параметров. Использование глобальных переменных в функциях не допускается. Привести примеры программ, использующих эти шаблоны для типов `int`, `float` и `double`.

Семинар 9. Динамические структуры данных

Теоретический материал: с. 114–127.

На предыдущих семинарах мы рассматривали задачи, в которых необходимый для хранения данных объем памяти был известен либо до компиляции программы, либо до начала ввода данных. В первом случае память резервировалась с помощью операторов описания, во втором — с помощью функций выделения памяти. В обоих случаях выделялся непрерывный участок памяти.

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память выделяется по мере необходимости отдельными блоками, связанными друг с другом при помощи указателей. Такой способ организации данных называется *динамическими структурами данных*, поскольку их размер изменяется во время выполнения программы. Из динамических структур в программах чаще всего используются различные линейные списки, стеки, очереди и бинарные деревья. Они различаются способами связи элементов и допустимыми операциями над ними. Динамическая структура может занимать несмежные участки оперативной памяти. В процессе работы программы элементы структуры могут по мере необходимости добавляться и удаляться.

Элемент любой динамической структуры данных состоит из *полей*, часть из которых предназначена для связи с соседними элементами. Вот, например, как выглядит описание элемента линейного списка для хранения целых чисел:

```
struct Node {  
    int d;           // Информационная часть (целое число)  
    Node *p;        // Указатель на следующий элемент  
};
```

В зависимости от решаемой задачи в программах применяются различные виды динамических структур. Рассмотрим их на примерах.

Задача 9.1. Стек

Написать программу сортировки вещественного массива из n элементов методом быстрой сортировки.

«Что-то подобное я уже здесь читал...» — подумаете вы, и будете правы. Вы это читали в задачах 3.3 и 7.5. А может быть, вы даже подумаете: «Да сколько ж можно!» Или еще хуже. И вот тут мы с вами не согласимся, потому что, когда мы решали эту задачу в первый раз, нам пришлось выделить для хранения границ неотсортированных фрагментов массива два массива и большая часть этой памяти пропадала совершенно зря. Но это еще полбеды, а настоящая беда может случиться, если массив для сортировки будет настолько длинным, что еще два выделить просто не удастся. И тут нам на помощь приходит динамическое выделение памяти — выделяется нужное количество байтов в нужное время, и ничего лишнего.

Стеком называется структура данных, в которой элемент, занесенный первым, извлекается последним. В алгоритме быстрой сортировки стек используется для хранения границ неупорядоченных фрагментов. В принципе, порядок, в котором они будут обрабатываться, не критичен (главное, чтобы в конце концов все фрагменты оказались отсортированными), но стек удобно использовать из-за простоты его реализации. Для стека определены всего две операции: занесение элемента и выборка элемента. При выборке элемент удаляется из стека, и это как раз то, что требуется. Для работы со стеком достаточно одной переменной — указателя на его вершину. Назовем ее *top*. Каждый элемент стека содержит два целых числа (левую и правую границы фрагмента массива) и указатель на следующий элемент:

```
struct Node {                      // Элемент стека
    int left, right;
    Node* p;
};
Node* top = 0;                    // Вершина стека1
```

Удобно оформить занесение и выборку элемента стека в виде отдельных функций. Функция помещения в стек обычно называется *push*, а выборки — *pop*. Все необходимое передается функциям через параметры:

```
Node* push( Node* top, const int l, const int r ) {           // Занесение в стек
    Node* pv = new Node;                                     // 1
    pv->left = l;                                           // 2
    pv->right = r;                                           // 3
    pv->p = top;                                           // 4
    return pv;                                           // 5
}
Node* pop( Node* top, int& l, int& r ) {                       // Выборка из стека
    Node* pv = top->p;                                     // 6
    l = top->left;                                           // 7
    r = top->right;                                           // 8
    delete top;                                           // 9
    return pv;                                           // 10
}
```

¹ В C++ определена стандартная константа `NULL`, значение которой равно нулю типа «указатель», но, поскольку правила преобразования типов обеспечивают правильное преобразование целого нуля в нуль типа «указатель», в ней нет необходимости.

Тщательно рассмотрим эти функции. Чтобы *занести в стек* границы фрагмента, надо передать в функцию `push` эти границы, а также указатель на вершину стека, в который мы собираемся их заносить. Перед границами указано ключевое слово `const`, чтобы подчеркнуть факт, что они не должны изменяться внутри функции.

Прежде всего мы описываем вспомогательную переменную-указатель и заносим в нее адрес нового элемента стека, который создается с помощью операции `new` (*оператор 1*). Выделяется столько памяти, сколько необходимо для хранения структуры типа `Node`. В *операторах 2 и 3* информационные поля этой структуры `left` и `right` заполняются значениями переданных в функцию границ фрагмента массива. Доступ к полям выполняется через указатель `pv` и операцию выбора `->`. Новый элемент становится вершиной стека. Поле его указателя должно ссылаться на элемент, помещенный в стек ранее. Эта ссылка создается в *операторе 4*. Если «заталкиваемый» в стек элемент является первым, то в качестве первого аргумента функции `push` надо задать `0`. Функция `push` возвращает указатель на вершину стека. Им всегда является указатель на только что занесенный элемент (*оператор 5*).

Выборка из стека (функция `pop`) выполняется аналогично. Сначала из вершины стека выбирается указатель на его следующий элемент (*оператор 6*), который станет новой вершиной стека. Этот указатель является возвращаемым значением функции (*оператор 10*). Информационная часть элемента заносится в переменные `l` и `r`, которые передаются в вызывающую функцию по ссылке (*операторы 7 и 8*). После того, как вся информация из элемента выбрана, его можно удалить (*оператор 9*).

Эти функции можно применить в любой программе, где требуется стек, изменив поля, составляющие его информационную часть, и соответствующие параметры. Теперь можно заменить в задаче 3.3 массивы на «классический» стек (листинг 9.1).

Листинг 9.1. Быстрая сортировка с использованием классического стека

```
#include <fstream>
#include <iostream>
using namespace std;
struct Node {
    int left, right;
    Node* p;
};
Node* push( Node* top, const int l, const int r );
Node* pop( Node* top, int &l, int &r );
// ----- главная функция -----
int main() {
    ifstream fin( "sort1.txt" );
    if ( !fin ) { cout << "Нет файла sort1.txt" << endl; return 1; }
    int n; fin >> n;
    float* arr = new float[n];
    for ( int i = 0; i < n; i++ ) fin >> arr[i];
    int left, right;
    Node* top = 0;
    top = push( top, 0, n - 1 );
    // ----- Сортировка
```

```

while ( top ) {
    top = pop( top, left, right );
    while ( left < right ) {          // Разделение < arr[left] .. arr[right] >
        int i = left, j = right;
        float middle = arr[( left + right ) / 2];
        while ( i < j ) {
            while ( arr[i] < middle ) i++;
            while ( middle < arr[j] ) j--;
            if ( i <= j ) {
                float temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
                i++; j--;
            }
        }
        if ( i < right )              // Запись в стек запроса из правой части
            top = push( top, i, right );
        right = j;                    // Теперь left и right ограничивают левую часть
    }
}
for ( int i = 0; i < n; i++ ) cout << arr[i] << ' ';          // Вывод результата
}
Node* push(Node* top, const int l, const int r) {              // ----- Занесение в стек
    Node* pv = new Node;
    pv->left = l; pv->right = r; pv->p = top;
    return pv;
}
Node* pop(Node* top, int &l, int &r) {                          // ----- Выборка из стека
    Node* pv = top->p;
    l = top->left; r = top->right; delete top;
    return pv;
}

```

Для большей общности здесь использован ввод из файла в динамический массив.

Задача 9.2. Линейный список

Написать программу работы с базой отдела кадров предприятия. База хранится в текстовом файле, его размер может быть произвольным. Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия и инициалы (30 поз., фамилия должна начинаться с первой позиции), год рождения (5 поз.), оклад (10 поз.). Программа должна обеспечивать поиск в базе по заданным критериям, корректировку и дополнение базы.

У вас, несомненно, еще свежо отвращение к аналогичным задачам, рассмотренным на семинарах 6 и 7. Специфика этой задачи состоит в том, что размер базы не ограничен, поэтому для ее хранения в памяти мы воспользуемся односвязным линейным списком. Каждый элемент такого списка включает информационную часть (фамилию, год рождения, оклад) и указатель на следующий элемент. Признаком

конца списка служит нуль в поле указателя. Список доступен через указатель на его первый элемент:

```
const int l_name = 31;
struct Man {           // ----- Элемент списка -----
    char name[l_name]; int birth_day; float pay;           // Информационная часть
    Man *next;           // Указатель на следующий элемент
};
Man *beg;               // Указатель на начало списка
```

Основное внимание при программировании этой задачи уделим разбиению на функции и спецификации их интерфейсов. Например, логично оформить в виде функции каждую операцию со списком (формирование, поиск, добавление и удаление элемента), поскольку они представляют собой законченные действия.

Интерфейс пользователя организуем в виде простейшего меню, которое будет выводиться на экран после каждого действия (в стандарт С++ не входят функции позиционирования курсора на экране, управления цветом и работы с экраном в графическом режиме, поскольку они системно-зависимые, поэтому наши возможности ограничены). Будем исходить из того, что все функции должны быть независимы, чтобы изменения в одной функции не могли влиять на поведение другой. Для этого всё, что функциям необходимо получать извне, будем передавать им через параметры.

Прежде всего определим интерфейс нашей программы. Нам кажется логичным предоставить пользователю следующие возможности: 1) добавление сотрудника; 2) удаление сотрудника; 3) поиск сотрудника; 4) корректировка сведений о сотруднике; 5) вывод базы на экран; 6) вывод базы в файл. Каждый пункт этого меню оформим в виде функции. Попытаемся путем рассуждений определить их интерфейсы.

Добавление сотрудника. Чтобы добавить сотрудника в базу, надо знать, кого и куда добавлять. Иными словами, в функцию надо передать указатель на начало списка и собственно добавляемый элемент. Чтобы функция могла добавлять в список и самый первый элемент, она должна возвращать указатель на начало списка:

```
Man* add( Man *beg, const Man &man );
```

Удаление сотрудника. Чтобы удалить сотрудника из базы, надо знать, из какой базы и какого сотрудника удалять. Удаление сотрудника логично выполнять по его фамилии и инициалам, следовательно, надо их предварительно запросить у пользователя. Чтобы удалить элемент из списка, надо предварительно найти этот элемент, то есть путем просмотра списка получить указатель на него.

Поиск элемента списка по фамилии потребует в нашей программе и сам по себе, и для корректировки списка, поэтому оформим его в виде отдельной функции. *Запрос фамилии* требуется и при удалении сотрудника, и при корректировке сведений, поэтому его также будет логично оформить в виде совсем небольшой вспомогательной функции. Назовем ее `get_name` (впрочем, без этой функции вполне можно обойтись, это дело вкуса):

```
void get_name( char *name );
```

Итак, запрос фамилии будет выполняться внутри функции удаления, поэтому извне в нее должен передаваться только указатель на начало списка:

```
Man* remove( Man *beg );
```

Эта функция, как и функция добавления, возвращает указатель на начало списка на случай, если удаление выполняется из его начала (при этом указатель изменяется).

Поиск сотрудника. В условии задачи сказано, что поиск должен выполняться по заданным критериям. В нашей примитивной базе всего три поля. Будем выполнять поиск по каждому из них. Для примера примем, что нам могут потребоваться:

- ☐ сведения об отдельном сотруднике по его фамилии и инициалам;
- ☐ сведения о сотрудниках старше заданного года рождения;
- ☐ сведения о сотрудниках, имеющих оклад больше введенного с клавиатуры.

Режим работы функции поиска — это ее внутреннее дело, поэтому извне ей передается только указатель на начало списка:

```
void find_man( Man *beg );
```

Поиск в списке по конкретному критерию оформим в виде перегруженных функций, которые будут вызываться из `find_man`:

```
Man* find( Man *beg, char *name, Man *&prev ); // поиск по фамилии
void find( Man *beg, int birth_day );           // поиск по году рождения
void find( Man *beg, float pay );               // поиск по окладу
```

Мы добавили в список параметров первой функции указатель на элемент, предшествующий найденному (`prev`). Он передается по адресу (как ссылка на указатель), поскольку должен формироваться внутри функции. Этот указатель потребуется в подпрограмме удаления, поскольку при этом надо связать между собой предыдущий и следующий по отношению к удаляемому элементы.

Корректировка сведений. Функция аналогична функции удаления за исключением того, что указатель на начало списка измениться не может, поэтому функция возвращает признак успешности выполнения корректировки (на всякий случай):

```
int edit( Man *beg );
```

Вывод базы на экран и в файл выполняются аналогично. Естественно, что для вывода в файл требуется указать, в какой именно, поэтому параметром этой функции должен быть указатель на символьную строку, содержащую имя файла:

```
void print_dbase( Man *beg ); // На экран
int write_dbase( char *filename, Man *beg ); // В файл
```

Начальное формирование списка из файла выполняется в функции `read_dbase`:

```
Man* read_dbase( char *filename );
```

Ввод информации о сотруднике и вывод меню также являются логически законченными действиями, поэтому они тоже оформлены в виде функций `read_man` и `menu`. Приведем текст программы (листинг 9.2).

Листинг 9.2. Линейный список

```

#include <stdio>
#include <stdlib>
#include <string>
using namespace std;
const int l_name = 31;
struct Man {
    char name[l_name];
    int  birth_day;
    float pay;
    Man *next;
};
Man* add( Man *beg, const Man &man );
int  edit( Man *beg );
Man* find( Man *beg, char *name, Man *& prev );
void find( Man *beg, int birth_day );
void find( Man *beg, float pay );
void find_man( Man *beg );
void get_name( char *name );
int  menu();
void print_dbase( Man *beg );
Man* read_dbase( char *filename );
Man  read_man();
Man* remove( Man *beg );
int  write_dbase( char *filename, Man *beg );
// ----- Главная функция -----
int main() {
    Man *beg = read_dbase( "dbase.txt" );
    if ( !beg ) return 1;
    while ( true ) {
        switch ( menu() ) {
            case 1: add( beg, read_man() );           break;
            case 2: beg = remove( beg );             break;
            case 3: find_man( beg );                 break;
            case 4: edit( beg );                     break;
            case 5: print_dbase( beg );               break;
            case 6: write_dbase( "dbase.txt", beg );  break;
            case 7:                                  return 0;
            default: puts( " Надо вводить число от 1 до 7" ); break;
        }
    }
}
Man* add( Man *beg, const Man &man ) { // ----- Добавление сотрудника
    Man* pv = new Man;                  // Формирование нового элемента
    *pv = man;
    pv->next = 0;
    if ( beg ) {                        // Список не пуст
        Man* temp = beg;
        while ( temp->next ) temp = temp->next;    // Поиск конца списка
        temp->next = pv;                          // Привязывание нового элемента
    }
}

```

```

    else beg = pv;                                     // Список пуст
    return beg;
}
int edit( Man *beg ) { // ----- Корректировка сведений
    char name[l_name], buf[80];
    get_name( name );                                 // Кого ищем?
    Man* prev;
    Man* pv = find( beg, name, prev );
    if ( !pv ) return 1;                               // Не нашли
    do { puts( "Введите новый оклад " ); gets( buf );
    } while ( !( pv->pay = (float) atof(buf) ) );
    return 0;
}
// ----- Поиск сотрудника по фамилии
Man* find( Man* pv, char* name, Man *& prev ) {
    prev = 0;
    while( pv ) {
        if ( strstr( pv->name, name ) )
            if ( pv->name[strlen( name )] == ' ' ) {
                printf( "%30s%5i%10.2f\n", pv->name, pv->birth_day, pv->pay );
                return pv;
            }
        prev = pv;
        pv = pv->next;
    }
    puts( "Такого сотрудника нет\n" );
    return 0;
}
// ----- Поиск и вывод сотрудников по году рождения
void find( Man* pv, int birth_day ) {
    while ( pv ) {
        if ( pv->birth_day < birth_day )
            printf( "%30s%5i%10.2f\n", pv->name, pv->birth_day, pv->pay );
        pv = pv->next;
    }
}
// ----- Поиск и вывод сотрудников по окладу
void find( Man* pv, float pay ) {
    while ( pv ) {
        if ( pv->pay >= pay )
            printf( "%30s%5i%10.2f\n", pv->name, pv->birth_day, pv->pay );
        pv = pv->next;
    }
}
void find_man( Man* beg ) { // ----- Поиск
    char buf[l_name];
    int birth_day, option;
    float pay;
    Man *prev = 0, *pv = 0;
    do { puts( "1 - поиск по фамилии, 2 - по году рождения,\n\

```

продолжение ➤

Листинг 9.2 (продолжение)

```

        3 - по окладу, 4 - отмена\n ");
    gets( buf );
} while ( !( option = atoi( buf ) ) );
switch ( option ) {
    case 1: get_name( buf );
            pv = find( beg, buf, prev ); break;
    case 2: do { puts( "Введите год рождения\n" ); gets( buf ); }
            while ( !( birth_day = atoi( buf ) ) );
            find( beg, birth_day ); break;
    case 3: do { puts( "Введите оклад\n" ); gets( buf ); }
            while ( !( pay = (float) atof( buf ) ) );
            find( beg, pay ); break;
    case 4: return;
    default: puts( "неверный режим\n" );
}
}

void get_name( char* name ) { // ----- Запрос фамилии
    puts( "Введите фамилию И.О. " ); gets( name );
}

int menu() { // ----- Вывод меню
    char buf[10];
    int option;
    do { puts( "===== " );
        puts( "1 - добавление сотрудника\t 4 - корректировка сведений" );
        puts( "2 - удаление сотрудника\t\t 5 - вывод базы на экран" );
        puts( "3 - поиск сотрудника\t\t 6 - вывод базы в файл\t\t\t 7 - выход" );
        gets( buf );
        option = atoi( buf );
    } while ( !option );
    return option;
}

void print_dbase( Man* beg ) { // ----- Вывод базы на экран
    Man* pv = beg;
    while ( pv ) {
        printf( "%s%i%10.2f\n", pv->name, pv->birth_day, pv->pay );
        pv = pv->next;
    }
}

Man* read_dbase( char* filename ) { // ----- Чтение базы из файла
    FILE* fin;
    Man man, *beg = 0;
    if ( ( fin = fopen( filename, "r" ) ) == 0 ) {
        printf( "Нет файла %s\n", filename ); return 1;
    }
    while ( !feof( fin ) ) {
        fgets( man.name, 1_name, fin );
        fscanf( fin, "%i%f\n", &man.birth_day, &man.pay );
        beg = add( beg, man );
    }
}

```



```

    fclose( fin );
    return beg;
}
Man read_man() { // ----- Ввод информации о новом сотруднике
    Man man; char buf[80];
    get_name( man.name );
    for ( int i = strlen( man.name ); i < l_name; i++ ) man.name[i] = ' ';
    man.name[l_name - 1] = '\0';
    do { puts( "Введите год рождения " ); gets( buf ); }
    while ( !( man.birth_day = atoi( buf ) ) );
    do { puts( "Введите оклад " ); gets( buf ); }
    while ( !( man.pay = (float) atof( buf ) ) );
    return man;
}
Man* remove( Man* beg ) { // ----- Удаление сотрудника
    char name [l_name];
    get_name( name ); // Кого удаляем?
    Man* prev;
    Man* pv = find( beg, name, prev );
    if ( pv ) { // Если нашли
        if ( pv == beg ) beg = beg->next; // Удаление из начала списка
        else // Удаление из середины или конца списка
            prev->next = pv->next;
        delete pv; // Освобождение памяти из-под элемента
    }
    return beg;
}
// ----- Вывод базы в файл
int write_dbase( char *filename, Man *pv ) {
    FILE *fout;
    if ( ( fout = fopen( filename, "w" ) ) == NULL ) {
        puts( "Ошибка открытия файла" ); return 1; }
    while ( pv ) {
        fprintf( fout, "%s%5i%10.2f\n", pv->name, pv->birth_day, pv->pay );
        pv = pv->next;
    }
    fclose( fout );
    return 0;
}

```

Обратите внимание, что и прототипы, и сами функции расположены в алфавитном порядке и снабжены хорошо читаемыми комментариями. Это упрощает отладку программы. В программе предусмотрена «защита от дурака»: если пользователь введет неверный режим или нечисловые символы там, где этого делать нельзя, программа корректно обработает эту ситуацию (впрочем, программа в целом остается крайне незащищенной от переполнения буфера ввода, но с этим придется мириться до второй части практикума). При отладке вывода в файл рекомендуем вам задать имя, отличное от исходного файла, с тем, чтобы случайно его не испортить. Напомним, что специфика работы с кириллицей рассматривалась в задачах 1.1 и 5.1.

На этом примере вы можете потренироваться в технологии создания программы «сверху вниз»: сначала отладить главную функцию, а затем постепенно добавлять к ней остальные. На месте еще не добавленных функций обычно ставятся так называемые *заглушки* — функции, единственным действием которых является вывод сообщения о том, что эта функция была вызвана.

Забегая вперед, предложим более грамотное решение, касающееся использованной в этой задаче структуры данных. В программах, рассмотренных ранее, под именем `Man` мы описывали структуру, содержащую сведения о сотруднике. В этой задаче мы в угоду своим интересам добавили в нее поле указателя, которое, вообще говоря, к сведениям о сотруднике отношения не имеет. Более правильным было бы оставить структуру `Man` неизменной и унаследовать от нее структуру `Node`, которая и являлась бы элементом списка:

```
struct Man {
    char name[1_name];
    int birth_day;
    float pay;
};
struct Node: Man {
    Man *next;
};
```

Указание имени `Man` после двоеточия при описании структуры `Node` означает, что все ее поля «переходят» в структуру `Node`. Здесь мы воспользовались тем, что структура является формой класса, а для классов определено наследование, которое мы рассмотрим во второй части практикума.

Задача 9.3. Бинарное дерево

Написать программу учета нарушений правил дорожного движения. Для каждого автомобиля необходимо хранить в базе список нарушений. Для каждого нарушения фиксируется дата, время, вид нарушения и размер штрафа. При оплате всех штрафов автомобиль удаляется из базы.

Опишем внешнюю спецификацию нашей программы. Она должна позволять вносить нарушение в базу, удалять его при поступлении сведений об оплате штрафа, получать справку о нарушениях, совершенных на заданной машине, сохранять информацию в файле и считывать ее из файла. Позаимствуем принцип организации меню из предыдущей программы.

Учитывая специфику потенциальных пользователей программы, сделаем меню максимально простым: 1) ввод сведений о нарушении; 2) ввод сведений об оплате штрафа (удаление соответствующей записи о нарушении); 3) справка о нарушениях, совершенных на автомобиле с заданным номером; 4) выход.

Теперь определим внутреннюю форму представления данных. Так как число нарушителей в нашей стране приближается к общему количеству автовладельцев, база может приобретать огромные размеры, и скорость выполнения операций поиска

и корректировки становится актуальной. Очень эффективна в этом отношении структура данных, называемая *бинарным деревом*. Рассмотрим его свойства.

Каждый элемент (*узел*) дерева характеризуется уникальным *ключом*. Кроме ключа, узел должен содержать две ссылки: на свое левое и правое поддерево. У всех узлов левого поддерева ключи меньше, чем ключ данного узла, а у всех узлов правого поддерева — больше. В таком дереве, называемом *деревом поиска*, можно найти любой элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Время поиска определяется высотой дерева, которая пропорциональна двоичному логарифму количества его элементов (сравните с линейным списком, в котором время поиска пропорционально количеству его элементов).

Каждый автомобиль имеет уникальный госрегистрационный номер, он и будет ключом. Информационная часть узла будет содержать ссылку на список нарушений. Опишем элемент списка нарушений (штраф по-английски — «fine») и узел дерева поиска:

```
const int l_time = 20, l_type = 40, l_number = 12;
struct Fine { // -----Штраф (элемент списка)
    char time[l_time]; // Время нарушения
    char type[l_type]; // Вид нарушения
    float price; // Размер штрафа
    Fine *next; // Указатель на следующий элемент
};
struct Node { // -----Узел дерева
    char number[l_number]; // Номер автомобиля
    Fine *beg; // Указатель на начало списка нарушений
    Node *left; // Указатель на левое поддерево
    Node *right; // Указатель на правое поддерево
};
```

Опишем общий *алгоритм работы программы*. При вводе нового нарушения запрашивается номер автомобиля и выполняется поиск в базе. Если автомобиль уже фигурирует в базе, в список его нарушений добавляется новое, а если нет, то создаются новый узел дерева и первый элемент списка нарушений.

При вводе информации об уплате штрафа выполняется поиск заданного автомобиля, а затем — поиск соответствующего нарушения. При этом запрашивается время совершения нарушения, поскольку все остальные параметры нескольких нарушений могут совпадать (например, автолюбителя семь раз за день оштрафовали за превышение скорости при развороте задним ходом на перекрестке под запрещающий сигнал светофора). Если после удаления записи о нарушении список оказывается пустым, соответствующий узел дерева удаляется. При запуске программы данные считываются из файла, а при окончании работы с программой файл обновляется.

Разобьем программу на функции и определим интерфейс каждой из них. Выполнение начинается с формирования двоичного дерева из файла базы нарушений. Назовем функцию чтения `read_dbase`. Она получает файл, из которого следует считывать информацию, а возвращает указатель на корень сформированного дерева:

```
Node* read_dbase( char* filename );
```

Наличие файла не обязательно — ведь надо же иметь возможность начать работать с пустой базой, не говоря уже о греющей душу автолюбителя вероятности гибели базы вследствие (не)преднамеренной порчи диска. При отсутствии файла выдается предупреждающее сообщение и возвращается нулевой указатель.

Для реализации первого пункта меню (ввод сведений о нарушении) необходимо выполнить два отдельных действия: ввести нужную информацию и занести ее в дерево. Вводимые сведения удобно представить в виде структуры:

```
struct Data {                                // ----- Исходные данные
    char  number[l_number]; // Номер автомобиля
    char  time[l_time];     // Время нарушения
    char  type[l_type];     // Вид нарушения
    float price;            // Размер штрафа
};
```

Функция `input` будет запрашивать ввод этих данных, проверять их корректность и возвращать их в вызывающую функцию. Задумаемся, в каких еще случаях, кроме формирования нового нарушения, может потребоваться ввод данных? Он необходим и для второго, и для третьего пунктов меню, но набор сведений различается: если для внесения данных в базу требуется вся информация (номер автомобиля, время и вид нарушения, размер штрафа), то для удаления оплаченного штрафа необходимы только номер автомобиля и время нарушения. Для получения справки вводится только номер машины. Чтобы ввод был сосредоточен в одной функции, опишем режим ее работы в виде перечисления и будем передавать его в качестве параметра:

```
enum Action { INSERT, DEL, INFO };
Data input( Action action );
```

Занесение введенных сведений в дерево также оформим в виде функций. Поскольку процедура создания корневого узла отличается от создания остальных узлов, таких функций будет две. Для создания корневого узла требуется передать в функцию структуру `Data`. Функция возвращает указатель на созданный корень дерева:

```
Node * first( Data data );
```

Занесение остальных узлов совместим с поиском, так как в обоих случаях требуется выполнять спуск по дереву. Назовем функцию `search_insert`. Чтобы использовать ее и при удалении узла из дерева, придется расширить ее интерфейс, добавив туда, кроме указателя на корень дерева, заносимых (искомых) данных и режима работы, указатель на родителя узла и признак, к левому или правому поддереву родителя принадлежит искомый узел. Функция возвращает указатель на найденный (вставленный) узел:

```
enum Dir { LEFT, RIGHT } ;
Node* search_insert( Node* root, const Data& data,
                    Action action, Dir& dir, Node*& parent );
```

Признак типа `Dir` передается в вызывающую функцию по ссылке, а указатель на родительский узел — по адресу, чтобы можно было передать наружу их значения.

Функция удаления записи об одном нарушении `remove_fine` получает в качестве параметра указатель на узел, из списка которого следует удалить запись, а также время нарушения, которое служит ключом списка. Время для единообразия передается в составе структуры `Data`. Функция удаляет запись из списка или выдает информацию о том, что запись не найдена, и возвращает признак, по которому можно определить, пуст ли список нарушений:

```
int remove_fine( Node* p, const Data& data );
```

Если список пуст, необходимо удалить соответствующий узел дерева. Для этого служит функция `remove_node`:

```
Node* remove_node( Node* root, Node* p, Node* parent, Dir dir );
```

Ради этой функции мы и вводили в состав параметров функции поиска `search_insert` указатель на родительский узел и признак левого или правого поддеревья, поскольку надо куда-то привязать поддеревья удаляемого узла. Удалить можно любой узел, в том числе и корневой, поэтому функция `remove_node` должна возвращать указатель на корень дерева на тот случай, если он изменится.

Для получения справки о нарушениях, совершенных на автомобиле с заданным номером, служит функция `print_node`:

```
void print_node( const Node& node );
```

По окончании работы с базой она выводится в файл, для этого предназначена функция `write_dbase`. Ей передается файл, в который будет выводиться информация, и указатель на корень дерева:

```
void write_dbase( ofstream f, const Node* root );
```

Почему мы в данном случае передаем в функцию не имя файла, а уже открытый выходной поток, мы объясним позже, при анализе соответствующего алгоритма.

Разбиение программы на функции и спецификация их интерфейсов — процесс итеративный, поскольку сразу обычно не удастся учесть все детали и тонкости. Однако мы от души рекомендуем вам уделять этому этапу как можно больше внимания¹, поскольку чем четче разделена программа на независимые части, тем меньше времени потребует ее отладка. Приведем текст программы (листинг 9.3), а затем дадим необходимые пояснения.

Листинг 9.3. Бинарное дерево

```
#include <fstream>
#include <stdlib>
#include <cstring>
#include <ctime>
#include <iomanip>
using namespace std;
const char* filename = "dbase";
enum Action { INSERT, DEL, INFO };
enum Dir    { LEFT, RIGHT };
```

продолжение ➤

¹ Хотя, положив руку на сердце, мы сами не всегда следуем своим советам.

Листинг 9.3 (продолжение)

```

const int l_time = 20, l_type = 40, l_number = 12;
struct Fine { // ----- Штраф (элемент списка)
    char time[l_time]; // Время нарушения
    char type[l_type]; // Вид нарушения
    float price; // Размер штрафа
    Fine* next; // Указатель на следующий элемент
};
struct Node { // ----- Узел дерева
    char number[l_number]; // Номер автомобиля
    Fine* beg; // Указатель на начало списка нарушений
    Node* left; // Указатель на левое поддерево
    Node* right; // Указатель на правое поддерево
};
struct Data { // ----- Исходные данные
    char number[l_number]; // Номер автомобиля
    char time[l_time]; // Время нарушения
    char type[l_type]; // Вид нарушения
    float price; // Размер штрафа
};
Node* descent ( Node* p );
Node* first ( Data data );
Data input ( Action action );
int menu();
void print_node ( const Node& node );
void print_dbase( Node* p );
Node* read_dbase ( char* filename );
int read_fine ( ifstream f, Data& data );
int remove_fine( Node* p, const Data& data );
void remove_fines( Node* p );
Node* remove_node( Node* root, Node* p, Node* parent, Dir dir );
Node* remove_tree( Node* p );
Node* search_insert( Node* root, const Data& data, Action action,
    Dir& dir, Node*& parent );
void write_dbase( ofstream f, const Node* root );
void write_node( ofstream f, const Node& node );
int main() { // ----- Главная функция -----
    Node* p, *parent;
    Node* root = read_dbase( filename );
    ofstream fout;
    Dir dir;
    while ( true ) {
        switch ( menu() ) {
            case 1: // Ввод сведений о нарушении
                if ( !root ) root = first( input( INSERT ) );
                else search_insert( root, input( INSERT ), INSERT, dir, parent );
                break;
            case 2: // Ввод сведений об оплате штрафа
                if ( !root ) { cout << "База пуста" << endl; break; }
                Data data = input( DEL );

```

```

        if ( !( p = search_insert( root, data, DEL, dir, parent ) ) )
            cout << "Сведения об а/м отсутствуют " << endl;
        else if ( remove_fine( p, data ) == 2 ) // Удалены все нарушения
            root = remove_node( root, p, parent, dir );
        break;
    case 3: // Справка
        if ( !root ) { cout << "База пуста" << endl; break; }
        if ( !(p = search_insert( root, input(INFO), INFO, dir, parent )) )
            cout << "Сведения отсутствуют " << endl;
        else print_node( *p );
        break;
    case 4: // Выход
        fout.open( filename );
        if ( !fout.is_open() ) {
            cout << "Ошибка открытия файла " << filename; return 1; }
        write_dbase( fout, root );
        return 0;
    case 5: print_dbase( root ); break; // Отладка
    default: cout << " Надо вводить число от 1 до 4" << endl; break;
}}}
Node* descent( Node* p ) { // ----- Спуск по дереву
    Node* prev, *y = p->right;
    if ( !y->left ) y->left = p->left; // 1
    else {
        do { prev = y; y = y->left; } // 2
        while( y->left );
        y->left = p->left; // 3
        prev->left = y->right; // 4
        y->right = p->right; // 5
    }
    return y;
}
Node* first( Data data ) { // ----- Формирование корневого элемента дерева
    // Создание записи о нарушении:
    Fine* beg = new Fine;
    strncpy( beg->time, data.time, l_time );
    strncpy( beg->type, data.type, l_type );
    beg->price = data.price;
    beg->next = 0;
    // Создание первого узла дерева:
    Node* root = new Node;
    strncpy( root->number, data.number, l_number );
    root->beg = beg; root->left = root->right = 0;
    return root;
}
Data input( Action action ) { // ----- Ввод нарушения
    Data data;
    char buf[10], temp1[3], temp2[3];
    int day, month, hour, min;
    cout << "Введите номер а/м" << endl;

```

продолжение ➤

Листинг 9.3 (продолжение)

```

cin.getline( data.number, l_number );
if ( action == INFO ) return data;
do { cout << "Введите дату нарушения в формате ДД.ММ.ГГ:" << endl;
    cin >> buf;
    strncpy( temp1, buf, 2 ); strncpy( temp2, &buf[3], 2 );
    day = atoi( temp1 ); month = atoi( temp2 );
} while ( !( day > 0 && day < 32 && month > 0 && month < 13 ) );
strcpy( data.time, buf ); strcat( data.time, " " );
do { cout << "Введите время нарушения в формате ЧЧ:ММ :" << endl;
    cin >> buf;
    strncpy( temp1, buf, 2 ); strncpy( temp2, &buf[3], 2 );
    hour = atoi( temp1 ); min = atoi( temp2 );
} while ( !( hour >= 0 && hour < 24 && min >= 0 && min < 60 ) );
strcat( data.time, buf );
cin.get();
if ( action == DEL ) return data;
cout << "Введите тип нарушения type" << endl;
cin.getline( data.type, l_type );
do { cout << "Введите размер штрафа:" << endl;
    cin >> buf;
} while ( !( data.price = ( float )atof( buf ) ) );
cin.get();
return data;
}

int menu() { // ----- Вывод меню
    char buf[10];
    int option;
    do { cout << "===== " << endl;
        cout << "1 - Ввод сведений о нарушении" << endl;
        cout << "2 - Ввод сведений об оплате штрафа" << endl;
        cout << "3 - Справка" << endl;
        cout << "4 - Выход" << endl;
        cout << "===== " << endl;
        cin >> buf; option = atoi( buf );
    } while ( !option );
    cin.get();
    return option;
}

void print_node( const Node& node ) { // ----- Вывод на экран сведений об а/м
    cout << "Номер а/м: " << node.number << endl;
    Fine* pf = node.beg;
    float summa = 0;
    while ( pf ) {
        cout << "Вид нарушения: " << pf->type << endl;
        cout << "Дата и время: " << pf->time;
        cout << " Размер штрафа: " << pf->price << endl;
        summa += pf->price;
        pf = pf->next;
    }
}

```



```

    cout << "Общая сумма штрафов: " << summa << endl;
}
void print_dbase( Node *p ) {    // ----- Вывод на экран всего дерева
    if ( p ) {
        print_node( *p );
        print_dbase ( p->left );
        print_dbase ( p->right );
    }
}
Node * read_dbase ( char* filename ) {    // ----- Чтение базы из файла
    Node *parent;
    Dir dir;
    Data data;
    ifstream f( filename, ios::in | ios::nocreate );
    if ( !f ) { cout << "Нет файла " << filename << endl; return 1; }
    f.getline( data.number, l_number );    // Номер а/м
    if ( f.eof( ) ) { cout << "Пустой файл ( 0 байт )" << endl; return 1; }
    read_fine( f, data );    // Первое нарушение
    Node* root = first( data );    // Формирование корня дерева
    while ( !read_fine( f, data ) )    // Последующие нарушения
        search_insert( root, data, INSERT, dir, parent );
        // Формирование дерева:
    while ( f.getline( data.number, l_number ) ) {    // Номер а/м
        read_fine( f, data );    // Первое нарушение
        search_insert( root, data, INSERT, dir, parent );
        while ( !read_fine( f, data ) )    // Последующие нарушения
            search_insert( root, data, INSERT, dir, parent );
    }
    return root;
}
// ----- Чтение из файла информации о нарушении
int read_fine( ifstream f, Data& data ) {
    f.getline( data.time, l_time );
    if( data.time[0] == '=' ) return 1;    // Конец списка нарушений
    f.getline( data.type, l_type );
    f >> data.price;
    f.get();
    return 0;    // Нарушение считано успешно
}
// ----- Удаление нарушения из списка
int remove_fine( Node* p, const Data& data ) {
    Fine* prev, *pf = p->beg;
    bool found = false;
    while ( pf && !found ) {    // Цикл по списку нарушений
        if( !strcmp( pf->time, data.time ) ) found = true;    // Нарушение найдено
        else { prev = pf; pf = pf->next; }    // Переход к след. элементу списка
    }
    if ( !found ) { cout << "Сведения о нарушении отсутствуют." << endl;
        return 1; }
    if ( pf == p->beg ) p->beg = pf->next; // Удаление из начала списка
}

```

продолжение ➤

Листинг 9.3 (продолжение)

```

    else          prev->next = pf->next; // Удаление из середины или конца списка
    delete pf;    // Освобождение памяти из-под элемента
    if ( !p->beg ) return 2;           // Список пуст
    return 0;
}
// ----- Удаление узла дерева
Node* remove_node( Node* root, Node* p, Node* parent, Dir dir ) {
    Node *y;          // Узел, заменяющий удаляемый
    if ( !p->left )    y = p->right;    // 11
    else if ( !p->right ) y = p->left;   // 12
    else              y = descent( p ); // 13
    if ( p == root ) root = y;         // 14
    else {            // 15
        if ( dir == LEFT ) parent->left = y;
        else               parent->right = y;
    }
    delete p;          // 16
    return root;
}
// ----- Поиск с включением
Node* search_insert( Node* root, const Data& data, Action action,
                    Dir& dir, Node*& parent ) {
    Node* p = root;    // Указатель на текущий элемент
    bool found = false; // Признак успешного поиска
    int cmp;           // Результат сравнения ключей
    parent = 0;
    while ( p && !found ) { // Цикл поиска по дереву
        cmp = strcmp( data.number, p->number ); // Сравнение ключей
        if ( !cmp ) found = true;             // Нашли!
        else { parent = p;
            if ( cmp < 0 ) { p = p->left; dir = LEFT; } // Спуск влево
            else           { p = p->right; dir = RIGHT; } // Спуск вправо
        }
    }
    if ( action != INSERT ) return p;
    // Создание записи о нарушении:
    Fine* pf = new Fine;
    strncpy( pf->time, data.time, l_time );
    strncpy( pf->type, data.type, l_type );
    pf->price = data.price;
    pf->next = 0;
    if ( !found ) { // Создание нового узла:
        p = new Node;
        strncpy( p->number, data.number, l_number );
        p->beg = pf; p->left = p->right = 0;
        if ( dir == LEFT ) parent->left = p; // Присоед. к левому поддереву предка
        else parent->right = p; // Присоединение к правому поддереву предка
    }
}

```

```

    else {
        Fine* temp = p->beg;           // Узел существует, добавление нарушения в список
        while ( temp->next ) temp = temp->next;           // Поиск конца списка
        temp->next = pf;
    }
    return p;
}
void write_dbase( ofstream f, const Node *p ) { // ----- Вывод базы в файл
    if ( p ) {
        write_node( f, *p );
        write_dbase ( f, p->left );
        write_dbase ( f, p->right );
    }
}
void write_node( ofstream f, const Node& node ) { // - Вывод в файл сведений об а/м
    f << node.number << endl;
    Fine* pf = node.beg;
    while( pf ) {
        f << pf->time << endl << pf->type << endl << pf->price << endl;
        pf = pf->next;
    }
    f << "=" << endl;           // Признак окончания сведений об а/м
}

```

Рассмотрим функцию записи дерева в файл `write_dbase`. Поскольку дерево — структура рекурсивная, удобно сделать эту функцию рекурсивной. Для каждого поддерева в файл выводятся сведения о корне, затем о левом и правом поддереве. Именно из-за рекурсии мы открыли файл вне этой функции и передали в нее готовый к записи поток. Для вывода информации об одном узле используется вспомогательная функция `write_node`. Для простоты данные выводятся в текстовом виде в отдельные строки (в реальной системе пришлось бы их хитроумно зашифровать). Признаком окончания списка нарушений для одного автомобиля служит символ «=». Он используется при чтении базы из файла.

Функция чтения из файла `read_dbase` весьма проста. В ней используется вспомогательная функция считывания записи об одном нарушении `read_fine`. Сначала вводится информация о номере первого автомобиля и его первом нарушении и заносится в корень дерева с помощью функции `first`, затем организуется цикл по всем остальным нарушениям данного автомобиля, если они есть. Нарушения добавляются в список с помощью функции `search_insert`.

Далее аналогичные действия производятся для всех остальных автомобилей. Конечно, можно было бы не искать требуемый узел и конец списка нарушений заново для каждого нарушения из списка, а хранить соответствующие указатели, но это потребовало бы дополнительного кода и не дало бы существенного выигрыша во времени, так как база считывается всего один раз до начала работы.

Наверно, вы обратили внимание, что в меню четыре пункта, а в операторе `switch` главной функции пять вариантов выбора. Последний вариант не входит в интерфейс

пользователя, он может понадобиться программисту при отладке или сопровождении программы. Для отладки была также написана функция вывода базы `print_dbase`.

Функция поиска с включением `search_insert` исчерпывающе, на наш взгляд, документирована в программе. Кроме того, описание аналогичной функции приведено в Учебнике на с. 125. Обратим наше внимание на более интересную функцию `remove_node`, которая удаляет узел из дерева.

Процесс удаления можно разбить на этапы:

- ☐ найти узел, который будет поставлен на место удаляемого;
- ☐ реорганизовать дерево так, чтобы не нарушились его свойства;
- ☐ присоединить новый узел к узлу-предку удаляемого узла;
- ☐ освободить память из-под удаляемого узла.

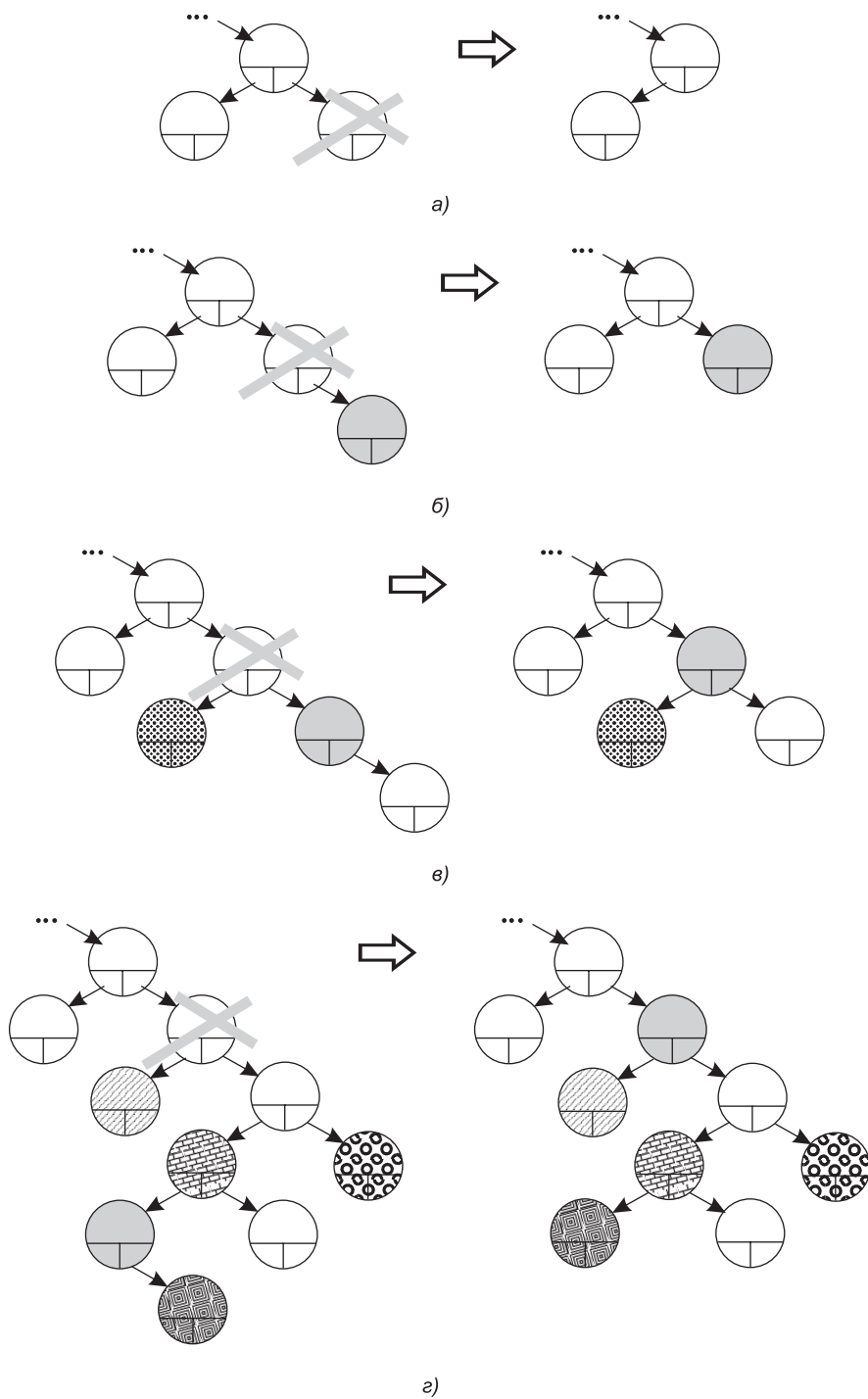
Удаление узла происходит по-разному в зависимости от его расположения в дереве. Если узел является листом, то есть не имеет потомков, достаточно обнулить соответствующий указатель узла-предка (рис. 9.1, *а*). Если узел имеет только одного потомка, то этот потомок ставится на место удаляемого узла, а в остальном дерево не изменяется (рис. 9.1, *б*). Хуже всего, когда у узла есть оба потомка, но и здесь есть простой особый случай: если у его правого потомка нет левого потомка, удаляемый узел заменяется своим правым потомком, а левый потомок удаляемого узла подключается вместо отсутствующего левого потомка. Согласимся, что это звучит не очень понятно, поэтому рассмотрим этот случай на рис. 9.1, *в*.

В общем же случае на место удаляемого узла помещается самый левый лист его правого поддерева (или наоборот — самый правый лист его левого поддерева), это не нарушает свойств дерева поиска. Этот случай иллюстрируется рис. 9.1, *г*.

Корень дерева удаляется аналогичным образом за исключением того, что заменяющий его узел не требуется подсоединять к узлу-предку. Вместо этого обновляется указатель на корень дерева.

Рассмотрим реализацию этого алгоритма в программе `remove_node`. В функцию передается указатель на удаляемый узел `p`. Сначала находим указатель на узел `u`, который должен заменить удаляемый. Если у узла `p` нет левого поддерева, на его место будет поставлена вершина (возможно, пустая) его правого поддерева (*оператор 11*). Иначе, если у узла `p` нет правого поддерева, на его место будет поставлена вершина его левого поддерева (*оператор 12*).

В противном случае оба поддерева узла существуют, и для определения заменяющего узла вызывается вспомогательная функция `descent`, выполняющая спуск по дереву. В этой функции прежде всего проверяется особый случай, описанный выше (*оператор 1*). Если же условие отсутствия левого потомка у правого потомка удаляемого узла не выполняется, организуется цикл (*оператор 2*), на каждой итерации которого указатель на текущий узел запоминается в переменной `prev`, а указатель `u` смещается вниз и влево по дереву до того момента, пока не станет ссылаться на узел, не имеющий левого потомка — он-то нам и нужен!

**Рис. 9.1.** Удаление из бинарного дерева

В *операторе 3* к этой пустующей ссылке присоединяется левое поддерево удаляемого узла. Перед тем, как присоединять к этому узлу правое поддерево удаляемого узла (*оператор 5*), требуется «пристроить» его собственное правое поддерево. Мы присоединяем его к левому поддереву предка заменяющего узла *у* (*оператор 4*), поскольку этот узел перейдет на новое место.

Функция `descent` возвращает указатель на узел, заменяющий удаляемый. Если мы удаляем корень, надо обновить указатель на него (*оператор 14*), иначе — присоединить этот узел к соответствующему поддереву предка удаляемого узла (*оператор 15*). После удаления узла освобождаем память (*оператор 16*).

Как вы могли убедиться, работа с динамическими структурами данных требует внимания и тщательности. Однако можно заметить, что в различных задачах изменяется только информационная часть компонентов, а принципы работы остаются неизменными, поэтому фактически функции для работы с каждой из этих структур пишутся и отлаживаются один раз. Разработчики библиотек тоже заметили этот факт и включили в стандартную библиотеку C++ так называемые *шаблоны*, реализующие основные динамические структуры данных. Во второй части практикума мы рассмотрим их применение. Для понимания шаблонов вам придется изучить многие средства и возможности C++, но предпринятые усилия быстро себя окупят.

Итоги

1. Динамическими структурами данных называются блоки в динамической памяти, связанные друг с другом с помощью указателей. Структуры различаются способами связи отдельных элементов и допустимыми операциями над ними. Наиболее распространенными структурами являются линейный список (односвязный или двусвязный), стек, очередь и бинарное дерево.
2. Элемент любой динамической структуры данных состоит из информационных полей и полей указателей.
3. Для стека определены операции помещения элемента в вершину и выборки элемента из вершины. Для очереди определены операции помещения элемента в конец очереди и выборка элемента из ее начала. Допускается вставлять и удалять элементы в произвольное место линейного списка.
4. Бинарное дерево состоит из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Каждый узел дерева характеризуется уникальным ключом. Допускается вставлять и удалять элементы в произвольное место дерева.
5. Если для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, то такое дерево называется деревом поиска. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле, что гораздо эффективнее поиска в списке.
6. Динамические структуры в некоторых случаях более эффективно реализовывать с помощью массивов (см. Учебник, с. 126).

Задания

Задания этого семинара соответствуют приведенным в Учебнике на с. 165.

Вариант 1

Составить программу, которая содержит динамическую информацию о наличии автобусов в автобусном парке. Сведения о каждом автобусе включают номер автобуса, фамилию и инициалы водителя и номер маршрута. Программа должна обеспечивать:

- ☐ начальное формирование данных обо всех автобусах в парке в виде списка;
- ☐ при выезде каждого автобуса из парка вводится номер автобуса, и программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- ☐ при въезде каждого автобуса в парк вводится номер автобуса, и программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- ☐ по запросу выдаются сведения об автобусах, находящихся в парке, или об автобусах, находящихся на маршруте.

Вариант 2

Составить программу, которая содержит текущую информацию о книгах в библиотеке. Сведения о книгах включают:

- ☐ номер УДК; фамилию и инициалы автора;
- ☐ название; год издания;
- ☐ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ☐ начальное формирование данных о книгах в виде двоичного дерева;
- ☐ добавление данных о книгах, вновь поступающих в библиотеку;
- ☐ удаление данных о списываемых книгах;
- ☐ по запросу выдаются сведения о наличии книг в библиотеке, упорядоченные по годам издания.

Вариант 3

Составить программу, которая содержит текущую информацию о заявках на авиабилеты. Каждая заявка включает:

- ☐ пункт назначения; номер рейса;
- ☐ фамилию и инициалы пассажира; желаемую дату вылета.

Программа должна обеспечивать:

- ☐ хранение всех заявок в виде списка;
- ☐ добавление заявок в список;
- ☐ удаление заявок;

- ☐ вывод заявок по заданному номеру рейса и дате вылета;
- ☐ вывод всех заявок.

Вариант 4

Составить программу, которая содержит текущую информацию о заявках на авиабилеты. Каждая заявка включает:

- ☐ пункт назначения; номер рейса;
- ☐ фамилию и инициалы пассажира; желаемую дату вылета.

Программа должна обеспечивать:

- ☐ хранение всех заявок в виде двоичного дерева;
- ☐ добавление и удаление заявок;
- ☐ по заданному номеру рейса и дате вылета вывод заявок с их последующим удалением;
- ☐ вывод всех заявок.

Вариант 5

Составить программу, которая содержит текущую информацию о книгах в библиотеке. Сведения о книгах включают:

- ☐ номер УДК; фамилию и инициалы автора;
- ☐ название; год издания;
- ☐ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ☐ начальное формирование данных обо всех книгах в библиотеке в виде списка;
- ☐ при выдаче каждой книги на руки вводится номер УДК, и программа уменьшает значение количества книг на единицу или выдает сообщение о том, что требуемой книги в библиотеке нет или требуемая книга находится на руках;
- ☐ при возвращении каждой книги вводится номер УДК, и программа увеличивает значение количества книг на единицу;
- ☐ по запросу выдаются сведения о наличии книг в библиотеке.

Вариант 6

Составить программу, которая содержит динамическую информацию о наличии автобусов в автобусном парке. Сведения о каждом автобусе включают:

- ☐ номер автобуса;
- ☐ фамилию и инициалы водителя;
- ☐ номер маршрута;
- ☐ признак того, где находится автобус — на маршруте или в парке.

Программа должна обеспечивать:

- ☐ начальное формирование данных обо всех автобусах в виде списка;
- ☐ при выезде каждого автобуса из парка вводится номер автобуса, и программа устанавливает значение признака «автобус на маршруте»;

- ☐ при въезде каждого автобуса в парк вводится номер автобуса, и программа устанавливает значение признака «автобус в парке»;
- ☐ по запросу выдаются сведения об автобусах, находящихся в парке, или об автобусах, находящихся на маршруте.

Вариант 7

Создать программу, отыскивающую проход по лабиринту. Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице.

Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов. Для хранения пути использовать стек.

Вариант 8

Гаражная стоянка имеет одну стояночную полосу, причем единственный въезд и он же выезд находится в одном конце полосы. Если владелец автомобиля приходит забрать автомобиль, который не является ближайшим к выходу, то все автомобили, загораживающие проезд, удаляются, машина данного владельца выводится со стоянки, а другие машины возвращаются на стоянку в исходном порядке.

Написать программу, которая моделирует процесс прибытия и отъезда машин. Прибытие или отъезд автомобиля задается командной строкой, которая содержит признак прибытия или отъезда и номер машины. Программа должна выводить сообщение при прибытии или выезде любой машины. При выезде автомобиля со стоянки сообщение должно содержать число случаев, когда машина удалялась со стоянки для обеспечения выезда других автомобилей.

Вариант 9

Написать программу, моделирующую заполнение гибкого магнитного диска. Общий объем памяти на диске 360 Кбайт. Файлы имеют произвольную длину от 18 байт до 32 Кбайт. В процессе работы файлы либо записываются на диск, либо удаляются с него.

В начале работы файлы записываются подряд друг за другом. После удаления файла на диске образуется свободный участок памяти, и вновь записываемый файл либо размещается на свободном участке, либо, если файл не помещается в свободный участок, размещается после последнего записанного файла.

В случае, когда файл превосходит длину самого большого свободного участка, выдается аварийное сообщение. Требование на запись или удаление файла задается в командной строке, которая содержит имя файла, его длину в байтах, признак записи или удаления. Программа должна выдавать по запросу сведения о занятых и свободных участках памяти на диске.

Указание: создать списки занятых и свободных участков памяти на диске.

Вариант 10

В файловой системе каталог файлов организован в виде линейного списка. Для каждого файла в каталоге содержится имя файла, дата создания и количество обращений к файлу. Написать программу, которая обеспечивает:

- ☐ начальное формирование каталога файлов;
- ☐ вывод каталога файлов;
- ☐ удаление файлов, дата создания которых меньше заданной;
- ☐ выборку файла с наибольшим количеством обращений.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 11

Предметный указатель организован в виде линейного списка. Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, — от одного до десяти.

Написать программу, которая обеспечивает:

- ☐ начальное формирование предметного указателя;
- ☐ вывод предметного указателя;
- ☐ вывод номеров страниц для заданного слова.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 12

Текст помощи для некоторой программы организован в виде линейного списка. Каждый компонент текста помощи содержит термин (слово) и текст, содержащий пояснения к этому термину. Количество строк текста, относящихся к одному термину, составляет от одной до пяти. Написать программу, которая обеспечивает:

- ☐ начальное формирование текста помощи;
- ☐ вывод текста помощи;
- ☐ вывод поясняющего текста для заданного термина.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 13

Картотека в бюро обмена квартир организована в виде линейного списка. Сведения о каждой квартире включают количество комнат, этаж, площадь и адрес. Написать программу, которая обеспечивает:

- ☐ начальное формирование картотеки;
- ☐ ввод заявки на обмен;
- ☐ поиск в картотеке подходящего варианта: при равенстве количества комнат и этажа и различии площадей в пределах 10% соответствующая карточка выводится

и удаляется из списка, в противном случае поступившая заявка включается в список;

- ☐ вывод всего списка.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 14

Англо-русский словарь построен в виде двоичного дерева. Каждый компонент содержит английское слово, соответствующее ему русское слово и счетчик количества обращений к данной компоненте.

Первоначально дерево формируется в порядке английского алфавита. В процессе эксплуатации словаря при каждом обращении к компоненту к счетчику обращений добавляется единица. Написать программу, которая

- ☐ обеспечивает начальный ввод словаря с конкретными значениями счетчиков обращений;
- ☐ формирует новое представление словаря в виде двоичного дерева по следующему алгоритму: а) в старом словаре ищется компонент с наибольшим значением счетчика обращений; б) найденный компонент заносится в новый словарь и удаляется из старого; в) переход к п. а) до исчерпания исходного словаря.
- ☐ производит вывод исходного и нового словарей.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 15

Анкета для опроса населения содержит две группы вопросов. Первая группа содержит сведения о респонденте: возраст; пол; образование (начальное, среднее, высшее). Вторая группа содержит собственно вопрос анкеты, ответом на который может являться либо ДА, либо НЕТ. Написать программу, которая

- ☐ обеспечивает начальный ввод анкет и формирует из них линейный список;
- ☐ на основе анализа анкет выдает ответы на следующие вопросы: а) сколько мужчин старше 40 лет, имеющих высшее образование, ответили ДА на вопрос анкеты; б) сколько женщин моложе 30 лет, имеющих среднее образование, ответили НЕТ на вопрос анкеты; в) сколько мужчин моложе 25 лет, имеющих начальное образование, ответили ДА на вопрос анкеты;
- ☐ производит вывод всех анкет и ответов на вопросы.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 16

Написать программу, которая содержит текущую информацию о книгах в библиотеке. Сведения о книгах включают:

- ☐ номер УДК; фамилию и инициалы автора;
- ☐ название; год издания;
- ☐ количество экземпляров данной книги в библиотеке.

Программа должна обеспечивать:

- ☐ начальное формирование данных о всех книгах в библиотеке в виде списка;
- ☐ добавление данных о книгах, вновь поступающих в библиотеку;
- ☐ удаление данных о списываемых книгах;
- ☐ по запросу выдаются сведения о наличии книг в библиотеке, упорядоченные по годам издания.

Вариант 17

На междугородной телефонной станции картотека абонентов, содержащая сведения о телефонах и их владельцах, организована в виде линейного списка. Написать программу, которая

- ☐ обеспечивает начальное формирование картотеки в виде линейного списка;
- ☐ производит вывод всей картотеки;
- ☐ вводит номер телефона и время разговора;
- ☐ выводит извещение на оплату телефонного разговора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 18

На междугородной телефонной станции картотека абонентов, содержащая сведения о телефонах и их владельцах, организована в виде двоичного дерева. Написать программу, которая

- ☐ обеспечивает начальное формирование картотеки в виде двоичного дерева; производит вывод всей картотеки;
- ☐ вводит номер телефона и время разговора;
- ☐ выводит извещение на оплату телефонного разговора.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 19

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов. Для каждого поезда указывается номер поезда; станция назначения; время отправления. Данные в информационной системе организованы в виде линейного списка. Написать программу, которая

- ☐ обеспечивает первоначальный ввод данных в информационную систему и формирование линейного списка;
- ☐ производит вывод всего списка;
- ☐ вводит номер поезда и выводит все данные об этом поезде;
- ☐ вводит название станции назначения и выводит данные обо всех поездах, следующих до этой станции.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Вариант 20

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов. Для каждого поезда указывается номер поезда; станция назначения; время отправления. Данные в информационной системе организованы в виде двоичного дерева. Написать программу, которая

- ☐ обеспечивает первоначальный ввод данных в информационную систему и формирование двоичного дерева;
- ☐ производит вывод всего дерева;
- ☐ вводит номер поезда и выводит все данные об этом поезде;
- ☐ вводит название станции назначения и выводит данные о всех поездах, следующих до этой станции.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок ввода.

Часть II. Объектно-ориентированное программирование

В этой части практикума дается введение в объектно-ориентированную технологию создания программ. На примерах описываются принципы и особенности создания и наследования классов, использование стандартных шаблонов и создание собственных, основы работы с потоками, «скользящие места» и технология отладки.

Семинар 10. Классы

Теоретический материал: с. 178–199.

Появление объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) — это технология, возникшая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленных программных продуктов. Следствия — срыв сроков проектов, перерасход бюджета, урезанная функциональность и множество ошибок.

Существенная черта промышленной программы — ее *сложность*: один разработчик не в состоянии охватить все аспекты системы, поэтому в ее создании участвует целый коллектив. Следовательно, к первичной сложности самой задачи, вытекающей из предметной области, добавляются проблемы управления процессом разработки.

Так как сложные системы разрабатываются в расчете на длительную эксплуатацию, то появляются еще две проблемы: *сопровождение* системы (устранение обнаруженных ошибок) и ее *модификация*, поскольку у заказчика постоянно появляются новые требования и пожелания. Часто затраты на сопровождение и модификацию сопоставимы с затратами на собственно разработку системы.

Способ управления сложными системами был известен еще в древности — *divide et impera* (разделяй и властвуй). То есть выход — в *декомпозиции* системы на все меньшие и меньшие подсистемы, каждую из которых можно разрабатывать независимо. Но если в рамках структурного подхода декомпозиция понимается как разбиение алгоритма, когда каждый из модулей системы выполняет один из этапов общего процесса, то ООП предлагает совершенно другой подход.

Суть его в том, что в качестве *критерия декомпозиции* принимается принадлежность элементов системы к различным *абстракциям проблемной области*. Откуда же они берутся? Исключительно из головы программиста, который, анализируя предметную область, вычленяет из нее отдельные объекты. Для каждого из них определяются свойства, существенные для решения задачи. Так из реальных объектов предметной области получают абстрактные программные объекты.

Почему *объектно-ориентированная декомпозиция* оказалась более эффективной, чем *функциональная* (синонимы — *структурная, процедурная, алгоритмически-ориентированная*) декомпозиция? Тому есть много причин. Чтобы в них разобраться, рассмотрим критерии качества проекта, связанные с его декомпозицией.

Критерии качества декомпозиции проекта

Со *сложностью приложения* трудно что-либо сделать — она определяется целью создания программы. А вот *сложность реализации* можно попытаться контролировать. Первый вопрос, возникающий при декомпозиции: на какие компоненты (модули, функции, классы) нужно разбить программу? Очевидно, что с ростом количества компонентов сложность программы растет. Особенно негативны последствия неоправданного разбиения на компоненты, когда оказываются разделенными действия, по сути тесно связанные между собой.

Вторая проблема связана с организацией взаимодействия между компонентами. Взаимодействие упрощается и его легче контролировать, если каждый компонент рассматривается как «черный ящик», внутреннее устройство которого неизвестно, но известны выполняемые им функции, а также «входы» и «выходы». Вход компонента позволяет ввести в него значение некоторой *входной переменной*, а выход — получить значение *выходной переменной*. Совокупность входов и выходов черного ящика определяет *интерфейс* компонента. Интерфейс реализуется как набор функций (или *запросов* к компоненту), вызывая которые, *клиент* либо получает какую-то информацию, либо меняет состояние компонента.

Здесь термин «клиент» означает просто компонент, использующий услуги другого компонента, выполняющего в этом случае роль *сервера*. Взаимоотношение *клиент–сервер* довольно старо и использовалось уже в рамках структурного подхода, когда функция-клиент пользовалась услугами функции-сервера путем ее вызова.

Подытожим сказанное. Для оценки качества программного проекта нужно учитывать, кроме всех прочих, следующие два показателя.

Сцепление (cohesion) внутри компонента — характеризует степень взаимосвязи его отдельных частей. Пример: если внутри компонента решаются две подзадачи, которые можно легко разделить, то он обладает слабым (плохим) сцеплением.

Связанность (coupling) между компонентами — описывает интерфейс между компонентом-клиентом и компонентом-сервером. Общее количество входов и выходов сервера есть мера связанности. Чем меньше связанность между компонентами, тем проще понять и отследить их взаимодействие. Так как в больших проектах разные компоненты разрабатываются разными людьми, уменьшать связанность между компонентами очень важно.

Заметим, что описанные показатели имеют относительный характер и пользоваться ими следует благоразумно. Например, стремление максимально увеличить сцепление может привести к дроблению проекта на очень большое количество мелких функций.

Почему в случае функциональной декомпозиции трудно достичь слабой связанности между компонентами? Дело в том, что в сложной программной системе, реализующей структурную модель, практически невозможно обойтись без связи через глобальные структуры данных, а это означает, что любая функция в случае ошибки может их испортить. Такие ошибки очень трудно обнаружить. Добавьте к этому головную боль для сопровождающего программиста, который должен помнить десятки, если не сотни, обращений к общим данным из разных частей проекта. Соответственно, модификация существующего проекта в связи с новыми требованиями заказчика также потребует очень большой работы.

Другой проблемой в проектах с функциональной декомпозицией является «проклятие» общего глобального пространства имен. Члены команды, работающей над проектом, должны тратить немалые усилия по согласованию применяемых имен для своих функций, чтобы они были уникальными в рамках всего проекта.

Что принесло с собой ООП

Первым бросающимся в глаза отличием ООП от структурного программирования является использование классов. *Класс* — это тип, определяемый программистом, в котором объединяются структуры данных и функции их обработки. Конкретные переменные типа данных «класс» называются *экземплярами класса*, или *объектами*. Программы, разрабатываемые на основе концепций ООП, реализуют алгоритмы, описывающие взаимодействие между объектами.

Класс содержит константы и переменные, называемые *полями*, а также выполняемые над ними операции и функции. Функции класса называются *методами*¹. Предполагается, что доступ к полям класса возможен только через вызов соответствующих методов. Поля и методы являются *элементами (членами)* класса.

Эффективным механизмом ослабления связанности между компонентами при объектно-ориентированной декомпозиции является *инкапсуляция* — ограничение доступа к данным и их объединение с методами, обрабатывающими эти данные. Доступ к отдельным частям класса регулируется с помощью ключевых слов `public` (открытая часть), `private` (закрытая часть) и `protected` (защищенная часть).

Методы, расположенные в открытой части, формируют *интерфейс* класса и могут свободно вызываться клиентом через соответствующий объект класса. Доступ к закрытой секции класса возможен только из его собственных методов, а к защищенной — из его собственных методов, а также из методов классов-потомков (механизму наследования посвящен следующий семинар).

Инкапсуляция повышает надежность программ, предотвращая непреднамеренный ошибочный доступ к полям объекта. Кроме этого, программу легче модифицировать, поскольку при сохранении интерфейса класса можно менять его реализацию, и это не затронет внешний программный код (код клиента).

С понятием инкапсуляции тесно связано понятие *сокрытия информации*. С другой стороны, это понятие соприкасается с понятием *разделения ответственности*

¹ Широко используется и другое название — *функции-члены*, возникшее при подстрочном переводе англоязычной литературы.

между клиентом и сервером. Клиент не обязан знать, *как реализованы* те или иные методы в сервере. Ему достаточно знать, *что делает* данный метод и как к нему *обратиться*. При хорошем проектировании имена методов обычно отражают суть выполняемой ими работы, поэтому сопровождающий программист может читать код клиента как художественную литературу.

Заметим, что класс обеспечивает локальную (в пределах класса) область видимости имен. Поэтому методы, реализующие схожие подзадачи в разных классах, могут иметь одинаковые имена. То же относится и к полям разных классов. Таким образом, «проклятия общего глобального пространства имен» здесь попросту нет.

С ООП связаны еще два инструмента, грамотное использование которых повышает качество проектов: наследование классов и полиморфизм. *Наследование* — механизм получения нового класса из существующего путем его дополнения или изменения. Благодаря этому возможно повторное использование кода. Наследование позволяет создать иерархии родственных типов, совместно использующих код и интерфейсы.

Полиморфизм дает возможность создавать множественные определения для операций и функций. Какое именно определение будет использоваться, зависит от контекста программы. Вы уже знакомы с двумя разновидностями полиморфизма — перегрузкой функций и шаблонами функций. ООП дает еще три возможности: перегрузку операций, использование виртуальных методов и шаблоны классов. Перегрузка операций позволяет применять для собственных классов те же операции, что используются для встроенных типов C++. Виртуальные методы обеспечивают возможность выбрать на этапе выполнения нужный метод среди одноименных методов базового и производного классов. Шаблоны классов позволяют описать классы, инвариантные относительно типов данных.

Отметим, что в реальном проекте, разработанном на базе объектно-ориентированной декомпозиции, находится место и для процедурной декомпозиции (например, при реализации сложных методов).

От структуры — к классу

Прообразом класса в C++ является структура в C. В то же время в C++ структура обрела новые свойства и теперь является частным видом класса. Со структурой `struct` в C++ можно делать все, что можно делать с классом. Тем не менее обычно структуры в C++ используют лишь для удобства работы с небольшими наборами данных без какого-либо собственного поведения.

Новые понятия легче изучать, отталкиваясь от уже освоенного материала. Давайте возьмем задачу 6.1 и посмотрим, что можно с ней сделать, применив средства ООП.

Задача 10.1. Поиск в простой базе (массив объектов)

В текстовом файле хранится база отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике.

Формат записи: фамилия и инициалы (30 поз., фамилия должна начинаться с первой позиции), год рождения (5 поз.), оклад (10 поз.). Написать программу, которая по заданной фамилии выводит на экран сведения о сотруднике, подсчитывая средний оклад всех запрошенных сотрудников.

Для начала преобразуем в класс структуру Man, которая применялась в листинге 6.1 для хранения сведений об одном сотруднике. Мы предполагаем, что наш новый тип будет обладать более сложным поведением, чем просто чтение и запись его полей:

```
class Man {
    char name[l_name + 1];
    int birth_year;
    float pay;
};
```

Все поля класса по умолчанию закрыты (private), поэтому, если клиентская функция main объявит объект Man man, а потом попытается обратиться к какому-либо его полю, например man.pay = value, произойдет ошибка компиляции. Поэтому в состав класса надо добавить методы доступа к его полям. Эти методы должны быть общедоступными, или открытыми (public).

Но сначала критически рассмотрим определения полей. Поле name объявлено как статический массив. Это не очень гибкое решение. Желательно предусмотреть простой механизм настройки на произвольное значение длины l_name, чтобы класс Man можно было применять в разных приложениях. Используем динамический массив символов (char* pName). Возникает вопрос: кто и где будет выделять память под этот массив? Один из принципов ООП гласит, что все объекты должны быть самодостаточными, то есть полностью себя обслуживать.

Таким образом, в состав класса необходимо включить метод, который обеспечит выделение памяти под динамический массив при создании объекта (переменной типа Man). Метод, который автоматически вызывается при создании экземпляра класса, называется *конструктором*. Имя конструктора совпадает с именем класса. Парным конструктору является метод, называемый *деструктором*, который автоматически вызывается перед уничтожением объекта. Имя деструктора отличается от имени конструктора только наличием предвещающего символа ~ (тильда).

Ясно, что если в конструкторе была выделена динамическая память, то в деструкторе нужно побеспокоиться о ее освобождении. Напомним, что объект, созданный как локальная переменная в некотором блоке, уничтожается, когда при выполнении достигнут конец блока. Если же объект создан с помощью операции new, например, Man* pMan = new Man;, то для его уничтожения применяется операция delete. Итак, наш класс принимает следующий вид:

```
class Man {
public:
    Man(int lName = 30) { pName = new char[lName + 1]; } // конструктор
    ~Man() { delete [] pName; } // деструктор
```

```
private:
    char* pName;
    int   birth_year;
    float pay;
}; // объявление класса должно завершаться точкой с запятой!
```

Отметим, что здесь методы класса *определены как встроенные (inline)*. При другом способе описания методы только *объявляются* внутри класса, а их *реализация* записывается вне определения класса, как показано ниже:

```
////////// Man.h (интерфейс класса) //////////
class Man {
public:
    Man( int lName = 30 );           // конструктор
    ~Man();                         // деструктор
private:
    char* pName;
    int   birth_year;
    float pay;
};
////////// Man.cpp (реализация класса) //////////
#include "Man.h"
Man::Man( int lName ) { pName = new char[lName + 1]; }
Man::~Man() { delete [] pName; }
```

При внешнем определении метода перед его именем указывается имя класса, за которым следует операция доступа к области видимости ::. Выбор способа определения метода зависит в основном от его размера: короткие методы можно определить как встроенные, что может привести к более эффективному коду.

Продолжим процесс проектирования интерфейса нашего класса. Какие методы нужно добавить в класс? С какими сигнатурами? На этом этапе полезно задаться вопросом: *какие обязанности* должны быть возложены на класс Man? Первую обязанность мы уже реализовали: объект класса хранит сведения о сотруднике. Чтобы ими воспользоваться, клиент должен иметь возможность получить эти сведения, изменить их и вывести на экран. Кроме этого, для поиска сотрудника желательно иметь возможность сравнивать его имя с заданным.

Начнем с методов, обеспечивающих доступ к полям класса. Для *считывания* значений полей добавим методы GetName, GetBirthYear, GetPay. Очевидно, что аргументы им не нужны, а тип возвращаемого значения совпадает с типом поля. Для *записи* значений полей добавим методы SetName, SetBirthYear, SetPay. Чтобы определиться с их сигатурой, представим себе, как они будут вызываться клиентом. Вспомните, как вводились данные в задаче 6.1 (см. листинг 6.1, цикл, помеченный комментарием 2). Поступив аналогично, получим в теле цикла код вида

```
man[i].SetName( buf );
man[i].SetBirthYear( atoi( &buf[l_name] ) );
man[i].SetPay( atof( &buf[l_name + l_year] ) );
```

¹ *Сигнатура, прототип, заголовок функции* — синонимы.

Вряд ли такое решение можно признать удачным. Ведь аргументы вида `atoi(&buf[l_name])` фактически являются сущностями типа «как делать», а не «что делать»! Иными словами, крайне нежелательно нагружать клиента избыточной информацией. Детали реализации должны быть спрятаны (инкапсулированы) внутрь класса. Поэтому в данной задаче более удачной является сигнатура метода `void SetBirthYear(const char* fromBuf)`. Аналогичные размышления можно повторить и для метода `SetPay`.

СОВЕТ

Распределяя обязанности между клиентом и сервером, инкапсулируйте подробности реализации в серверном компоненте.

Теперь перейдем к функции `main`, решающей подзадачу поиска сотрудника в базе по заданной фамилии и вывода сведений о нем (*операторы 7–10* листинга 6.1). Напомним, что в задаче 6.1 поиск сотрудника с заданным именем был реализован с помощью следующего цикла (для удобства имя `dbase` изменено на `man`):

```
for ( int i = 0; i < n_record; ++i ) {
    if ( strstr( man[i].name, name ) )                // 1
        if ( man[i].name[strlen(name)] == ' ' ) {    // 2
            strcpy( name, man[i].name );              // 3
            cout << name << man[i].birth_year << ' ' << man[i].pay << endl; // 4
            // . . .
        }
    }
}
```

Опять задумаемся над вопросом: какая информация здесь *избыточна для клиента*? Здесь решаются две задачи: сравнение имени в очередной записи с заданным именем `name` (*операторы 1 и 2*) и вывод информации (*операторы 3 и 4*). Поручим решение первой подзадачи методу `CompareName(const char* name)`, инкапсулировав в него подробности решения, а решение второй подзадачи — методу `Print`. Проектирование интерфейса класса `Man` завершено. Посмотрите на текст программы, в которой реализован и использован описанный выше класс `Man` (листинг 10.1).

Листинг 10.1. Проект Task10_1

```
// //////////////////////////////////////// Man.h ////////////////////////////////////////
const int l_name = 30;
const int l_year = 5;
const int l_pay = 10;
const int l_buf = l_name + l_year + l_pay;
class Man {
public:
    Man(int lName = 30);
    ~Man();
    bool CompareName(const char*) const;
    int GetBirthYear() const { return birth_year; }
    float GetPay() const { return pay; }
```

```

    char* GetName() const { return pName; }
    void Print() const;
    void SetBirthYear(const char*);
    void SetName(const char*);
    void SetPay(const char*);
private:
    char* pName;
    int birth_year;
    float pay;
};
// //////////////////////////////////////// Man.cpp ////////////////////////////////////////
#include <iostream>
#include <cstring>
#include "Man.h"
using namespace std;
Man::Man( int lName ) {
    cout << "Constructor is working... ";
    pName = new char[lName + 1];
}
Man::~Man() {
    cout << "Destructor is working...";
    delete [] pName;
}
void Man::SetName(const char* fromBuf) {
    strncpy(pName, fromBuf, l_name);
    pName[l_name] = 0;
}
void Man::SetBirthYear(const char* fromBuf) {
    birth_year = atoi(fromBuf + l_name);
}
void Man::SetPay(const char* fromBuf) {
    pay = atof(fromBuf + l_name + l_year);
}
bool Man::CompareName(const char* name) const {
    if ((strstr(pName, name)) && (pName[strlen(name)] == ' ')) return true;
    else return false;
}
void Man::Print() const {
    cout << pName << birth_year << ' ' << pay << endl;
}
// //////////////////////////////////////// Main.cpp ////////////////////////////////////////
#include <windows.h> // содержит прототип OemToChar
#include <iostream>
#include <fstream>
#include "Man.h"
using namespace std;
const char filename[] = "dbase.txt";
int main() {
    const int maxn_record = 10;
    Man man[maxn_record];

```

продолжение ➤

Листинг 10.1 (продолжение)

```

char buf [l_buf + 1];
char name[l_name + 1];
setlocale(LC_ALL, "Russian");
ifstream fin(filename);
if (!fin) { cout << "Нет файла " << filename << endl; return 1; }
int i = 0;
while (fin.getline( buf, l_buf )) {
    if (i >= maxn_record) { cout << "Слишком длинный файл"; return 1; }
    man[i].SetName( buf );
    man[i].SetBirthYear( buf );
    man[i].SetPay( buf );
    i++;
}
int n_record = i, n_man = 0;
float mean_pay = 0;
while (true) {
    cout << "Введите фамилию или слово end: ";
    cin >> name;
    OemToChar(name, name); // для ввода кириллицы
    if (0 == strcmp(name, "end")) break;
    bool not_found = true;
    for (int i = 0; i < n_record; ++ i ) {
        if (man[i].CompareName(name)) {
            man[i].Print();
            n_man++; mean_pay += man[i].GetPay();
            not_found = false;
            break;
        }
    }
    if (not_found) cout << "Такого сотрудника нет" << endl;
}
if (n_man) cout << " Средний оклад: " << mean_pay / n_man << endl;
}

```

Обратите внимание на следующие моменты.

Константные методы. Заголовки методов класса, которые *не должны изменять поля* класса, снабжены модификатором `const` после списка параметров. Если вы по ошибке попытаетесь в теле метода что-либо присвоить полю класса, компилятор не позволит вам это сделать. Другое достоинство ключевого слова `const` — облегчение сопровождения программы. Например, если обнаружено некорректное поведение приложения и выяснено, что «кто-то» портит одно из полей объекта, можно сразу исключить из списка подозреваемых константные методы класса. Поэтому использование `const` в объявлениях методов, не изменяющих объект, считается хорошим стилем программирования.

Отладочная печать в конструкторе и деструкторе. Вывод сообщений типа «Constructor is working», «Destructor is working» очень помогает на начальном этапе освоения классов и при поиске труднодиагностируемых ошибок.

Отладка программы. С технологией создания многофайловых проектов вы уже знакомы по первой части практикума. Создайте проект с именем Task10_1 и добавьте к нему файлы из листинга 10.1. Поместите в текущий каталог проекта файл с базой данных dbase.txt, заполнив его соответствующей информацией, например:

Иванов И.П.	1957	4200
Петров А.Б.	1947	3800
Сидоров С.С.	1938	3000
Ивановский Р.Е.	1963	6200

Скомпилируйте и запустите программу на выполнение. Вы должны увидеть следующий вывод в консольном окне программы:

```
Constructor is working... Constructor is working... Constructor is working...
Constructor is working... Constructor is working... Constructor is working...
Constructor is working... Constructor is working... Constructor is working...
Constructor is working... Введите фамилию или слово end:
```

Благодаря отладочной печати мы можем проконтролировать, как создаются все 10 элементов массива man. Продолжим тестирование программы. Введите с клавиатуры текст: «Сидоров». После нажатия Enter вы должны увидеть на экране новый текст:

```
Сидоров С.С.      1938   3000
Введите фамилию или слово end:
```

Введите с клавиатуры текст «Петров» и нажмите Enter. Появятся две строки:

```
Петров А.Б.      1947   3800
Введите фамилию или слово end:
```

Введите с клавиатуры текст «end». После нажатия Enter появится вывод:

```
Средний оклад: 3400
Destructor is working... Destructor is working... Destructor is working...
Destructor is working... Destructor is working... Destructor is working...
Destructor is working... Destructor is working... Destructor is working...
Destructor is working... Press any key to continue
```

Все верно! После ввода end программа покидает цикл while (true) и печатает величину среднего оклада для вызванных ранее записей из базы данных. Затем программа завершается, а при выходе из области видимости все объекты вызывают свои деструкторы, и мы это тоже четко наблюдаем.

ПРИМЕЧАНИЕ

В реальных программах доступ к полям класса с помощью методов преследует еще одну важную цель — проверку заносимых значений. В качестве упражнения дополните методы SetBirthYear и SetPay проверками успешности преобразования из строки в число и осмысленности получившегося результата (например, на неотрицательность).

Итак, мы завершили перестройку структурной программы в программу, реализующую концепции ООП. Рекомендуем вам сравнить решение задачи 6.1 с новым,

начав со ввода исходных данных. В новой программе все понятно без лишних комментариев. В старой — нужно разбираться на уровне клиента с реализацией более низкого уровня. Перейдем теперь к рассмотрению других аспектов, необходимых для создания хорошо спроектированного класса.

Конструктор по умолчанию

Обратите внимание, что в списке параметров конструктора класса `Man` параметр `!Name` имеет значение по умолчанию. Если все параметры конструктора имеют значения по умолчанию или если конструктор вовсе не имеет параметров, он называется *конструктором по умолчанию*. Такой конструктор имеет несколько специальных областей применения.

Во-первых, он используется при описании массива объектов, например: `Man man[25]`. Здесь каждый из 25 объектов создается путем вызова конструктора по умолчанию. В случае отсутствия конструктора по умолчанию вы *не сможете объявлять массивы* объектов этого класса (если вы не определите в классе вообще ни одного конструктора, то компилятор создаст конструктор по умолчанию самостоятельно). Во-вторых, вы *не сможете использовать* объекты этого класса *в качестве полей другого класса*. Еще одно применение конструкторов по умолчанию относится к механизму наследования классов (см. семинар 11).

Инициализаторы конструктора

Существует альтернативный способ инициализации отдельных частей объекта в конструкторе — с помощью *списка инициализаторов*, расположенного после двоеточия между заголовком и телом конструктора. Каждый инициализатор имеет вид `имя_поля (выражение)` и обеспечивает присвоение полю объекта значения указанного выражения. Если инициализаторов несколько, они разделяются запятыми. Если полем объекта является объект другого класса, для его инициализации будет вызван соответствующий конструктор.

Рассмотрим класс `Point`, задающий точку на плоскости в декартовых координатах:

```
class Point {
    double x, y;
public:
    Point( double _x = 0, double _y = 0 ) { x = _x; y = _y; }
    // Point(double _x = 0, double _y = 0) : x(_x), y(_y) {} // 1
    ...
};
```

В *операторе 1* записан вариант со списком инициализации. Тело конструктора пустое, а действия по присваиванию начальных значений полям перенесены в инициализаторы. Кстати, в нашем примере все параметры конструктора имеют значения по умолчанию, благодаря чему он может использоваться и как конструктор с параметрами, и как конструктор по умолчанию.

В этом примере может быть выбран любой вариант конструктора, однако есть три ситуации, в которых инициализация полей объекта возможна *только с использованием*

инициализаторов конструктора: для констант, для ссылок и для полей, являющихся объектами класса, в котором есть один или более конструкторов, но отсутствует конструктор по умолчанию.

Инициализация элементов класса с помощью списка инициализаторов является более эффективной по быстродействию, чем присваивание начальных значений в теле конструктора. Это объясняется особенностями *порядка инициализации объектов*. Рассмотрим этот порядок на примере. В общем случае класс содержит как поля стандартных типов, так и поля, являющиеся объектами других классов:

```
class Coo {
    int    x1;
    double x2;
    Point  p1, p2;
public:
    Coo( int _x1, double _x2 ) { x1 = _x1; x2 = _x2; }
    ...
};
```

При создании объекта класса Coo сначала вызываются конструкторы объектных полей (конструкторы по умолчанию класса Point для полей p1 и p2), а затем — конструктор класса Coo.

Если конструктор класса имеет список инициализаторов, то порядок инициализации включенных в него объектов определяется не самим списком, а тем, в каком порядке эти поля объявлены в теле класса. Поэтому во избежание путаницы рекомендуется записывать список инициализаторов в порядке, соответствующем объявлению.

Если элемент класса, являющийся объектом, инициализируется не в списке инициализаторов, а в теле конструктора, то сначала он получает значение при помощи конструктора по умолчанию своего класса, а потом это значение изменяется в теле конструктора (ясно, что эффективность при этом ухудшается).

Конструктор копирования

Конструктор так же, как и остальные методы класса, можно перегружать. Одной из важнейших форм перегружаемого конструктора является *конструктор копирования (copy constructor)*. Он вызывается в трех ситуациях:

- ☐ при инициализации нового объекта другим объектом в операторе объявления;
- ☐ при передаче объекта в функцию в качестве аргумента по значению;
- ☐ при возврате объекта из функции по значению.

Если при объявлении класса конструктор копирования не задан, компилятор создает *конструктор копирования по умолчанию*, который дублирует объект, выполняя побайтное копирование его полей. Однако при этом возможно появление проблем, связанных с копированием указателей. Например, если объект, передаваемый в функцию в качестве аргумента по значению, содержит указатель pMem на выделенную область памяти, то в копии объекта этот указатель будет ссылаться на ту же самую область. При возврате из функции вызывается деструктор копии

объекта, освобождающий память, на которую указывает `pMem`. При выходе из области видимости исходного объекта его деструктор попытается еще раз освободить память, на которую указывает `pMem`. Результат обычно весьма плачевный.

Чтобы исключить нежелательные побочные эффекты в каждой из трех перечисленных ситуаций, необходимо создать конструктор копирования, в котором реализуется необходимый контроль за созданием копии объекта.

Конструктор копирования имеет следующую общую форму:

```
имя_класса( const имя_класса & obj ) {
    // тело конструктора
}
```

Здесь `obj` — ссылка на объект, для которого должна создаваться копия. Например, пусть имеется класс `Сoo`, а `y` — объект типа `Сoo`. Тогда следующие операторы вызовут конструктор копирования класса `Сoo`:

```
Сoo x = y;           // y явно инициализирует x
Func1( y );          // y передается в качестве аргумента по значению
y = Func2();          // y получает возвращаемый объект
```

В двух первых случаях конструктору копирования передается ссылка на `y`. В последнем случае конструктору копирования передается ссылка на объект, возвращаемый функцией `Func2`. Пример реализации конструктора копирования будет показан при решении задачи 10.2.

Перегрузка операций

Любая операция, определенная в C++, за исключением `::`, `?:`, `.`, `*`, `#`, `##`, может быть перегружена для созданного вами класса. Это делается с помощью функций специального вида, называемых *функциями-операциями* (операторными функциями):

```
возвращаемый_тип operator # (список параметров) { тело функции }
```

Вместо знака `#` ставится знак перегружаемой операции. Функция-операция может быть реализована либо как функция класса, либо как внешняя (обычно дружественная) функция. В первом случае количество параметров у функции-операции на единицу меньше, так как первым операндом при этом считается сам объект, вызвавший данную операцию.

Покажем два варианта перегрузки операции сложения для класса `Point`:

```
class Point {                                     // Вариант 1: в форме метода класса
    double x, y;
public:
    //...
    Point operator +( Point& );
};
Point Point::operator +( Point& p ) {
    return Point( x + p.x, y + p.y );
}
class Point {                                     // Вариант 2: в форме внешней глобальной функции
```

```
//. . .
    friend Point operator +( Point&, Point& );
};
Point operator +( Point& p1, Point& p2 ) {
    return Point( p1.x + p2.x, p1.y + p2.y );
}
```

Внешняя глобальная функция, как правило, объявляется дружественной классу, чтобы иметь доступ к его закрытым элементам. Независимо от формы реализации операции «+» теперь можно написать

```
Point p1( 0, 2 ), p2( -1, 5 );
Point p3 = p1 + p2;
```

Важно понимать, что, встретив выражение `p1 + p2`, компилятор в случае первой формы перегрузки вызовет метод `p1.operator +(p2)`, а в случае второй формы перегрузки — глобальную функцию `operator +(p1, p2)`. Результатом выполнения данных операторов будет точка `p3` с координатами $x = -1, y = 7$. Заметим, что для инициализации объекта `p3` будет вызван конструктор копирования по умолчанию, но он нас устраивает, поскольку в классе нет полей-указателей.

Какую из двух форм следует выбирать? Используйте перегрузку в форме метода класса, если нет каких-либо причин, препятствующих этому. Например, если первый аргумент (левый операнд) относится к одному из базовых типов (к примеру, `int`), то перегрузка операции возможна только в форме внешней функции.

Перегрузка операций инкремента

Операция инкремента имеет две формы: *префиксную* и *постфиксную*. Для первой формы сначала изменяется состояние объекта в соответствии с операцией, а затем объект используется в выражении. Для второй формы объект используется в том состоянии, которое он имел до начала операции, а потом его состояние изменяется.

Чтобы компилятор смог различить эти две формы, для них используются разные сигнатуры. Покажем реализацию операций инкремента на примере класса `Point`:

```
Point& Point::operator ++() {           // префиксный инкремент
    x++;    y++;
    return *this;
}
Point Point::operator ++( int ) {       // постфиксный инкремент
    Point old = *this;
    x++;    y++;
    return old;
}
```

В префиксной операции выполняется возврат результата по ссылке. Это предотвращает вызов конструктора копирования для создания возвращаемого значения и последующий вызов деструктора. В постфиксной операции возврат по ссылке не подходит, поскольку необходимо вернуть первоначальное состояние объекта,

сохраненное в локальной переменной `old`. Таким образом, *префиксный инкремент является более эффективной операцией, чем постфиксный инкремент*.

Заметим, что ранее мы спокойно пользовались постфиксной формой инкремента, например, в заголовке цикла `for`, поскольку для переменных встроенного типа разницы в эффективности нет. Когда мы будем применять контейнерные классы стандартной библиотеки, параметр в заголовке цикла часто будет представлять объект-итератор, и для него префиксная форма будет более эффективной.

ПРИМЕЧАНИЕ

Все сказанное об операции инкремента относится также и к *операции декремента*.

Перегрузка операции присваивания

О перегрузке этой операции следует поговорить особо по нескольким причинам. Во-первых, если вы не определите эту операцию в разрабатываемом классе, *компилятор создаст операцию присваивания по умолчанию*, которая выполняет поэлементное копирование объекта. При этом возможны те же проблемы, что возникают при использовании конструктора копирования по умолчанию (см. выше). Поэтому запомните правило: если в классе требуется определить конструктор копирования, его верной спутницей должна быть операция присваивания, и наоборот. Во-вторых, операцию присваивания *можно определить только в форме метода класса*, и, в-третьих, *она не наследуется*, в отличие от всех остальных операций.

Определим операцию присваивания для класса `Man` из задачи 10.1:

```
class Man { //////////////////////////////////////////////////// Man.h (интерфейс класса) //
public:
    // . . .
    Man& operator =( const Man& ); // операция присваивания
private:
    char* pName;
    // . . .
};
////////////////////////////////////////////////// Man.cpp (реализация класса) //
Man& Man::operator =( const Man& man ) {
    if ( this == &man ) return *this; // проверка на самоприсваивание
    delete [] pName; // уничтожение предыдущего значения
    pName = new char[strlen( man.pName ) + 1];
    strcpy( pName, man.pName );
    birth_year = man. birth_year;
    pay = man.pay;
    return *this;
}
```

Обратите внимание на несколько важных моментов:

- ❑ убедитесь, что не выполняется присваивание вида `x = x;`. Если левая и правая части ссылаются на один и тот же объект, то делать ничего не надо. Если не пере-

хватить этот особый случай, то следующий шаг уничтожит значение, на которое указывает `pName`, еще до того, как оно будет скопировано;

- ☐ удалите предыдущие значения полей в динамически выделенной памяти;
- ☐ выделите память под новые значения полей и скопируйте в нее эти значения;
- ☐ возвратите значение объекта, на которое указывает `this` (то есть `*this`).

Статические элементы класса

Если необходимо создать переменную, значение которой будет общим для всех объектов данного класса, следует *объявить* в классе поле с модификатором `static` и дать его *определение* в глобальной области видимости программы:

```
class Coo {  
    static int count;           // объявление в классе  
    // остальной код  
};  
int Coo::count = 1;           // определение и инициализация  
// int Coo::count;           // по умолчанию инициализируется нулем
```

Просто воспользоваться для этой цели глобальной переменной будет неграмотно, это нарушит принцип инкапсуляции данных. Аналогично статическим полям, могут быть объявлены и статические методы класса (с модификатором `static`). Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель `this`. Статические методы не могут быть константными (`const`) и виртуальными (`virtual`). Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

Все рассмотренные выше аспекты найдут применение при решении задачи 10.2.

Задача 10.2. Реализация класса треугольников

Для некоторого множества заданных координатами своих вершин треугольников найти треугольник максимальной площади (если максимальную площадь имеют несколько треугольников, найти первый из них). Предусмотреть возможность перемещения треугольников и проверки включения одного треугольника в другой. Программа должна содержать меню, позволяющее выполнить проверку всех методов класса.

Для реализации этой задачи составить описание класса треугольников на плоскости. Предусмотреть возможность объявления в клиентской программе (`main`) экземпляра треугольника с заданными координатами вершин. Предусмотреть наличие в классе методов, обеспечивающих: 1) перемещение треугольников на плоскости; 2) определение отношения $>$ для пары заданных треугольников (мера сравнения — площадь треугольников); 3) определение отношения включения типа: «Треугольник 1 входит (не входит) в Треугольник 2».

Сейчас мы еще не готовы провести полномасштабную объектно-ориентированную декомпозицию программы: не хватает информации из следующего семинара.

Поэтому применим гибридный подход: разработку главного клиента `main` проведем по технологии функциональной декомпозиции, а функции-серверы, вызываемые из `main`, будем использовать объекты.

Начнем с выявления основных понятий (классов) для нашей задачи. Первый очевидный класс `Triangle` необходим для представления треугольников. Из нескольких способов определения треугольника на плоскости выберем самый простой — через три точки, задающие его вершины. Этот выбор опирается на понятие, отсутствующее в условии задачи, — понятие *точки*. Точку на плоскости также можно представить разными способами; остановимся на представлении парой вещественных чисел, задающих координаты точки по осям x и y .

Таким образом, с понятием точки связывается как минимум пара атрибутов. Если же представить, что можно делать с объектом типа точки — например, перемещать ее на плоскости или определять ее входжение в заданную фигуру, — то становится очевидной целесообразность определения класса `Point`.

Итак, объектно-ориентированная декомпозиция выявила два класса: `Triangle` и `Point`. Как они должны быть связаны друг с другом? На следующем семинаре взаимоотношения между классами будут рассматриваться подробно, а пока отметим, что чаще всего для двух классов имеет место одно из двух отношений: *наследования* (отношение *is a*) и *агрегации*, или *включения* (отношение *has a*).

Если класс `B` является *частным случаем* класса `A`, то говорят, что `B is a A` (например, класс треугольников есть частный вид класса многоугольников: `Triangle is a Polygon`). Если класс `A` *содержит в себе* объект класса `B`, то говорят, что `A has a B` (например, класс треугольников может содержать в себе объекты класса точек: `Triangle has a Point`).

Займемся теперь основным клиентом — `main`. Здесь мы применим функциональную декомпозицию, или технологию нисходящего проектирования: представим алгоритм как последовательность подзадач, каждой из которых соответствует вызываемая серверная функция. На начальном этапе проектирования тела этих функций могут быть заполнены «заглушками» (отладочной печатью). Если в какой-то серверной функции окажется слабое сцепление, она тоже разбивается на несколько подзадач.

То же самое выполняется и с классами, используемыми в программе: по мере реализации подзадач они пополняются необходимыми для этого методами. Такая технология облегчает отладку и поиск ошибок, сокращая общее время разработки.

В соответствии с описанной технологией представим решение задачи 10.2 как последовательность нескольких этапов. Иногда как бы по забывчивости мы будем допускать некоторые «ляпы», поскольку в процессе поиска ошибок можно глубже понять нюансы программирования с классами.

На первом этапе напишем код (листинг 10.2) для начального представления классов `Point` и `Triangle`, достаточный для того, чтобы создать несколько объектов типа `Triangle` и реализовать первый пункт меню — вывод всех объектов на экран.

Этап 1**Листинг 10.2.** Проект Task10_2

```

//////////////////////////////////// Point.h //////////////////////////////////
#ifndef POINT_H
#define POINT_H
class Point {
public:
    double x, y;
    Point(double _x = 0, double _y = 0) : x(_x), y(_y) {}           // Конструктор
    // Другие методы
    void Show() const;
    double DistanceTo(Point) const;                                // Расстояние до другой точки
};
#endif /* POINT_H */
//////////////////////////////////// Point.cpp //////////////////////////////////
#include <iostream>
#include <cmath>
#include "Point.h"
using namespace std;
void Point::Show() const { cout << " (" << x << ", " << y << ")"; }
double Point::DistanceTo(Point sp) const {
    return sqrt((x - sp.x) * (x - sp.x) + (y - sp.y) * (y - sp.y));
}
//////////////////////////////////// Triangle.h //////////////////////////////////
#ifndef TRIANGLE_H
#define TRIANGLE_H
#include "Point.h"
class Triangle {
public:
    static int count;                                             // Количество созданных объектов
    Triangle(Point, Point, Point, const char*); // Конструктор
    Triangle(const char*); // Конструктор пустого (нулевого) треугольника
    ~Triangle(); // Деструктор
    Point Get_v1() const { return v1; } // Получить значение v1
    Point Get_v2() const { return v2; } // Получить значение v2
    Point Get_v3() const { return v3; } // Получить значение v3
    char* GetName() const { return name; } // Получить имя объекта
    void Show() const; // Показать объект
private:
    char* objID; // Идентификатор объекта
    char* name; // Наименование треугольника
    Point v1, v2, v3; // Вершины
    double a; // Сторона, соединяющая v1 и v2
    double b; // Сторона, соединяющая v2 и v3
    double c; // Сторона, соединяющая v1 и v3
};

```

продолжение ➤

Листинг 10.2 (продолжение)

```

#endif /* TRIANGLE_H */
////////////////////////////////////Triangle.cpp //////////////////////////////////
// Реализация класса Triangle
#include <cmath>
#include <iostream>
#include <iomanip>
#include <string>
#include "Triangle.h"
using namespace std;
Triangle::Triangle(Point _v1, Point _v2, Point _v3, const char* ident)
    : v1(_v1), v2(_v2), v3(_v3) {
    char buf[16];
    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);
    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = v1.DistanceTo(v2);
    b = v2.DistanceTo(v3);
    c = v1.DistanceTo(v3);
    cout << "Constructor_1 for: " << objID << " (" << name << ")\n"; // для отладки
}
Triangle::Triangle(const char* ident) {
    char buf[16];
    objID = new char[strlen(ident) + 1];
    strcpy(objID, ident);
    count++;
    sprintf(buf, "Треугольник %d", count);
    name = new char[strlen(buf) + 1];
    strcpy(name, buf);
    a = b = c = 0;
    cout << "Constructor_2 for: " << objID << " (" << name << ")\n"; // для отладки
}
Triangle::~Triangle() {
    cout << "Destructor for: " << objID << endl;
    delete [] objID;
    delete [] name;
}
void Triangle::Show() const {
    cout << name << ":";
    v1.Show();    v2.Show();    v3.Show();
    cout << endl;
}
//////////////////////////////////// Main.cpp //////////////////////////////////
#include <iostream>
#include "Triangle.h"
using namespace std;

```

```

int Menu();
int GetNumber(int, int);
void ExitBack();
void Show(Triangle* [], int);
void Move(Triangle* [], int);
void FindMax(Triangle* [], int);
void IsIncluded(Triangle* [], int);
int Triangle::count = 0;           // Инициализация глобальной переменной
// ----- Главная функция
int main() {
    Point p1(0, 0);    Point p2(0.5, 1);    // Определения точек
    Point p3(1, 0);    Point p4(0, 4.5);
    Point p5(2, 1);    Point p6(2, 0);
    Point p7(2, 2);    Point p8(3, 0);
    Triangle triaA(p1, p2, p3, "triaA");    // Определения треугольников
    Triangle triaB(p1, p4, p8, "triaB");
    Triangle triaC(p1, p5, p6, "triaC");
    Triangle triaD(p1, p7, p8, "triaD");
    // Определение массива указателей на треугольники
    Triangle* pTria[] = { &triaA, &triaB, &triaC, &triaD };
    int n = sizeof (pTria) / sizeof (pTria[0]);
    setlocale( LC_ALL, "Russian" );
    bool done = false;
    while (!done) {           // Главный цикл
        switch (Menu()) {
            case 1: Show(pTria, n);           break;
            case 2: Move(pTria, n);           break;
            case 3: FindMax(pTria, n);         break;
            case 4: IsIncluded(pTria, n);      break;
            case 5: cout << "Конец работы." << endl; done = true;    break;
        }
    }
}

int Menu() { // ----- Вывод меню
    cout << "\n===== Главное меню =====" << endl;
    cout << "1 - вывести все объекты\t 3 - найти максимальный" << endl;
    cout << "2 - переместить\t\t 4 - определить отношение включения" << endl;
    cout << "\t\t 5 - выход" << endl;
    return GetNumber(1, 5);
}

int GetNumber(int min, int max) { // ----- Ввод целого числа в заданном диапазоне
    int number = min - 1;
    while (true) {
        cin >> number;
        if ((number >= min) && (number <= max) && (cin.peek() == '\n'))
            break;
        else {
            cout << "Повторите ввод (ожидается число от " << min
                << " до " << max << "):" << endl;
            cin.clear();
        }
    }
}

```

продолжение ➤

Листинг 10.2 (продолжение)

```

        while (cin.get() != '\n') {};
    }
}
return number;
}
void ExitBack() { // ----- Возврат в функцию с основным меню
    cout << "Нажмите Enter." << endl;
    cin.get(); cin.get();
}
void Show(Triangle* p_tri[], int k) { // ----- Вывод всех треугольников
    cout << "===== Перечень треугольников =====" << endl;
    for (int i = 0; i < k; ++i) p_tri[i]->Show();
    ExitBack();
}
void Move(Triangle* p_tri[], int k) { // ----- Перемещение
    cout << "===== Перемещение =====" << endl;
    // здесь будет код функции...
    ExitBack();
}
void FindMax(Triangle* p_tri[], int k) { // ----- Поиск максимального треугольника
    cout << "=== Поиск максимального треугольника ===" << endl;
    // здесь будет код функции...
    ExitBack();
}
void IsIncluded(Triangle* p_tri[], int k) { // --- Определение отношения включения
    cout << "===== Отношение включения =====" << endl;
    // здесь будет код функции...
    ExitBack();
}
}

```

Рекомендуем вам обратить внимание на следующие моменты в проекте Task10_2.

Класс Point (файлы Point.h, Point.cpp). Реализация класса содержит единственный метод Show, назначение которого — показать объект типа Point на экране. Заметим, что при решении реальных задач в какой-либо графической оболочке метод Show действительно нарисовал бы точку. Однако мы связаны стандартом C++, поэтому демонстрация точки сводится к выводу ее координат.

Класс Triangle (файлы Triangle.h, Triangle.cpp).

- ❑ Назначение большинства полей и методов очевидно из их имен и комментариев.
- ❑ Поле count исполняет роль глобального счетчика создаваемых объектов (свойство глобальности обеспечивается благодаря модификатору static); мы сочли удобным в конструкторах генерировать имена треугольников автоматически: «Треугольник 1», «Треугольник 2», и т. д., используя текущее значение count (возможны и другие способы именования треугольников).
- ❑ Поле char* objID введено для целей отладки и обучения; вскоре вы увидите, что благодаря отладочным операторам печати в конструкторах и деструкторе удобно наблюдать за созданием и уничтожением объектов.

- ❑ Конструктор пустого (нулевого) треугольника нужен для создания временных объектов, которые могут модифицироваться с помощью присваивания.
- ❑ Метод Show — к сожалению, здесь нам тоже не удастся нарисовать треугольник на экране; вместо этого печатаются координаты его вершин.

Основной модуль (файл Main.cpp).

- ❑ Инициализация глобальных переменных: обратите внимание на оператор `int Triangle::count = 0;` — если вы его забудете, компилятор обидится.
- ❑ Функция `main`:
 - определения восьми точек `p1, ..., p8` сделаны произвольно, но так, чтобы из них можно было составить треугольники;
 - определения четырех треугольников сделаны тоже произвольно, впоследствии на них будут демонстрироваться основные методы класса;
 - далее определяются массив указателей `Triangle* pTria[]` с адресами объявленных выше треугольников и его размер `n`; в таком виде удобно передавать адрес `pTria` и величину `n` в вызываемые серверные функции.
- ❑ Функция `Menu` — после вывода на экран списка пунктов меню вызывается функция `GetNumber`, возвращающая номер пункта, введенный пользователем с клавиатуры. Отчего было просто не написать: `cin >> number;`? Но тогда мы не обеспечили бы защиту программы от непреднамеренных ошибок при вводе. Вообще-то вопрос надежного ввода чисел с клавиатуры подробно разбирается на семинаре 13 при решении задачи 13.1.
- ❑ Функция `Show` выводит на экран перечень всех треугольников. В завершение вызывается функция `ExitBack`, которая обеспечивает заключительный диалог с пользователем после обработки очередного пункта меню.
- ❑ Остальные функции по обработке других пунктов меню выполнены в виде заглушек, выводящих только наименование соответствующего пункта.

Тестирование и отладка первой версии программы. После компиляции и запуска программы вы должны увидеть на экране следующее:

```
Constructor_1 for: triaA (Треугольник 1)
Constructor_1 for: triaB (Треугольник 2)
Constructor_1 for: triaC (Треугольник 3)
Constructor_1 for: triaD (Треугольник 4)
```

```
===== Г л а в н о е   м е н ю =====
1 - вывести все объекты   3 - найти максимальный
2 - переместить           4 - определить отношение включения
                          5 - выход
```

Введите с клавиатуры цифру 1¹. Программа выведет:

```
1
===== Перечень треугольников =====
Треугольник 1:  (0, 0) (0.5, 1) (1, 0)
Треугольник 2:  (0, 0) (0, 4.5) (3, 0)
```

продолжение ➤

¹ После ввода числовой информации всегда подразумевается нажатие Enter.

Треугольник 3: (0, 0) (2, 1) (2, 0)
 Треугольник 4: (0, 0) (2, 2) (3, 0)
 Нажмите Enter.

Выбор первого пункта меню проверен. Нажмите Enter. Программа выведет:

```
===== Г л а в н о е   м е н ю =====
. . .
```

Теперь проверим выбор второго пункта меню. После ввода цифры 2 на экране должно появиться:

```
2
===== Перемещение =====
Нажмите Enter.
```

Все правильно. После нажатия Enter программа возвращается в главное меню.

Для проверки ввода ошибочного символа введите с клавиатуры любой буквенный символ и нажмите Enter. Программа должна сообщить:

Повторите ввод (ожидается число от 1 до 5):

Проверяем завершение работы. Введите цифру 5. Программа выведет:

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Тестирование закончено. Обратите внимание на то, что деструкторы объектов вызываются в порядке, обратном вызову конструкторов.

Этап 2

Добавим в классы Point и Triangle методы, обеспечивающие перемещение треугольников, а в основной модуль — реализацию функции Move. Внесите следующие изменения в тексты модулей проекта.

1. **Point.h.** добавьте объявление операции-функции «+=», которая будет использоваться для реализации метода Move в классе Triangle:

```
void operator +=( Point& );
```

2. **Point.cpp.** Добавьте код реализации данной функции:

```
void Point::operator +=( Point& p ) { x += p.x; y += p.y; }
```

3. **Triangle.h.** Добавьте объявление метода:

```
void Move( Point );
```

4. **Triangle.cpp.** Добавьте код метода Move:

```
void Triangle::Move(Point dp) {
    v1 += dp;    v2 += dp;    v3 += dp;
}
```

5. **Main.cpp.** В список прототипов функций добавьте сигнатуру:

```
double GetDouble();
```

Добавьте в конец файла новую функцию GetDouble. Она предназначена для ввода вещественного числа и вызывается из функции Move. В ней предусмотрена защита от ввода недопустимых символов аналогично функции GetNumber:

```
double GetDouble() {
    double value;
    while (true) {
        cin >> value;
        if (cin.peek() == '\n') break;
        else {
            cout << "Повторите ввод (ожидается вещественное число):" << endl;
            cin.clear();
            while (cin.get() != '\n') {};
        }
    }
    return value;
}
```

Замените заглушку функции Move следующим кодом:

```
void Move(Triangle* p_tria[], int k) {
    cout << "===== Перемещение =====" << endl;
    cout << "Введите номер треугольника (от 1 до " << k << "): ";
    int i = GetNumber( 1, k ) - 1;
    p_tria[i]->Show();
    Point dp;
    cout << "Введите смещение по x: ";
    dp.x = GetDouble();
    cout << "Введите смещение по y: ";
    dp.y = GetDouble();
    p_tria[i]->Move( dp );
    cout << "Новое положение треугольника:" << endl;
    p_tria[i]->Show();
    ExitBack();
}
```

Выполнив компиляцию проекта, проведите его тестирование аналогично тестированию на первом этапе. После выбора второго пункта меню и ввода данных, задающих номер треугольника, величину сдвига по x и величину сдвига по y, вы должны увидеть на экране примерно следующее (жирным шрифтом выделено то, что вводилось с клавиатуры):

```
2
===== Перемещение =====
Введите номер треугольника (от 1 до 4): 1
Треугольник 1: (0, 0) (0.5, 1) (1, 0)
Введите смещение по x: 2.5
Введите смещение по y: -7
Новое положение треугольника:
```

продолжение ➤

Треугольник 1: (2.5, -7) (3, -6) (3.5, -7)
Нажмите Enter.

Этап 3

На этом этапе мы добавим в класс `Triangle` метод, обеспечивающий сравнение треугольников по их площади, а в основной модуль — реализацию функции `FindMax`. Внесите следующие изменения в тексты модулей проекта:

1. **Triangle.h:** добавьте объявление метода `Area` и функции-операции `>`:

```
double Area() const;           // площадь объекта
bool operator >(const Triangle&) const;
```

2. **Triangle.cpp:** добавьте код реализации метода `Area` и функции-операции `>`:

```
double Triangle::Area() const {
    double p = (a + b + c) / 2;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}
bool Triangle::operator >(const Triangle& tria) const {
    if (Area() > tria.Area()) return true;
    else return false;
}
```

3. **Main.cpp:** замените заглушку функции `FindMax` следующим кодом:

```
// ----- поиск максимального треугольника
void FindMax(Triangle* p_tria[], int k) {
    cout << "=== Поиск максимального треугольника ==> << endl;
    // Создаем объект triaMax, который по завершению поиска будет идентичен
    максимальному объекту. Инициализируем его значением 1-го объекта из массива:
    Triangle triaMax("triaMax");
    triaMax = *p_tria[0];
    for (int i = 1; i < 4; ++i)
        if (*p_tria[i] > triaMax) triaMax = *p_tria[i];
    cout << "Максимальный треугольник: " << triaMax.GetName() << endl;
    ExitBack();
}
```

Откомпилируйте программу и запустите. Выберите третий пункт меню. На экране должно появиться:

```
3
=== Поиск максимального треугольника ==
Constructor 2 for: triaMax (Треугольник 5)
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Максимальный треугольник найден правильно. После нажатия `Enter` должен появиться текст:

```
Destructor for: triaB
===== Г л а в н о е   м е н ю =====
1 - вывести все объекты   3 - найти максимальный
```


2 - переместить 4 - определить отношение включения
5 - выход

Чтобы завершить работу, введите цифру 5 и нажмите Enter.

Но... вместо нормального завершения программа ломается, а операционная система выдает сообщение об ошибке примерно следующего вида:

```
Debug Assertion Failed!  
Program: . . .  
File: dbgdel.cpp  
Line: 47
```

Чтобы понять, что происходит, давайте посмотрим на нашу отладочную печать. Перед тем как умереть, программа успела вывести на экран следующее:

```
5  
Конец работы.  
Destructor for: triaD  
Destructor for: triaC  
Destructor for: ННННННННННззззЗЗЗЗЗЗЗЗЗЗЗ
```

Пришло время для аналитической работы наших серых клеточек. Вспомните, как выглядела отладочная печать при завершении нормально работающей первой версии программы? Деструкторы вызывались в порядке, обратном вызову конструкторов. Значит, какая-то беда случилась после вызова деструктора для объекта triaB! Почему? Всмотримся в предыдущий вывод нашей программы. После того как функция FindMax выполнила основную работу и вывела результат на экран:

Максимальный треугольник: Треугольник 2

программа пригласила пользователя нажать клавишу Enter. Это приглашение выводится функцией ExitBack. А вот после нажатия клавиши Enter на экране появился текст:

```
Destructor for: triaB
```

после которого опять было выведено главное меню. Значит, деструктор для объекта triaB был вызван в момент возврата из функции FindMax. Но почему? Ведь объект triaB создается в основной функции main, а значит, там же должен и уничтожаться! Что-то нехорошее происходит, однако, в теле функции FindMax. Хотя внешне вроде все прилично... Стоп! Ведь внутри функции объявлен объект triaMax, и мы даже видели работу его конструктора:

```
Constructor_2 for: triaMax (Треугольник 5)
```

А где же вызов деструктора, который по идее должен произойти в момент возврата из функции FindMax? Кажется, мы нашли причину ненормативного поведения нашей программы. Объект triaMax после своего объявления неоднократно модифицируется с помощью операции присваивания. А теперь давайте вспомним, что, если мы не перегрузили операцию присваивания для некоторого класса, компилятор сделает это за нас, но в ней будут поэлементно копироваться все поля объекта. При наличии же полей типа указателей возможны неприятные проблемы.

В данном случае в поля `objID` и `name` объекта `triaMax` были скопированы значения одноименных полей объекта `triaB`. В момент выхода из функции `FindMax` деструктор объекта освободил память, на которую указывали эти поля. А при выходе из основной функции `main` деструктор объекта `triaB` попытался еще раз освободить эту же память. Результат вы уже видели...

Займемся починкой нашей программы. Нужно добавить в класс `Triangle` перегрузку операции присваивания, а заодно и конструктор копирования. Внесите следующие изменения в тексты модулей проекта:

4. **Triangle.h.** Добавьте объявления:

```
Triangle( const Triangle& );           // Конструктор копирования
Triangle& operator =( const Triangle& ); // Операция присваивания
```

5. **Triangle.cpp.** Добавьте реализацию:

```
// Конструктор копирования
Triangle::Triangle(const Triangle& tria) : v1(tria.v1), v2(tria.v2), v3(tria.v3) {
    cout << "Copy constructor for: " << tria.objID << endl;    // Отладочный вывод
    objID = new char[strlen(tria.objID) + strlen("(копия)") + 1];
    strcpy(objID, tria.objID);
    strcat(objID, "(копия)");
    name = new char[strlen(tria.name) + 1];
    strcpy(name, tria.name);
    v1 = tria.v1; v2 = tria.v2; v3 = tria.v3;
    a = tria.a;   b = tria.b;   c = tria.c;
}
// Присвоить значение объекта tria
Triangle& Triangle::operator =(const Triangle& tria) {
    // отладочный вывод:
    cout << "Assign operator: " << objID << " = " << tria.objID << endl;
    if (&tria == this) return *this;
    delete [] name;
    name = new char[strlen(tria.name) + 1];
    strcpy(name, tria.name);
    v1 = tria.v1; v2 = tria.v2; v3 = tria.v3;
    a = tria.a;   b = tria.b;   c = tria.c;
    return *this;
}
```

И в конструкторе копирования, и в операции присваивания перед копированием содержимого полей, на которые указывают поля типа `char*`, для них выделяется новая память. Обратите внимание, что в конструкторе копирования после переписи поля `objID` мы добавляем в конец этой строки текст (копия). А в операции присваивания это поле, идентифицирующее объект, вообще не затрагивается и остается в своем первоначальном значении. Все это полезно для целей отладки.

Откомпилируйте и запустите программу. Выберите третий пункт меню. На экране должен появиться текст:

```
3
=== Поиск максимального треугольника ==
```

```
Constructor_2 for: triaMax (Треугольник 5)
Assign operator: triaMax = triaA
Assign operator: triaMax = triaB
Максимальный треугольник: Треугольник 2
Нажмите Enter.
```

Обратите внимание на отладочный вывод операции присваивания. Продолжим тестирование. После нажатия **Enter** программа выведет

```
Destructor for: triaMax
===== Г л а в н о е   м е н ю =====
1 - вывести все объекты   3 - найти максимальный
2 - переместить           4 - определить отношение включения
                           5 - выход
```

Заметьте, что был вызван деструктор для объекта `triaMax`, а не `triaB`. Продолжим тестирование. Введите цифру 5. Программа выведет

```
5
Конец работы.
Destructor for: triaD
Destructor for: triaC
Destructor for: triaB
Destructor for: triaA
```

Все. Программа работает как часы. Нам осталось решить последнюю задачу — определение *отношения включения* одного треугольника в другой.

Этап 4

Треугольник DEF входит в треугольник ABC (или ABC содержит в себе DEF), если каждая из вершин, D, E, F, находится внутри треугольника ABC. Известен простой способ определения, попадает ли некоторая точка Y во внутреннюю область треугольника ABC: Если сумма площадей треугольников AYB, BYC и AYC равна площади треугольника ABC, то очевидно, что ABC содержит в себе Y. Чтобы реализовать эту идею решения, внесите следующие дополнения в тексты модулей.

1. **Triangle.h:** добавьте в класс `Triangle` объявления методов:

```
double Area(Point, Point, Point) const; // Площадь треугольника, образованного
                                         // заданными точками
bool Contains(Point) const; // Определяет, находится ли точка внутри треугольника
```

а также объявление дружественной функции:

```
friend bool TriaInTria(Triangle, Triangle); // Определяет,
                                         // входит ли один треугольник во второй
```

2. **Triangle.cpp:** добавьте реализацию методов:

```
double Triangle::Area(Point p1, Point p2, Point p3) const {
    double a1 = p1.DistanceTo(p2);
    double b1 = p2.DistanceTo(p3);
    double c1 = p1.DistanceTo(p3);
    double p = (a1 + b1 + c1) / 2;
```

продолжение ➤

```

    return sqrt(p * (p - a1) * (p - b1) * (p - c1));
}
bool Triangle::Contains(Point pt) const {
    const double calc_error = 0.001;
    double area1 = Area(pt, v1, v2);
    double area2 = Area(pt, v2, v3);
    double area3 = Area(pt, v3, v1);
    if (fabs(area1 + area2 + area3 - Area()) < calc_error) return true;
    else return false;
}

```

а также добавьте в конец файла реализацию внешней дружественной функции:

```

bool TriInTria(Triangle trial, Triangle tria2) {
    Point v1 = trial.Get_v1();
    Point v2 = trial.Get_v2();
    Point v3 = trial.Get_v3();
    return (tria2.Contains(v1) && tria2.Contains(v2) && tria2.Contains(v3));
}

```

3. Main.cpp: замените заглушку функции IsIncluded следующим:

```

// ----- Определение отношения включения
void IsIncluded(Triangle* p_tria[], int k) {
    cout << "===== Отношение включения =====" << endl;
    cout << "Введите номер 1-го треугольника (от 1 до " << k << "): ";
    int i1 = GetNumber(1, k) - 1;
    cout << "Введите номер 2-го треугольника (от 1 до " << k << "): ";
    int i2 = GetNumber(1, k) - 1;
    if (TriInTria(*p_tria[i1], *p_tria[i2]))
        cout << p_tria[i1]->GetName() << " - входит в - "
        << p_tria[i2]->GetName() << endl;
    else
        cout << p_tria[i1]->GetName() << " - не входит в - "
        << p_tria[i2]->GetName() << endl;
    ExitBack();
}

```

Модификация проекта завершена. Заметим, что погрешность сравнения площадей в методе Contains задается константой calc_error.

Откомпилируйте и запустите программу. Выберите пункт меню 4. Следуя указаниям программы, введите номера сравниваемых треугольников, например, 1 и 2. Вы должны получить такой результат:

```

4
===== Отношение включения =====
Введите номер 1-го треугольника (от 1 до 4): 1
Введите номер 2-го треугольника (от 1 до 4): 2
Copy constructor for: triaB
Copy constructor for: triaA
Destructor for: triaA(копия)
Destructor for: triaB(копия)

```

Треугольник 1 - входит в - Треугольник 2
Нажмите Enter.

Обратите внимание на отладочную печать: конструкторы копирования вызываются при передаче аргументов в функцию `TriaInTria`, а перед возвратом из этой функции вызываются соответствующие деструкторы.

Проведите следующий эксперимент: удалите с помощью скобок комментария конструктор копирования в файлах `Triangle.h` и `Triangle.cpp`, откомпилируйте проект и повторите тестирование четвертого пункта меню. Убедившись в неадекватном поведении программы, верните проект в нормальное состояние. Протестируйте остальные пункты меню. По завершении тестирования уберите с помощью символов комментария `//` всю отладочную печать. Повторите тестирование без отладочной печати. Решение задачи 10.2 завершено.

Итоги

1. Использование классов лежит в основе объектно-ориентированной декомпозиции программных систем, которая является более эффективным средством борьбы со сложностью систем, чем функциональная декомпозиция.
2. В результате декомпозиции система разделяется на компоненты (модули, функции, классы). Чтобы обеспечить высокое качество проекта, его способность к модификациям, удобство сопровождения, необходимо учитывать сцепление внутри компонента (оно должно быть сильным) и связанность между компонентами (она должна быть слабой), а также правильно распределять обязанности между компонентом-клиентом и компонентом-сервером.
3. Класс — это определяемый пользователем тип, лежащий в основе ООП. Класс содержит ряд полей (переменных), а также методов (функций), имеющих доступ к этим полям.
4. Доступ к отдельным частям класса регулируется с помощью ключевых слов: `public` (открытая часть), `private` (закрытая часть) и `protected` (защищенная часть). Последний вид доступа имеет значение только при наследовании классов. Методы, расположенные в открытой части, формируют *интерфейс* класса и могут вызываться через соответствующий объект.
5. Если в классе имеются поля типа указателей и осуществляется динамическое выделение памяти, то необходимо позаботиться о создании конструктора копирования и перегрузке операции присваивания.
6. Для создания полного, минимального и интуитивно понятного интерфейса класса широко применяется перегрузка методов и операций.

Задания

Вариант 1

Описать класс, реализующий стек. Написать программу, использующую этот класс для моделирования Т-образного сортировочного узла на железной дороге. Программа должна разделять на два направления состав, состоящий из вагонов двух

типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность формирования состава из файла и с клавиатуры.

Вариант 2

Описать класс, реализующий бинарное дерево, обладающее возможностью добавления новых элементов, удаления существующих, поиска элемента по ключу, а также последовательного доступа ко всем элементам.

Написать программу, использующую этот класс для представления англо-русского словаря. Программа должна содержать меню, позволяющее выполнить проверку всех методов класса. Предусмотреть возможность создания словаря из файла и с клавиатуры.

Вариант 3

Построить систему классов для описания плоских геометрических фигур: круг, квадрат, прямоугольник. Предусмотреть методы для создания объектов, перемещения на плоскости, изменения размеров и вращения на заданный угол.

Написать программу, демонстрирующую работу с этими классами. Программа должна содержать меню, позволяющее осуществить проверку всех методов классов.

Вариант 4

Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 5

Составить описание класса для представления комплексных чисел. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 6

Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 7

Составить описание класса прямоугольников со сторонами, параллельными осям координат. Предусмотреть возможность перемещения прямоугольников на пло-

скости, изменение размеров, построение наименьшего прямоугольника, содержащего два заданных прямоугольника, и прямоугольника, являющегося общей частью (пересечением) двух прямоугольников.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 8

Составить описание класса для определения одномерных массивов целых чисел (векторов). Предусмотреть возможность обращения к отдельному элементу массива с контролем выхода за пределы массива, возможность задания произвольных границ индексов при создании объекта и выполнения операций поэлементного сложения и вычитания массивов с одинаковыми границами индексов, умножения и деления всех элементов массива на скаляр, вывода на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 9

Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массива, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывода на экран элемента массива по заданному индексу и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 10

Составить описание класса многочленов от одной переменной, задаваемых степенью многочлена и массивом коэффициентов. Предусмотреть методы для вычисления значения многочлена для заданного аргумента, операции сложения, вычитания и умножения многочленов с получением нового объекта-многочлена, вывод на экран описания многочлена.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 11

Составить описание класса одномерных массивов строк, каждая строка задается длиной и указателем на выделенную для нее память. Предусмотреть возможность обращения к отдельным строкам массива по индексам, контроль выхода за пределы массивов, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, вывода на экран элемента массива и всего массива.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 12

Составить описание класса, обеспечивающего представление матрицы произвольного размера с возможностью изменения числа строк и столбцов, вывод на экран подматрицы любого размера и всей матрицы.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 13

Написать класс для эффективной работы со строками, позволяющий форматировать и сравнивать строки, хранить в строках числовые значения и извлекать их. Для этого необходимо реализовать:

- ☐ перегруженные операции присваивания и конкатенации;
- ☐ операции сравнения и приведения типов;
- ☐ преобразование в число любого типа;
- ☐ форматный вывод строки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 14

Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 15

Описать класс «записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 16

Описать класс «студенческая группа». Предусмотреть возможность работы с переменным числом студентов, поиска студента по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 17

Описать класс, реализующий тип данных «вещественная матрица» и работу с ними. Класс должен реализовывать следующие операции над матрицами:

- ☐ сложение, вычитание, умножение, деление (+, -, *, /) (умножение и деление как на другую матрицу, так и на число);
- ☐ комбинированные операции присваивания (+=, -=, *=, /=);
- ☐ операции сравнения на равенство (неравенство);
- ☐ операции вычисления обратной и транспонированной матрицы, операцию возведения в степень;
- ☐ методы вычисления детерминанта и нормы;
- ☐ методы, реализующие проверку типа матрицы (квадратная, диагональная, нулевая, единичная, симметрическая, верхняя треугольная, нижняя треугольная);
- ☐ операции ввода-вывода в стандартные потоки.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 18

Описать класс «множество», позволяющий выполнять основные операции — добавление и удаление элемента, пересечение, объединение и разность множеств.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Вариант 19

Описать класс, реализующий стек. Написать программу, использующую этот класс для отыскания прохода по лабиринту.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат открыт, то вход в него возможен со стороны, но не с угла. Каждый квадрат определяется его координатами в матрице. После отыскания прохода программа печатает найденный путь в виде координат квадратов.

Вариант 20

Описать класс «предметный указатель». Каждый компонент указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, от одного до десяти. Предусмотреть возможность формирования указателя с клавиатуры и из файла, вывода указателя, вывода номеров страниц для заданного слова, удаления элемента из указателя.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Семинар 11. Наследование

Теоретический материал: с. 200–210.

На этом семинаре мы не ограничимся наследованием, а рассмотрим более широкий круг вопросов:

- ❑ наследование классов и полиморфизм;
- ❑ отношения между классами и диаграммы классов на языке UML;
- ❑ введение в использование паттернов проектирования; технология проектирования программ с учетом будущих изменений.

Наследование классов

Классы в объектно-ориентированных программах используются для моделирования концепций реального и программного мира. Концепции или сущности предметной области находятся в различных взаимоотношениях. Одно из таких взаимоотношений — *отношение наследования* (именуемое также отношением *родитель–потомок* или отношением *обобщение–специализация*).

Например, если в условии задачи 10.2 рассматривать не только треугольники, но и четырехугольники, класс `Tetragon` имел бы общие черты с классом `Triangle`, поскольку и треугольники, и четырехугольники являются частным (специальным, конкретным) случаем более общего понятия «многоугольник». Поэтому было бы логичным создать класс `Polygon`, содержащий элементы, общие для классов `Triangle` и `Tetragon`, а последние два класса объявить «наследниками» базового (родительского) класса `Polygon`. Язык C++ позволяет легко это сделать:

```
class Polygon {  
    // . . .  
};  
class Triangle : public Polygon {  
public: Show();  
};  
class Tetragon : public Polygon {  
public: Show();  
};
```

В этом примере производные классы `Triangle` и `Tetragon` наследуют все элементы базового класса `Polygon`, но каждый из них имеет свой собственный метод `Show`. Иногда

отношение наследования называют отношением «*is a*», что можно перевести как «представляет собой». В данном примере *Triangle is a Polygon*, в такой же мере и *Tetragon is a Polygon*.

Общий синтаксис создания производного класса при *простом наследовании*:

```
class имя : ключ_доступа имя_базового_класса {  
    // тело класса  
};
```

В случае *множественного наследования* после двоеточия перечисляются через запятую все базовые классы со своими модификаторами доступа.

Производный класс, в свою очередь, сам может служить базовым классом. Такой набор связанных классов называется *иерархией классов*. Иерархия чаще всего является деревом, но может иметь и более общую структуру графа.

В примерах предыдущего семинара доступ к элементам класса регулировался с помощью двух модификаторов: *private* — закрытая часть класса, *public* — открытая часть класса. Для базовых классов возможно использование еще одного модификатора — *protected*, который определяет *защищенную* часть класса. Ее элементы являются доступными для любого производного класса, но в то же время они недоступны вне классов данной иерархии.

Кроме этого, доступность в производном классе регулируется *ключом доступа к базовому классу*, указываемому в объявлении производного класса. Этот ключ определяет *вид наследования*: открытое (*public*), защищенное (*protected*) или закрытое (*private*).

Открытое наследование сохраняет статус доступа всех элементов базового класса, *защищенное* — понижает статус доступа *public* элементов базового класса до *protected*, и наконец, *закрытое* — понижает статусы доступа *public* и *protected* элементов базового класса до *private*. Заметим, что в C++ отношение между классами «*is a*» имеет место только при открытом наследовании.

Замещение функций базового класса

Иногда в производном классе требуется несколько иная реализация метода, унаследованного из базового класса. Язык позволяет это сделать. *Замещение (перезагрузка)* метода производится путем объявления в производном классе метода с таким же именем. Если понадобится все-таки вызвать из потомка метод предка, используется операция доступа к области видимости `::`, например:

```
#include <iostream>  
using namespace std;  
class Base {  
public:  
    void Display() { cout << "Hello, world! "; }  
};  
class Derived : public Base {  
public:  
    void Display() {
```

продолжение ➤

```

        Base::Display();           // Вызов метода базового класса
        cout << "How are you? ";
    }
};
class SubDerived : public Derived {
public:
    void Display() {
        Derived::Display();       // Вызов метода базового класса
        cout << "Bye!" << endl;
    }
};
int main() {
    SubDerived sd;
    sd.Display();                 // Результат: Hello, world!  How are you?  Bye!
}

```

Конструкторы и деструкторы в производном классе

Конструкторы и деструкторы из базового класса не наследуются, поэтому при создании производного класса встает вопрос, нужны ли эти методы и как они должны быть реализованы. Решая эти вопросы, учитывайте следующее.

- ❑ Если в базовом классе вообще нет конструктора или есть конструктор по умолчанию, производному классу конструктор нужен, *только* если требуется инициализировать поля, введенные в этом классе.
- ❑ Если вы не определили ни одного конструктора, компилятор самостоятельно создаст конструктор по умолчанию, из которого будет вызван конструктор по умолчанию базового класса.
- ❑ Если в базовом классе есть конструктор с аргументами, то производный класс должен содержать конструктор со списком аргументов, включающим значения для передачи конструктору базового класса; конструктор базового класса вызывается в списке инициализации.

Необходимость **в деструкторе** для производного класса определяется тем, нужно ли освобождать какие-либо ресурсы, выделенные в конструкторе. Если такой необходимости нет, можно доверить компилятору создать деструктор по умолчанию. В нем обеспечивается вызов деструктора базового класса.

На этапе выполнения программы при создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, затем конструкторы объектов-элементов класса и в последнюю очередь — конструктор класса. При уничтожении объекта (например, когда покидается область его видимости) деструкторы вызываются в порядке, обратном вызову конструкторов.

Устранение неоднозначности при множественном наследовании

Предположим, что от базового класса А, имеющего некоторый элемент х, наследуются два класса, В и С, а класс D является производным от этих двух классов (множественное наследование). Если попытаться обратиться к элементу х из методов класса

D, компилятор воспримет выражение `D::x` как неоднозначное¹ и прекратит работу. Для решения этой проблемы в C++ предусмотрен механизм, благодаря которому в класс D будет включена только одна копия класса A. Достигается это добавлением спецификатора `virtual` перед модификаторами доступа к A в объявлениях классов B и C:

```
class A {
public:    A(int _x = 0) { x = _x; }
protected: int x;
};
class B : virtual public A {                // Виртуальное наследование
public:    void AddB(int y) { x += y; }
};
class C : virtual public A {                // Виртуальное наследование
public:    void AddC(int y) { x += y; }
};
class D : public B, public C {              // Здесь будет только одна копия полей класса A
public:    void ShowX() { cout << "x = " << x << endl; }
};
int main() {
    D d;
    d.ShowX();                            // Вывод: x = 0
    d.AddB(10); d.ShowX();                 // Вывод: x = 10
    d.AddC(5);  d.ShowX();                 // Вывод: x = 15
}
```

Доступ к объектам иерархии

Эффективнее всего работать с объектами одной иерархии через указатель на базовый класс. При открытом (`public`) наследовании такому указателю можно присваивать адрес объекта как базового класса, так и любого производного класса. Такая возможность представляется вполне логичной: в качестве представителя более общего класса используется объект специализированного класса.

Однако если не принять специальных мер, то при работе с таким указателем независимо от того, какой адрес ему присвоен, оказываются доступными *только элементы базового класса*. Например, рассмотрим следующий код:

```
class Base {
public: void Modify();
};
class Derived : public Base {
public: void Modify();
};
int main() {
    int mode;
    Base* pB;
    cin >> mode;
    if (mode == 1) pB = new Base;
    else          pB = new Derived;
    pB->Modify();                // на что указывает pB?
}
```

¹ Поскольку неясно, какой элемент класса A имеется в виду: унаследованный через класс B или унаследованный через класс C.

Так как компилятор не может предсказать, какой выбор будет сделан на этапе выполнения, он выбирает метод по типу указателя pB, то есть Base::Modify. Такая стратегия называется *ранним (статическим) связыванием*. Поэтому, если в производном классе замещен некоторый метод базового класса (например, Derived::Modify), он оказывается недоступным (листинг 11.1).

Листинг 11.1. Раннее связывание

```
#include <iostream>
using namespace std;
class Base {
public:
    Base(int _x = 10) { x = _x; }
    void ShowX() { cout << "x = " << x << "; "; }
    void ModifyX() { x *= 2; }
protected:
    int x;
};
class Derived : public Base {
public:
    void ModifyX() { x /= 2; }
};
int main() {
    Base b;    Derived d;
    b.ShowX(); d.ShowX();           // Вывод: x = 10; x = 10;
    Base* pB;
    pB = &b;   pB->ModifyX();   pB->ShowX();   // Вывод: x = 20;
    pB = &d;   pB->ModifyX();   pB->ShowX();   // Вывод: x = 20;
}
```

Как видите, второй вызов метода ModifyX происходит также из базового класса.

Виртуальные методы

Проблема доступа к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования *виртуальных методов*. Чтобы сделать некоторый метод виртуальным, надо в базовом классе предварить его заголовок спецификатором virtual. После этого он будет восприниматься как виртуальный во всех производных классах иерархии.

По отношению к виртуальным методам компилятор применяет стратегию *позднего (динамического) связывания*. Это означает, что на этапе компиляции он не определяет, какой из методов должен быть вызван, а передает ответственность программе, принимающей решение на этапе выполнения, когда уже точно известно, каков тип объекта, адресуемого через указатель. Все сказанное относится также к вызову методов *по ссылке* на базовый класс.

Для реализации динамического связывания компилятор создает *таблицу виртуальных методов (vtbl)*, а к каждому объекту добавляет скрытое поле ссылки (*vptr*) на таблицу vtbl. Это несколько снижает эффективность программы, поэтому

рекомендуется делать виртуальными только те методы, которые переопределяются в производных классах.

Чтобы увидеть динамическое связывание в действии, добавьте в листинг 11.1 спецификатор `virtual` перед заголовком метода `ModifyX` в базовом классе:

```
virtual void ModifyX() { x *= 2; }
```

Теперь второй вызов метода `ModifyX` происходит из производного класса, и программа должна вывести

```
x = 10; x = 10; x = 20; x = 5;
```

Виртуальный механизм работает только при использовании указателей и ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным*.

Если базовый класс содержит хотя бы один виртуальный метод, рекомендуется всегда снабжать этот класс *виртуальным деструктором*, даже если он ничего не делает. Наличие виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс, так как иначе деструктор производного класса вызван не будет¹.

Абстрактные классы. Чисто виртуальные методы

Иногда, разрабатывая иерархию классов, можно получить базовый класс, для которого создание объекта как бы теряет смысл. Например, иерархия классов геометрических фигур может произрастать из базового класса `Shape`, а в качестве производных будут выступать классы `Polygon`, `Ellipse` и т.д. Одним из методов базового класса будет, видимо, функция `Show` — показать объект. Но как можно показывать объект, не имеющий конкретной формы?

Классы, для которых нет смысла создавать объекты, объявляют как *абстрактные*. Абстрактный класс — это класс, содержащий хотя бы один *чисто виртуальный метод* (метод, объявленный в классе, но не имеющий конкретной реализации). Синтаксис объявления такого метода дополняется конструкцией `= 0`, например:

```
virtual void Show() = 0;
```

Компилятор не допустит создания объекта для абстрактного класса, если вы по забывчивости попытаетесь это сделать. Если в производном классе хотя бы одна чисто виртуальная функция базового класса останется без конкретной реализации, то производный класс также будет абстрактным классом, для которого запрещено создавать объекты.

Отношения между классами. Диаграммы классов на языке UML

Полиморфизм и наследование делают язык C++ чрезвычайно мощным инструментом для реализации объектно-ориентированной технологии проектирования.

¹ В небольшой учебной программе это может пройти без последствий. В нетривиальных программах возможно появление труднодиагностируемых ошибок.

Для эффективного восприятия новых технологических идей полезно знать хотя бы основы ставшего уже стандартом *унифицированного языка моделирования UML* (Unified Modeling Language).

Язык UML является визуальным средством представления *моделей программ*, то есть их графического представления в виде различных диаграмм, отражающих связи между объектами в программном коде.

Одной из основных диаграмм языка UML является *диаграмма классов*. Она описывает классы и отношения, существующие между ними. Такая диаграмма чрезвычайно удобна для сопоставления различных вариантов проектных решений. Класс изображается на диаграмме классов UML в виде прямоугольника, состоящего из трех частей. Имя класса указывается в верхней части. В средней части приводится список *атрибутов* с указанием типов (атрибут класса в UML соответствует термину *поле класса* в C++).

В нижней части записывается список *операций* (методов класса), возможно, с указанием списка типов аргументов и типа возвращаемого значения. Имя *абстрактного класса*, так же как и имена *абстрактных операций* (чисто виртуальных методов), выделяются курсивом. Перед именем поля или метода может указываться спецификатор доступа с помощью одного из трех символов: + для public, - для private, # для protected. Для статических элементов класса после спецификатора доступа записывается символ \$.

Во второй и третьей частях могут указываться не все элементы класса, а только те, которые представляют интерес на данном уровне абстракции. Если обе эти части пусты, они могут быть опущены. Например, варианты изображения класса Triangle на различных этапах анализа решения задачи 10.2 показаны на рис. 11.1.

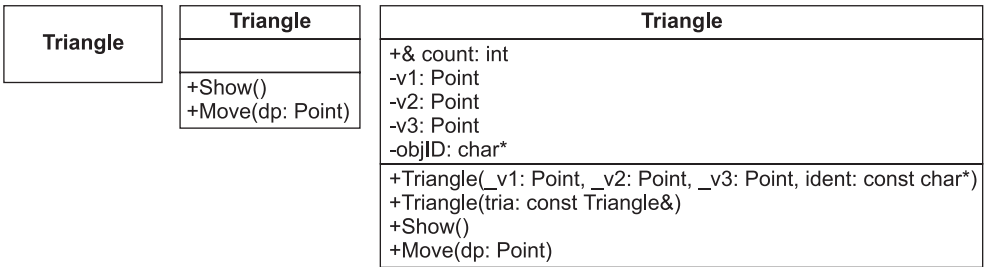


Рис. 11.1. Варианты изображения класса Triangle на диаграмме классов UML

Большинство объектно-ориентированных языков поддерживают следующие виды отношений между классами: ассоциация, наследование, агрегация и зависимость. Рассмотрим, что они означают и как изображаются на диаграмме классов.

Ассоциация

Если два класса концептуально взаимодействуют друг с другом, такое взаимодействие называется *ассоциацией*. Например, желая смоделировать торговую точку, мы вводим две абстракции: товары (класс Product) и продажи (класс Sale). Объект класса Sale — это некоторая сделка, в которой продано от 1 до n объектов класса

Product. На рис. 11.2 ассоциация между этими двумя классами показана соединяющей их линией. Над линией рядом с обозначением класса может быть указана так называемая *кратность* (*multiplicity*), указывающая, сколько объектов данного класса может быть связано с одним объектом другого класса. Символ звездочки * обозначает «произвольное количество».



Рис. 11.2. Отношения ассоциации

Ассоциация представляет наиболее абстрактную семантическую связь между двумя классами, выявляемую на ранней стадии анализа. В дальнейшем она, как правило, конкретизируется, переходя в одно из рассматриваемых далее отношений.

Наследование

Отношение *обобщения* (наследования, «*is a*») между классами показывает, что *подкласс* (производный класс) разделяет атрибуты и операции, определенные в одном или нескольких *суперклассах* (базовых классах). На диаграмме классов отношение наследования показывают линией со стрелкой в виде незакрашенного треугольника, которая указывает на базовый класс. Допускается объединять несколько стрелок в одну (см. рис. 11.3), с тем чтобы разгрузить диаграмму.

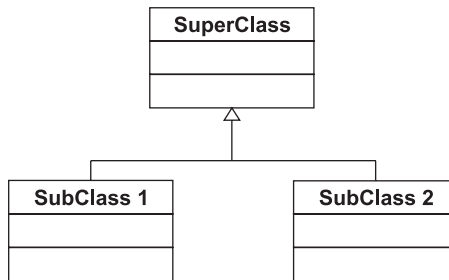


Рис. 11.3. Отношения наследования

Агрегация

Отношение *агрегации* показывает, что один класс содержит в качестве составной части объекты другого класса. Иными словами, это отношение *целое-часть*, или отношение «*has a*», между двумя классами. На диаграмме такая связь обозначается линией со стрелкой в виде незакрашенного ромба, которая указывает на целое. На рис. 11.4 показан пример так называемой *нестрогой* агрегации (или просто агрегации). Действительно, конкретный объект класса СпортЗал может содержать не все компоненты (спортивные снаряды), присутствующие на схеме.

Как распознать агрегацию в программном коде? Для реализации нестрогой агрегации часть включается в целое *по ссылке*: на языке C++ это обычно указатель на соответствующий класс. Если этот указатель равен нулю, то компонент отсутствует. В зависимости от решаемой задачи такой компонент может появляться и исчезать динамически в течение жизни объекта целое.



Рис. 11.4. Отношения агрегации

Строгая агрегация имеет специальное название — *композиция*. Она означает, что компонент не может исчезнуть, пока объект целое существует. Пример такого отношения мы уже встречали при решении задачи 10.2, в которой класс `Triangle` содержал в себе три объекта класса `Point`. На диаграмме отношение композиции обозначается линией со стрелкой в виде закрашенного ромба (рис. 11.5).

Проще всего композицию реализовать включением объектов-компонентов *по значению*. В то же время возможна реализация и включением по ссылке, но тогда времена жизни компонентов и объекта целое должны совпадать.

Зависимость

Отношение *зависимости (использования)* показывает, что один класс (`Client`) пользуется услугами другого класса (`Supplier`), например:

- метод класса `Client` использует значения некоторых полей класса `Supplier`;
- метод класса `Client` вызывает некоторый метод класса `Supplier`;
- метод класса `Client` имеет сигнатуру, в которой упоминается класс `Supplier`.

На диаграмме такое отношение изображается пунктирной линией со стрелкой, указывающей на класс `Supplier`. Например, отношения зависимости существует между классами `Triangle` и `Point` из задачи 10.2 (рис. 11.6).



Рис. 11.5. Отношение композиции

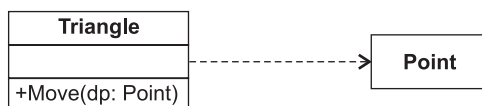


Рис. 11.6. Отношение зависимости (использования)

Паттерны проектирования

Одним из важнейших достижений в области ООП является методология паттернов проектирования, иногда называемых шаблонами проектирования¹. Впервые она была представлена в книге [6].

¹ Слово «шаблон» в отношении термина «pattern» не вполне удачно, поскольку вызывает путаницу с термином «template». Поэтому во многих книгах издательства «Питер» используется термин «паттерн». — *Примеч. ред.*

Паттерн — это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерны выявляются по мере накопления опыта разработок, когда программист использует одну и ту же схему организации и взаимодействия объектов в разных контекстах.

Поначалу паттерны можно было рассматривать как особенно изящный и хорошо продуманный способ решения определенного класса задач. Однако практика показала, что применение паттернов наряду с объектной технологией позволяет вывести процесс проектирования из области интуиции и представить систему на более высоком уровне абстракции — на уровне схем взаимодействия объектов. Любая абстракция исключает частные, второстепенные детали, и обычно при этом удается выделить в проблеме ее переменные и постоянные составляющие.

Основные трудности при разработке элегантной и удобной в сопровождении архитектуры вызывает идентификация так называемого «вектора изменений». Выявление важнейшего фактора перемен в проектируемой системе дает опорную точку для построения дальнейшей архитектуры.

Итак, паттерны предназначены прежде всего для *инкапсуляции изменений*. Если рассматривать их с этой точки зрения, то некоторые паттерны нам уже встречались. Например, наследование тоже может рассматриваться как паттерн, пусть даже реализованный на уровне компилятора. Оно выражает различия в поведении (переменная составляющая) объектов, обладающих одинаковым интерфейсом (постоянная составляющая). Впрочем, обычно возможности, напрямую поддерживаемые языком программирования, не принято относить к паттернам.

Один из фундаментальных принципов, провозглашенный в книге [6], гласит: «Отдавайте предпочтение композиции объектов перед наследованием классов». И действительно, иногда композиция радикально упрощает архитектуру, в которой кажется уместным наследование, а полученная архитектура становится более гибкой.

Краткие сведения о паттернах в данном разделе приведены по книге [6]. Разумеется, для более глубокого изучения этой темы нужно обратиться к специальной литературе, например, [5], [6] и [34].

Все паттерны в соответствии с их назначением разделены на три группы: порождающие, структурные и паттерны поведения. *Порождающие паттерны* описывают способы создания объектов, *структурные* — схемы организации классов и объектов, а *паттерны поведения* определяют типичные схемы взаимодействия классов и объектов. В каждой группе можно, в свою очередь, выделить два уровня, определяющих, применяется паттерн к классам или объектам. В табл. 11.1 приведена классификация паттернов.

Описание каждого паттерна в [6] выполнено по одной и той же схеме, включающей его назначение, область применения, структурную схему (в форме диаграммы классов UML), описание входящих в него объектов и их взаимодействий, а также пример реализации. Такой уровень документирования делает возможным использование паттерна в различных конкретных случаях, возникающих при проектировании программных систем.

Таблица 11.1. Классификация паттернов проектирования

Назначение/ Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

В качестве примера мы приведем здесь описание только паттерна Стратегия.

Паттерн Стратегия (Strategy)

Этот паттерн инкапсулирует семейство алгоритмов, делая их взаимозаменяемыми. Применять его целесообразно в следующих случаях:

- ❑ имеются родственные классы, различающиеся только поведением. Стратегия позволяет гибко конфигурировать класс, задавая одно из возможных поведений;
- ❑ требуется иметь несколько разных вариантов алгоритма. Варианты алгоритмов могут быть реализованы в виде иерархии классов, что позволяет вычленить общую для всех классов функциональность. Инкапсулированные алгоритмы можно затем применять в разных контекстах;
- ❑ в классе определено много вариантов поведения, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Диаграмма классов паттерна приведена на рис. 11.7.

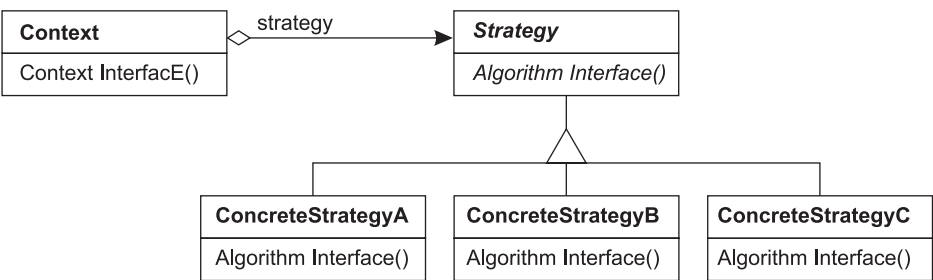


Рис. 11.7. Структура паттерна Стратегия

Паттерн состоит из следующих классов.

- ❑ **Strategy** (стратегия) — объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy.

- ❑ **ConcreteStrategy** (конкретная стратегия) — наследник класса Strategy. Реализует алгоритм, использующий интерфейс, объявленный в классе Strategy.
- ❑ **Context** (контекст) — конфигурируется объектом класса ConcreteStrategy. Хранит ссылку на объект класса Strategy и может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.
- ❑ **Пример реализации.** Термин «стратегия» означает лишь то, что у проблемы имеется несколько решений. Допустим, что вы забыли, как зовут встреченного вами знакомого. Из неловкого положения можно выйти несколькими способами, как показано в листинге 11.2 (идея примера позаимствована из [38]).

Листинг 11.2. Использование паттерна Strategy

```
#include <iostream>
using namespace std;
class NameStrategy {
public: virtual void greet() = 0;
};
class SayHi : public NameStrategy {
public: void greet() { cout << "Привет! Как дела?" << endl; }
};
class Ignore : public NameStrategy {
public: void greet() { cout << "(Сделать вид, что не заметил человека)" << endl; }
};
class Admission : public NameStrategy {
public: void greet() { cout << "Простите, я забыл Ваше имя." << endl; }
};
class Context { // ----- Контекст управляет выбором стратегии:
    NameStrategy& strategy;
public:
    Context(NameStrategy& strat) : strategy(strat) {}
    void greet() { strategy.greet(); }
};
int main() {
    SayHi    sayhi;
    Ignore   ignore;
    Admission admission;
    Context c1(sayhi), c2(ignore), c3(admission);
    c1.greet(); c2.greet(); c3.greet();
}
```

Завершая краткое введение в паттерны проектирования, заметим, что хороший критерий профессионализма — выделение паттернов самостоятельно, на основании собственного опыта. При решении следующих задач мы покажем, как это делается.

Проектирование программы с учетом будущих изменений

Одним из показателей качества программной системы является ее способность к модификациям в связи с появлением новых требований заказчика. На предыдущем

семинаре уже отмечалось, что большое значение имеет правильная декомпозиция системы на компоненты (модули, классы, функции).

Однако тактические решения локальных проблем также влияют на то, сколь сложным окажется добавление новой функциональности к разработанному ранее проекту. Сейчас мы рассмотрим некоторые концепции поиска таких решений.

Одной из причин негибкости процедурно-ориентированных систем является частое использование в них управляющих конструкций на базе оператора `switch`. Рассмотрим типичную ситуацию: некоторая функция `SomeFunc` в зависимости от значения параметра `mode` вызывает одну из обрабатывающих процедур:

```
typedef enum { mode1, mode2, mode3 } ModeType;
void SomeFunc (ModeType mode) {
    switch (mode) {
        case mode1: DoSomething1();    break;
        case mode2: DoSomething2();    break;
        case mode3: DoSomething3();    break;
    }
}
```

Допустим, что перечень значений перечисляемого типа `ModeType` согласован с заказчиком и утвержден техническим заданием. Вся система, таким образом, разрабатывается с использованием утвержденного перечня. Однако через некоторое время заказчик обнаруживает, что необходимо предусмотреть еще одно значение параметра `mode`. Хорошо, если он еще платежеспособен и сможет заказать модификацию системы. Но для разработчиков наступят черные дни, когда придется искать по всем модулям переключатели `switch` и вносить нужные добавления. При этом потребуются полное повторное тестирование всех модулей, использующих эту функцию, так как могут появиться неожиданные побочные эффекты.

Попытаемся найти более удачное решение проблемы типа «переключатель». Одна из рекомендаций, приведенных в [6], звучит так: «Найдите то, что должно или может *измениться* в вашем дизайне, и *инкапсулируйте сущности*, подверженные изменениям!» В нашем случае переменной сущностью являются вызываемые процедуры. Можем ли мы их инкапсулировать? Да — если реализуем эти процедуры как виртуальные методы полиморфных объектов!

Идея заключается в следующем: объект типа «переключатель» (класс `Switch`) должен иметь дело с некоторой абстрактной «процедурой вообще», а это можно реализовать, создав абстрактный класс `AbstractEntity` с чисто виртуальным методом `DoSomething`. Конкретные процедуры в виде одноименных замещенных методов будут содержаться в производных от `AbstractEntity` классах `Entity1`, `Entity2`, и т. д.

Чтобы делать «осознанный» выбор, класс `Switch` должен быть осведомлен об адресах объектов `Entity1`, `Entity2`, и т. д. Это можно обеспечить, поместив список адресов в одно из его полей типа `std::vector<AbstractEntity*>`¹. Описываемая идея поясняется на диаграмме классов (рис. 11.8).

¹ Класс `std::vector` содержится в стандартной библиотеке шаблонов STL, которая изучается на семинаре 15. Тем не менее мы будем пользоваться этим удобным инструментом работы с динамическими массивами уже сейчас.

Обратите внимание, что между классами Switch и AbstractEntity имеется отношение *композиции* (стрелка с закрашенным ромбиком), так как класс Switch содержит поле типа `std::vector<AbstractEntity*>`. В то же время класс Switch *использует* класс AbstractEntity (пунктирная стрелка), поскольку один из его методов возвращает значение типа AbstractEntity*.

Как же все это работает? Клиент обращается с запросом SelectEntity к объекту Switch (то есть вызывает одноименный метод). Неважно, как реализован метод SelectEntity, существенно то, что он возвращает значение типа AbstractEntity*, содержащее адрес объекта одного из подклассов класса AbstractEntity. После этого через полученный указатель клиент вызывает конкретную процедуру DoSomething одного из объектов Entity1, Entity2, ...

Чем примечательно предлагаемое решение? Необычайной легкостью модификации. Если возникла необходимость дополнить список переключаемых сущностей еще одним k-м экземпляром, то достаточно добавить производный класс EntityK, объявить объект этого класса и добавить адрес нового объекта в список `vector<AbstractEntity*>`.

Очевидно, что предложенное на рис. 11.8 решение есть не что иное, как еще один *паттерн проектирования*, который мы назвали Switch. Пример его реализации приводится в решении задачи 11.1. Понятно, что скорее всего аналогичное решение уже неоднократно применялось программистами на практике, но ведь решение становится паттерном только после его описания на требуемом уровне абстракции!

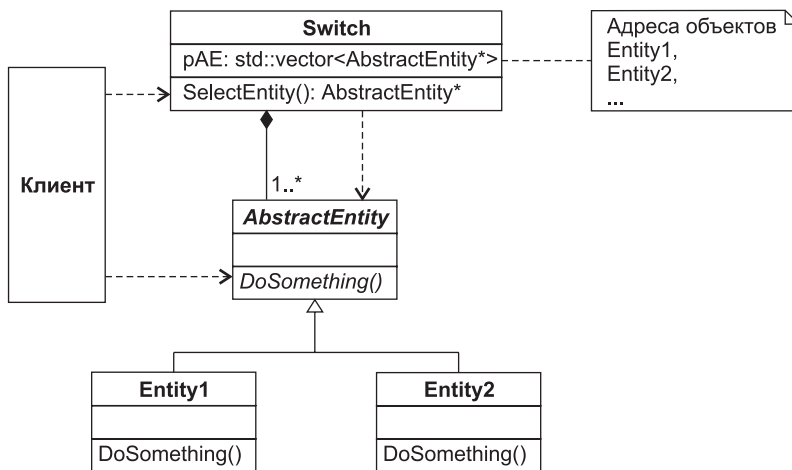


Рис. 11.8. Паттерн Switch (переключатель)

Задача 11.1. Функциональный калькулятор

Разработать программу, имитирующую работу функционального калькулятора, который позволяет выбрать с помощью меню какую-либо из известных ему функций,

затем предлагает ввести значение аргумента x , возможно, коэффициентов и после ввода выдает соответствующее значение функции.

В первой версии калькулятора «база знаний» содержит две функции:

- экспоненту $y = e^x$;
- линейную функцию $y = ax + b$.

Решение задачи начнем с выявления понятий (классов) и их взаимосвязей.

Интерфейс пользователя нашего калькулятора, как следует из его описания, должен обеспечить меню для выбора вида функции. У нас уже есть опыт создания меню для консольных приложений, в которых отсутствует оконная графика, — например, в задаче 10.2. Функция `menu` в той программе выводит список выбираемых пунктов с номерами и после ввода пользователем номера пункта возвращает это значение главной функции. Далее следует традиционный переключатель:

```
switch (Menu()) {  
    case 1: Show(...); break;  
    case 2: Move(...); break;  
    ...  
    case 5: // Конец работы  
}
```

Но вряд ли стоит повторять такое решение. Ведь мы уже знаем, что подобная реализация может резко усложнить модификацию программы. Мы также вооружены новым паттерном проектирования `Switch`, который дает красивое, легко модифицируемое решение. Попробуем применить этот паттерн на практике. Несложно увидеть, что роль класса `Switch` здесь будет играть класс `Menu`.

Что является изменяемой сущностью в нашей задаче? — Вид функции, для которой нужно выполнить вычисления (в более общем случае — вид некоторого объекта, для которого нужно выполнить некоторую операцию). Следовательно, требуется инкапсулировать эту изменяемую сущность так, чтобы класс `Menu` имел дело с некоторой абстрактной «функцией вообще» (с некоторым «объектом вообще»). Отсюда вывод: необходим абстрактный класс `Function`, обеспечивающий единый унифицированный интерфейс для всех его потомков, в данном случае — для классов `Exp` и `Line`. В результате мы приходим к диаграмме классов на рис. 11.9.

На диаграмме, естественно, показаны не все элементы классов, а только наиболее существенные для данного этапа анализа. Кроме метода `Calculate`, выполняющего роль метода `DoSomething` в паттерне `Switch`, появился метод `GetName`, извлекающий наименование функции. Этим методом будет пользоваться объект `Menu` для вывода перечня выбираемых функций.

Прежде чем привести код программы, скажем несколько слов о работе с классами `std::string` и `std::vector`. Более подробно они изучаются на семинаре 12.

Класс `string` позволяет создать строку `s` типа `string`, с которой очень удобно работать: ее можно инициализировать значением C-строки: `std::string s("C-строка");` или присвоить ей значение другой строки посредством операции присваивания `=`. Память, выделенная для объекта `s`, автоматически освобождается, как только

программа покидает область его видимости. Для использования класса необходимо подключить заголовочный файл <string>.

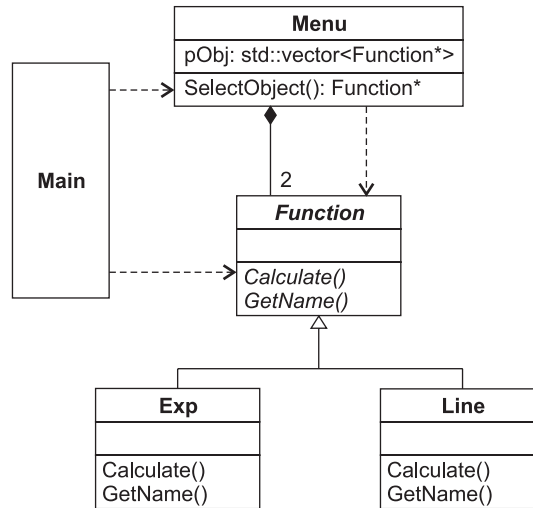


Рис. 11.9. Диаграмма классов программы «Функциональный калькулятор»

Контейнер `vector` является шаблонным классом и позволяет создавать динамические массивы (термины «*вектор*» и «*динамический массив*» мы будем использовать как синонимы). Для использования класса необходимо подключить заголовочный файл <vector>. В классе есть конструктор по умолчанию, создающий вектор нулевой длины, а также конструктор с инициализацией создаваемого вектора обычным одномерным C-массивом. Можно добавить в конец имеющегося вектора новый элемент с помощью метода `push_back`. Доступ к любому элементу вектора можно выполнять по индексу, как и в обычном массиве. Текущее количество элементов в векторе можно определить методом `size`.

Поскольку `vector` — шаблонный класс, он позволяет создавать конкретные экземпляры классов для любого типа элементов, задаваемого в качестве аргумента в угловых скобках после имени класса. Поясним использование класса на примере, в котором показано использование векторов `v1`, `v2`, `v3`, предназначенных для хранения элементов типа `char`, `int` и `double` соответственно:

```

#include <iostream>
#include <vector>
using namespace std;
int main() {
    int a[5] = { 5, 4, 3, 2, 1 };
    double b[] = { 1.1, 2.2, 3.3, 4.4 };
    vector<char> v1;
    v1.push_back('A'); v1.push_back('B'); v1.push_back('C');
    for (int i = 0; i < v1.size(); ++i) cout << v1[i] << ' '; cout << endl;
    vector<int> v2(a, a + 5);

```

продолжение ➞

```

    for (int i = 0; i < v2.size(); ++i) cout << v2[i] << ' '; cout << endl;
    vector<double> v3(b, b + sizeof(b) / sizeof(double));
    for (int i = 0; i < v3.size(); ++i) cout << v3[i] << ' '; cout << endl;
}

```

Эта программа должна вывести на экран:

```

A B C
5 4 3 2 1
2.2 3.3 4.4

```

Обратите внимание на два способа инициализации вектора одномерным массивом. Первый (для вектора `v2`) используется, когда длина массива задана константой. Второй (для вектора `v3`) оказывается единственным возможным, когда длина массива определяется на стадии компиляции. В листинге 11.3 приведена реализация программы «Функциональный калькулятор».

Листинг 11.3. Функциональный калькулятор

```

////////////////////////////////////////////////// Function.h
#ifndef FUNCTION_H
#define FUNCTION_H
#include <string>
class Function {
public:
    virtual ~Function() {}
    virtual const std::string& GetName() const = 0;
    virtual void Calculate() = 0;
protected:
    double x; // аргумент
};
#endif /* FUNCTION_H */
////////////////////////////////////////////////// Exp.h
#include "Function.h"
class Exp : public Function { // ----- Класс для представления функции  $y = e^x$ 
public:
    Exp() : name("e ^ x") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name; // математическое обозначение функции
};
extern Exp f_exp;
////////////////////////////////////////////////// Exp.cpp
#include <iostream>
#include <math.h>
#include "Exp.h"
using namespace std;
void Exp::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter x = ";
}

```

```

    cin >> x; cin.get();
    cout << "y = " << exp(x) << endl; cin.get();
}
Exp f_exp;    // Глобальный объект
// //////////////////////////////////////// Line.h
#include "Function.h"
class Line : public Function { // --- Класс для представления функции  $y = a * x + b$ 
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;    // математическое обозначение функции
    double a, b;
};
extern Line f_line;
// //////////////////////////////////////// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;
void Line::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = ";    cin >> a;
    cout << "Enter b = ";    cin >> b;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    cout << "y = " << (a * x + b) << endl;
    cin.get();
}
Line f_line;    // Глобальный объект
// //////////////////////////////////////// Menu.h
#include <vector>
#include "Function.h"
class Menu {
public:
    Menu(std::vector<Function*>);
    Function* SelectObject() const;
private:
    int SelectItem(int) const;
    std::vector<Function*> pObj;
};
// //////////////////////////////////////// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Function*> _pObj) : pObj(_pObj) {
    pObj.push_back(0);    // для выбора пункта 'Exit'
}
Function* Menu::SelectObject() const {
    int nItem = pObj.size();
    cout << "=====\\n";

```

продолжение ➤

Листинг 11.3 (продолжение)

```

    cout << "Select one of the following function:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i+1 << ". "; // номер пункта меню на 1 больше индекса массива pObj
        if (pObj[i]) cout << pObj[i]->GetName() << endl;
        else cout << "Exit" << endl;
    }
    int item = SelectItem(nItem);
    return pObj[item - 1];
}
int Menu::SelectItem(int nItem) const {
    cout << "-----\n";
    int item;
    while ( true ) {
        cin >> item;
        if ((item > 0) && (item <= nItem) && (cin.peek() == '\n')) {
            cin.get(); break;
        }
        else {
            cout << "Error (must be number from 1 to " << nItem << "):" << endl;
            cin.clear();
            while ( cin.get() != '\n' ) {};
        }
    }
    return item;
}
// ////////////////////////////////////// Main.cpp
#include <iostream>
#include "Function.h"
#include "Exp.h"
#include "Line.h"
#include "Menu.h"
using namespace std;
Function* pObjjs[] = { &f_exp, &f_line };
vector<Function*> funcList(pObjjs, pObjjs + sizeof(pObjjs) / sizeof(Function*));
int main() {
    Menu menu(funcList);
    while ( Function* pObj = menu.SelectObject() )
        pObj->Calculate();
    cout << "Bye!\n";
}

```

Несколько пояснений по тексту программы.

В абстрактном классе `Function` конструктор отсутствует. Класс содержит *виртуальный* деструктор (см. рекомендацию, приведенную в разделе «Виртуальные методы»). Существует рекомендация размещать в абстрактном классе *только методы*, а все поля объявлять в производных классах (то есть делать класс чисто интерфейсным). Этим гарантируется, что любые изменения и дополнения не затронут абстрактный класс. В данном случае мы все-таки описали поле `x` для значения

аргумента, поскольку рассматриваем только функции с одним аргументом и при этом полагаем, что аргумент всегда имеет тип `double`. А вот значения коэффициентов содержатся в полях производных классов.

Обратите внимание на сигнатуру метода `GetName` в классе `Function` (*оператор 1*). Метод объявлен константным, так как он не должен изменять состояние полей класса. Кроме того, чтобы исключить вызов конструктора копирования, мы хотим вернуть значение по ссылке: `std::string&`. Однако сигнатуру `virtual std::string& GetName const` не пропустит компилятор, так как возврат по ссылке `string&` противоречит модификатору `const`. Выход из этого противоречия — возврат по константной ссылке: `const std::string&`.

Конкретные методы `Calculate` в классах `Exp` и `Line` обеспечивают ввод исходных данных (аргумент и, по необходимости, значения коэффициентов) и вычисляют значение соответствующей функции. Отметим, что в модуле `Exp.cpp` объявлен глобальный объект `Exp f_exp`, а в модуле `Line.cpp` — глобальный объект `Line f_line`.

Конструктор класса `Menu` обеспечивает инициализацию поля `pObj` вектором объектов типа `Function*`, передаваемым через его аргумент. После копирования аргумента в поле `pObj` (в инициализаторе конструктора) к вектору с помощью метода `pObj.push_back(0)` добавляется нулевой указатель, который используется для реализации пункта меню `Exit` (см. метод `Menu::SelectObject`).

В методе `Menu::SelectObject` количество пунктов меню определяется вызовом `pObj.size`. После вывода оператором `for` списка пунктов меню вызывается метод `SelectItem`, обеспечивающий защищенный от ошибок ввод пользователем номера выбираемого пункта. Заметим, что метод `SelectItem` объявлен в секции `private` класса `Menu`, так как он используется только внутри класса.

В главном модуле `main.cpp` объявлен глобальный массив `pObjs[]`, содержащий список адресов объектов `&f_exp`, `&f_line`. Этим массивом инициализируется вектор `funcList`.

Функция `main` лаконична: в ней объявлен объект `menu` класса `Menu`, которому передается вектор `funcList`, а затем идет цикл `while`, работа которого очевидна. Метод `menu.SelectObject` возвращает адрес конкретного объекта (`f_exp` или `f_line`), через который вызывается метод `Calculate` соответствующего класса. При возврате нулевого адреса цикл завершается, и программа завершается.

Необходимо также пояснить, по какому принципу размещаются *директивы* `using` в тексте модулей. Напомним, что благодаря директиве `using namespace std` становятся видимыми (известными компилятору) все имена из пространства имен `std` стандартной библиотеки C++. В нашей программе это, например, имена `cin`, `cout`, `string`, `vector`.

Есть и другие способы сделать видимым некоторое конкретное имя: с помощью *объявления* `using` (например, `using std::string` — действует до конца блока, в котором оно сделано) или с помощью квалификатора `std::` (например, `std::string`). Проще всего использовать директиву `using`, но при этом возникают две опасности: возможность конфликта имен из пространства `std` с вашими именами и, что более неприятно, возможность неоднозначной трактовки кода компилятором при

беспорядочном употреблении директив `using`. В этом вопросе мы солидарны с Гербом Саттером [25] и приводим ниже его рекомендации.

СОВЕТ

Правило №1. *Не используйте директивы `using`, равно как и объявления `using`, в заголовочных файлах.* Именно в этом случае возможно появление указанных выше проблем. Поэтому в заголовочных файлах используйте только квалификатор `std::` перед конкретным именем.

Правило №2. *В файлах реализации директивы и объявления `using` не должны располагаться перед директивами `include`.* Иначе использование `using` может изменить семантику заголовочного файла из-за введения неожиданных имен.

Откомпилируйте и проверьте работу программы. Еще раз сравните код функций `main` в этой программе и в задаче 10.2, чтобы увидеть элегантность нового решения. А теперь представим, что от заказчика поступило предложение сделать наш калькулятор более «мощным», добавив в него новую функцию — параболу. Доработка сведется к добавлению в проект класса `Parabola`:

```
// //////////////////////////////////////// Parabola.h
#include "Function.h"
// Класс для представления функции  $y = a * x^2 + b * x + c$ 
class Parabola : public Function {
public:
    Parabola() : name("a * x^2 + b * x + c") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name; // мат. обозначение функции
    double a, b, c;
};
extern Parabola f_parabola;
// //////////////////////////////////////// Parabola.cpp
#include <iostream>
#include "Parabola.h"
using namespace std;
void Parabola::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = "; cin >> a;
    cout << "Enter b = "; cin >> b;
    cout << "Enter c = "; cin >> c;
    cout << "Enter x = "; cin >> x;
    cin.get();
    cout << "y = " << (a * x * x + b * x + c) << endl;
    cin.get();
}
Parabola f_parabola; // Глобальный объект
```

Кроме этого, в основном модуле `main.cpp` необходимо добавить директиву `#include "Parabola.h";`, а в списке инициализации массива `r0bjs` — адрес нового

объекта `&f_parabola`. Все! Весь прежний код проекта остается без изменений. Новая версия калькулятора готова.

Задача 11.2. Продвинутый функциональный калькулятор

Наш заказчик никак не уговорится. Теперь ему пришла идея расширить набор операций, которые способен выполнять функциональный калькулятор: пусть он либо вычисляет значение заданной функции для некоторого аргумента, либо выполняет табуляцию функции в заданном интервале с заданным шагом.

Идея неплохая. Это позволит резко повысить спрос на рынке на наш калькулятор, поскольку калькуляторы других фирм не умеют это делать. За работу!

Нам нужно решить две проблемы. Во-первых, видимо, придется добавить в базовый класс `Function` новый чисто виртуальный метод `Tabulation`, а также заместить его во всех подклассах. То есть придется вносить изменения в код всех классов иерархии `Function`, а это плохо. Во-вторых, в продвинутом функциональном калькуляторе необходимо реализовать двухуровневое меню: на первом уровне выбирается вид функции, на втором уровне — вид операции. В предыдущей задаче мы справились с проблемой *легко модифицируемого кода* одноуровневого меню с помощью паттерна проектирования `Switch`. Неужели на втором уровне придется писать по старинке гирлянду `case'ов` в блоке оператора `switch`? Как мы знаем, это плохо для дальнейшей модификации программы.

Попробуем найти более удачное решение. А что, если попытаться развить идею одноуровневого переключателя (паттерн `Switch`) для случая двух уровней? Например, переключатель такого типа мог бы выбирать на первом уровне некоторую конкретную сущность `EntityA_I` из иерархии абстрактного базового класса `AbstractEntityA`, а затем, на втором уровне, — некоторую конкретную сущность `EntityB_J` из иерархии абстрактного базового класса `AbstractEntityB`, после чего клиенту предоставляется возможность реализовать требуемую ассоциацию между выбранными сущностями¹.

Паттерн проектирования `DoubleSwitch`, реализующий эту идею, имеет диаграмму классов, показанную на рис. 11.10.

Обратите внимание, что метод `SelectEntityA` класса `Switch` возвращает указатель `AbstractEntityA*`, который передается далее в качестве аргумента методу `SelectEntityB`, осуществляющему выбор на втором уровне уже *для конкретной сущности первого уровня*. Абстрактный класс второго уровня `AbstractEntityB` позволяет клиенту абстрагироваться от вида выполняемой операции: он имеет дело с «операцией вообще» `Operate`.

После выбора на втором уровне благодаря полиморфизму клиент может вызвать метод `Operate`, относящийся к конкретной сущности второго уровня. Передача ссылки `AbstractEntityA*` методу `Operate` позволяет последнему иметь доступ к атомарным

¹ Красиво сказано, не правда ли?

операциям `AtomOperate1`, `AtomOperate2` и т. п., определенным в конкретном объекте иерархии `AbstractEntityA`. Совокупность этих примитивов должна быть достаточной для реализации любого метода `Operate` в иерархии `AbstractEntityB`. Как правило, это операции, обеспечивающие доступ ко всем защищенным или закрытым полям объектов иерархии `AbstractEntityA` или выполняющие обработку информации, специфичную для данного класса.

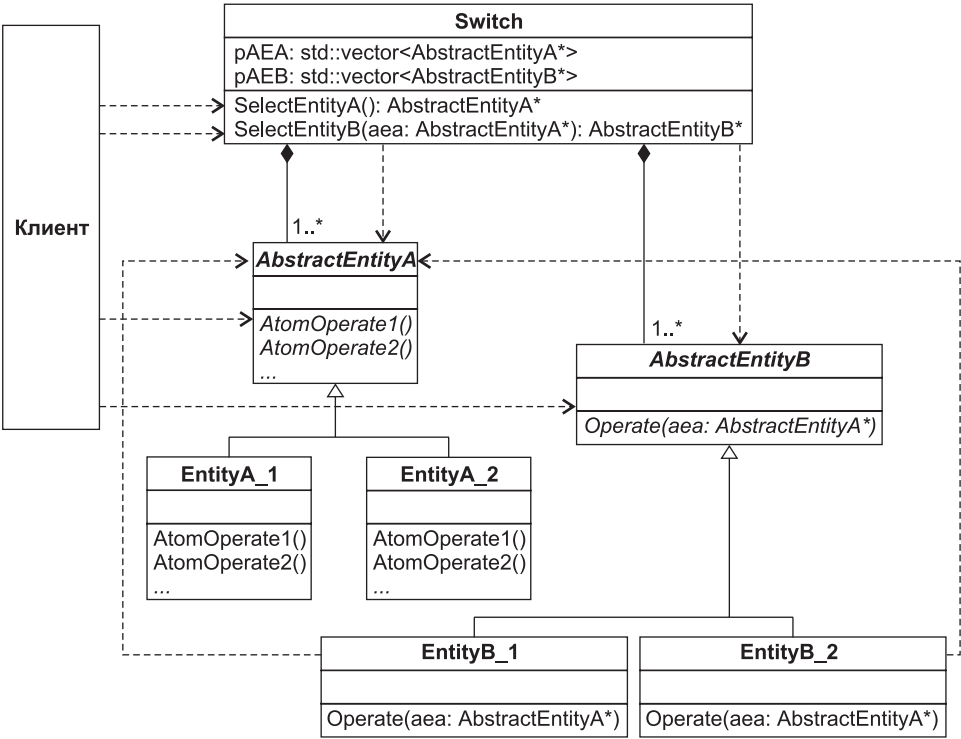


Рис. 11.10. Паттерн DoubleSwitch

Теперь попытаемся применить этот паттерн к нашей задаче. Ясно, что в качестве класса `Switch` будет выступать класс `Menu`, как и в предыдущей задаче, а в качестве класса `AbstractEntityA` — класс `Function`. Роль класса `AbstractEntityB` мы поручим новому абстрактному классу `Action`, обеспечивающему унифицированный интерфейс для конкретных классов `Calculation`, `Tabulation`, `AnyAction`. Последний класс-пустышку мы добавим просто для иллюстрации того, что список выполняемых операций может легко расширяться.

Роль метода `AbstractEntityB::Operate(AbstractEntityA*)` будет возложена на метод `Action::Operate(Function*)`. Ну и, наконец, метод `Calculate` перекочет из иерархии базового класса `Function` в конкретный класс `Calculation`, принадлежащий иерархии базового класса `Action`. Взамен мы должны пополнить иерархию `Function` такими атомарными операциями, как `SetArg` — установить значение аргумента,

SetCoeff — установить значения коэффициентов, GetVal — получить значение функции.

Мы не будем приводить здесь диаграмму классов предлагаемого решения, оставляя это в качестве упражнения читателю. Текст программы приведен в листинге 11.4.

Листинг 11.4. Продвинутой функциональный калькулятор

```
// ////////////////////////////////////// Function.h
#ifndef FUNCTION_H
#define FUNCTION_H
#include <string>
class Function {
public:
    virtual ~Function() {}
    void SetArg(double arg) { x = arg; }
    virtual void SetCoeff() = 0;
    virtual double GetVal() const = 0;
    virtual const std::string& GetName() const = 0;
protected:
    double x;      // аргумент
};
#endif /* FUNCTION_H */ // ////////////////////////////////////// Exp.h
#include <math.h>
#include "Function.h"
class Exp : public Function { // --- Класс для представления функции  $y = e^x$ 
public:
    Exp() : name("e ^ x") {}
    const std::string& GetName() const { return name; }
    void SetCoeff() {}
    double GetVal() const { return exp(x); }
private:
    std::string name; // мат. обозначение функции
};
extern Exp f_exp;
// ////////////////////////////////////// Exp.cpp
#include "Exp.h"
Exp f_exp; // Глобальный объект
// ////////////////////////////////////// Line.h
#include "Function.h"
class Line : public Function { // --- Класс для представления функции  $y = a * x + b$ 
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void SetCoeff();
    double GetVal() const { return (a * x + b); }
private:
    std::string name; // мат. обозначение функции
    double a, b;
};
```

продолжение ➞

```

extern Line f_line;
// //////////////////////////////////////// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;
void Line::SetCoeff() {
    cout << "Enter a = ";    cin >> a;
    cout << "Enter b = ";    cin >> b;
}
Line f_line;    // Глобальный объект
// //////////////////////////////////////// Action.h
#ifndef ACTION_H
#define ACTION_H
#include "Function.h"
class Action {
public:
    virtual ~Action() {}
    virtual void Operate(Function*) = 0;
    virtual const std::string& GetName() const = 0;
};
#endif /* ACTION_H */
// //////////////////////////////////////// Calculation.h
#include "Action.h"
class Calculation : public Action {
public:
    Calculation() : name("Calculation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;    // обозначение операции
};
extern Calculation calculation;
// //////////////////////////////////////// Calculation.cpp
#include <iostream>
#include "Calculation.h"
using namespace std;
void Calculation::Operate(Function* pFunc) {
    cout << "Calculation for function y = " << pFunc->GetName() << endl;
    pFunc->SetCoeff();
    double x;
    cout << "Enter x = ";    cin >> x;
    cin.get();
    pFunc->SetArg(x);
    cout << "y = " << pFunc->GetVal() << endl;
    cin.get();
}
Calculation calculation;    // Глобальный объект
// //////////////////////////////////////// Tabulation.h

```

```

#include "Action.h"
class Tabulation : public Action {
public:
    Tabulation() : name("Tabulation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;    // обозначение операции
};
extern Tabulation tabulation;
// ////////////////////////////////////// Tabulation.cpp
#include <iostream>
#include <iomanip>
#include "Tabulation.h"
using namespace std;
void Tabulation::Operate(Function* pFunc) {
    cout << "Tabulation for function y = ";
    cout << pFunc->GetName() << endl;
    pFunc->SetCoeff();
    double x_beg, x_end, x_step;
    cout << "Enter x_beg = ";    cin >> x_beg;
    cout << "Enter x_end = ";    cin >> x_end;
    cout << "Enter x_step = ";    cin >> x_step;
    cin.get();
    cout << "-----" << endl;
    cout << "      x          y" << endl;
    cout << "-----" << endl;
    for (double x = x_beg; x <= x_end; x += x_step) {
        pFunc->SetArg(x);
        cout << setw(6) << x << setw(14) << pFunc->GetVal() << endl;
    }
    cin.get();
}
Tabulation tabulation;    // Глобальный объект
// ////////////////////////////////////// AnyAction.h
#include "Action.h"
class AnyAction : public Action {
public:
    AnyAction() : name("Any action") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;    // обозначение операции
};
extern AnyAction any_action;
// ////////////////////////////////////// AnyAction.cpp
#include <iostream>
#include "AnyAction.h"
using namespace std;
void AnyAction::Operate(Function*) {

```

продолжение ⇨

Листинг 11.4 (продолжение)

```

    cout << "Здесь могла бы быть Ваша реклама!" << endl;
    cin.get();
}
AnyAction any_action; // Глобальный объект
// ////////////////////////////////////// Menu.h
#include <vector>
#include "Function.h"
#include "Action.h"
class Menu {
public:
    Menu(std::vector<Function*>, std::vector<Action*>);
    Function* SelectObject() const;
    Action* SelectAction(Function*) const;
private:
    int SelectItem(int) const;
    std::vector<Function*> pObj;
    std::vector<Action*> pAct;
};
// ////////////////////////////////////// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Function*> _pObj, vector<Action*> _pAct)
    : pObj(_pObj), pAct(_pAct) {
    pObj.push_back(0); // для выбора пункта 'Exit'
}
Function* Menu::SelectObject() const {
    // Возьмите код аналогичной функции из листинга 11.3
}
Action* Menu::SelectAction(Function* pObj) const {
    int nItem = pAct.size();
    cout << "=====\n";
    cout << "Select one of the following Action:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}
int Menu::SelectItem(int nItem) const {
    // Возьмите код аналогичной функции из листинга 11.3
}
// ////////////////////////////////////// Main.cpp
#include <iostream>
#include "Function.h"
#include "Exp.h"
#include "Line.h"
#include "Action.h"
#include "Calculation.h"

```

```

#include "Tabulation.h"
#include "AnyAction.h"
#include "Menu.h"
using namespace std;
Function* pObjjs[] = { &f_exp, &f_line };
vector<Function*> funcList(pObjjs, pObjjs + sizeof(pObjjs)/sizeof(Function*));
Action* pActs[] = { &calculation, &tabulation, &any_action };
vector<Action*> operList(pActs, pActs + sizeof(pActs) / sizeof(Action*));
int main() {
    Menu menu(funcList, operList);
    while ( Function* pObj = menu.SelectObject() ) {
        Action* pAct = menu.SelectAction(pObj);
        pAct->Operate(pObj);
    }
    cout << "Bye!\n";
}

```

Обратите внимание на следующее:

- ☐ метод `Exp::SetCoeff` имеет пустое тело, то есть ничего не делает, поскольку для вычисления экспоненты коэффициенты не требуются;
- ☐ в связи с тем, что все методы класса `Exp` — встроенные, файл реализации `Exp.cpp` содержит только объявление глобального объекта `f_exp`.

Отметим, что предложенное решение позволяет легко модифицировать наш калькулятор, решая проблемы добавления как новых функций, так и новых операций.

Задача 11.3. Работа с объектами символьных и шестнадцатеричных строк

Написать программу, демонстрирующую работу с объектами двух типов: символьная строка (`SymbString`) и шестнадцатеричная строка (`HexString`), для чего создать систему соответствующих классов.

Каждый объект должен иметь как минимум два атрибута: идентификатор и значение, представленные в виде строк символов. В поле значения объекты `SymbString` могут хранить произвольный набор символов, в то время как объекты `HexString` — только изображение шестнадцатеричного числа.

Клиенту (функции `main`) должны быть доступны следующие операции: создать объект, удалить объект, показать значение объекта (строку символов), показать изображение эквивалентного десятичного числа (только для объектов `HexString`), показать изображение эквивалентного двоичного числа (только для объектов `HexString`). Предусмотреть меню, позволяющее демонстрировать эти операции.

При решении этой задачи мы воспользуемся еще одним из видов диаграмм UML — *диаграммой видов деятельности (Activity Diagram)*, чтобы расширить наш кругозор в сфере современных методов проектирования.

Диаграмма видов деятельности UML очень похожа на старые блок-схемы алгоритмов. В ней точками принятия решений и переходами описывается последова-

тельность шагов, именуемых в UML *видами деятельности (Activity)*. Каждый вид деятельности, ассоциируемый обычно с некоторой процедурой, изображается прямоугольником с закругленными углами, а переходы между ними — стрелками. Точки принятия решений можно изображать одним из двух способов: либо просто показываются все возможные переходы после завершения некоторого вида деятельности, либо изображается переход к маленькому ромбику, похожему на блок ветвления, а затем все возможные пути выходят из этого ромбика.

Анализируя условие задачи, мы пытаемся, во-первых, выявить объекты/классы предметной области и, во-вторых, сконструировать основной алгоритм, работающий с этими объектами. В данном случае представляется естественным алгоритм, изображенный в виде диаграммы видов деятельности на рис. 11.11.

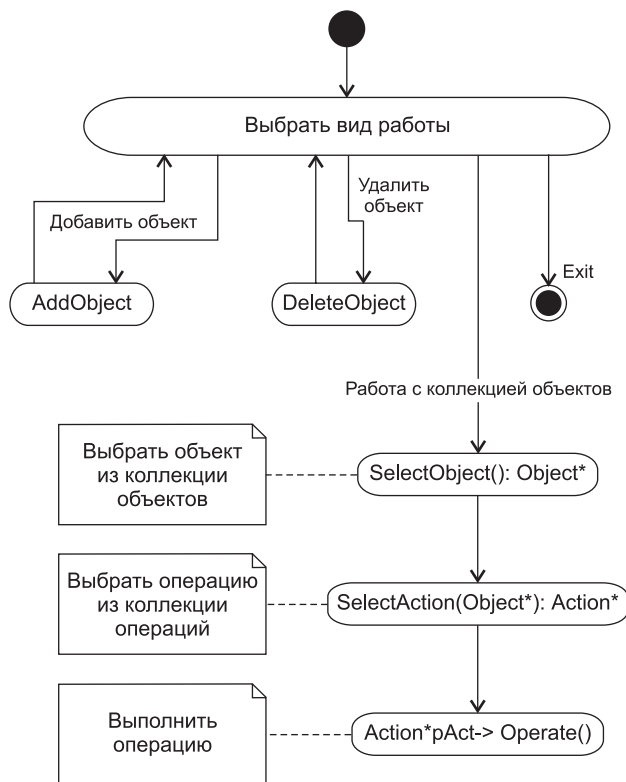


Рис. 11.11. Диаграмма видов деятельности для задачи 11.3

Начальная точка алгоритма обозначается в виде закрашенного кружка, конечная — в виде «глазка». При необходимости на диаграмме можно размещать комментарии в виде прямоугольника с текстом, напоминающего листок бумаги с отогнутым углом. Графическое представление алгоритма позволяет более четко выявить решаемые проблемы. Так, меню в нашей программе должно обеспечить выбор на трех уровнях:

- 1) выбор вида работы (добавить объект к коллекции, удалить объект из коллекции, работать с коллекцией объектов, выйти из программы);
- 2) выбор объекта из коллекции объектов;
- 3) выбор операции из коллекции операций.

Опираясь на опыт реализации двухуровневого меню в предыдущей задаче, подумаем о способе реализации трехуровневого меню. Первой приходит мысль: а не создать ли нам паттерн проектирования `ThreeLevelSwitch`, развивающий идеи паттерна `DoubleSwitch`? Однако с каждым повышением размерности переключателя код будет становиться все сложнее для понимания. Поэтому важно вовремя остановиться, помня, что любую идею можно довести до абсурда.

Принимаем следующий постулат: перечень действий (работ), выбираемых на первом уровне, тщательно продуман заказчиком, и вероятность его изменения крайне мала. В связи с этим первый уровень меню мы реализуем без использования паттернов — в блоке оператора `switch`, а вот для второго и третьего уровней применим уже испытанный паттерн `DoubleSwitch`, который позволит в будущем легко добавлять как типы обрабатываемых объектов, так и виды применяемых операций.

Теперь можно поговорить о классах. Два из них: `SymbString` и `HexString` — продиктованы условием задачи. Именно объекты этих классов будут пополнять коллекцию объектов, с которыми в дальнейшем можно выполнять те или иные операции (в нашем случае — показать значение объекта, показать изображение эквивалентного десятичного числа и т. д.). Для реализации паттерна проектирования `DoubleSwitch` нам потребуется абстрактный базовый класс `AString`, наследниками которого будут классы `SymbString` и `HexString`, а также абстрактный базовый класс `Action`, имеющий в качестве дочерних классы `ShowStr`, `ShowDec`, `ShowBin`.

Для реализации меню создадим, как и в предыдущей задаче, класс `Menu`, содержащий аналогичные методы `SelectObject` и `SelectAction`, но к ним в компанию добавим метод `SelectJob`. Поскольку список операций, применяемых к объектам, заранее известен (эти операции инкапсулированы в классах `ShowStr`, `ShowDec` и `ShowBin`), этот список, как и раньше, разместим в поле `pAct`. А вот со списком объектов ситуация более сложная, чем в задаче 11.2, так как эти объекты должны создаваться и уничтожаться динамически — в процессе работы программы. Пока отложим этот вопрос, мы вскоре к нему вернемся.

Какие сущности в решаемой задаче не охвачены перечисленными классами? В условии задачи упоминаются операции: *создать объект* и *удалить объект*. Конечно, можно поручить эти операции функции `main`, разместив заодно в ней и коллекцию обрабатываемых объектов, но это было бы в высшей степени нехорошо! Почему? — Потому, что такое решение понизило бы сцепление внутри главного клиента `main`.

СОВЕТ

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция, — отвечала за выполнение одной четко определенной задачи.

Последуем этому совету и создадим класс `Factory`, отвечающий за создание и удаление объектов из иерархии `AString`. Класс будет содержать поле `std::vector<AString*> pObj` для хранения коллекции объектов, и два метода: `AddObject` и `DeleteObject`.

Пора переходить к кодированию. Предлагаемое решение приведено в листинге 11.5.

Листинг 11.5. Работа с объектами символьных и шестнадцатеричных строк

```
// //////////////////////////////////////// AString.h
#ifndef ASTRING_H
#define ASTRING_H
#include <string>
class AString {
public:
    virtual ~AString() {}
    virtual const std::string& GetName() const = 0;
    virtual const std::string& GetVal() const = 0;
    virtual int GetSize() const = 0;
};
#endif //ASTRING_H
// //////////////////////////////////////// SymbString.h
#include <string>
#include "AString.h"
class SymbString : public AString {
public:
    SymbString(std::string _name) : name(_name) {}
    SymbString(std::string _name, std::string _val) : name(_name), val(_val) {}
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name, val;
};
// //////////////////////////////////////// HexString.h
#include <string>
#include "AString.h"
const std::string alph = "0123456789ABCDEF";
bool IsHexStrVal(std::string);
class HexString : public AString {
public:
    HexString(std::string _name) : name(_name) {}
    HexString(std::string, std::string);
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name, val;
};
```



```
// ////////////////////////////////////// HexString.cpp
#include <iostream>
#include "HexString.h"
using namespace std;
bool IsHexStrVal(string _str) { // 1
    for (int i = 0; i < _str.size(); ++i)
        if (-1 == alph.find_first_of(_str[i]))
            return false;
    return true;
}
HexString::HexString(string _name, string _val) : name(_name) {
    if (IsHexStrVal(_val)) val = _val;
}
// ////////////////////////////////////// Action.h
#ifndef ACTION_H
#define ACTION_H
#include "AString.h"
class Action {
public:
    virtual ~Action() {}
    virtual void Operate(AString*) = 0;
    virtual const std::string& GetName() const = 0;
protected:
    long GetDecimal(AString* pObj) const;
};
#endif /* ACTION_H */
// ////////////////////////////////////// Action.cpp
#include <iostream>
#include "Action.h"
#include "HexString.h"
using namespace std;
long Action::GetDecimal(AString* pObj) const {
    if (dynamic_cast<HexString*>(pObj)) {
        long dest;
        string source = pObj->GetVal();
        sscanf(source.c_str(), "%lX", &dest);
        return dest;
    }
    else { cout << "Action not supported." << endl; return -1; }
}
// ////////////////////////////////////// ShowStr.h
#include "Action.h"
class ShowStr : public Action {
public:
    ShowStr() : name("Show string value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name; // обозначение операции
};
```

продолжение ➤

Листинг 11.5 (продолжение)

```

extern ShowStr show_str;
// ////////////////////////////////////// ShowStr.cpp
#include <iostream>
#include "ShowStr.h"
using namespace std;
void ShowStr::Operate(AString* pObj) {
    cout << pObj->GetName() << ": " << pObj->GetVal() << endl;
    cin.get();
}
ShowStr show_str; // Глобальный объект
// ////////////////////////////////////// ShowDec.h
#include "Action.h"
class ShowDec : public Action {
public:
    ShowDec() : name("Show decimal value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name; // обозначение операции
};
extern ShowDec show_dec;
// ////////////////////////////////////// ShowDec.cpp
#include <iostream>
#include "ShowDec.h"
#include "HexString.h"
using namespace std;
void ShowDec::Operate(AString* pObj) {
    cout << pObj->GetName() << ": ";
    long decVal = GetDecimal(pObj);
    if (decVal != -1) cout << GetDecimal(pObj) << endl;
    cin.get();
}
ShowDec show_dec; // Глобальный объект
// ////////////////////////////////////// ShowBin.h
#include "Action.h"
class ShowBin : public Action {
public:
    ShowBin() : name("Show binary value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string GetBinary(AString*) const;
    std::string name; // обозначение операции
};
extern ShowBin show_bin;
// ////////////////////////////////////// ShowBin.cpp
#include <iostream>
#include "ShowBin.h"
#include "ShowDec.h"

```

```

#include "AString.h"
using namespace std;
void ShowBin::Operate(AString* pObj) {
    cout << pObj->GetName() << ": " << GetBinary(pObj) << endl;
    cin.get();
}
string ShowBin::GetBinary(AString* pObj) const {
    int nBinDigit = 4 * pObj->GetSize();
    char* binStr = new char[nBinDigit + 1];
    for (int k = 0; k < nBinDigit; ++k)
        binStr[k] = '0';
    binStr[nBinDigit] = 0;
    long decVal = GetDecimal(pObj);
    if (-1 == decVal) return string("");
    int i = nBinDigit - 1;
    while ( decVal > 0 ) {
        binStr[i--] = 48 + (decVal % 2);
        decVal >>= 1;
    }
    string temp(binStr);
    delete [] binStr;
    return temp;
}
ShowBin show_bin;    // Глобальный объект
// //////////////////////////////////////// Factory.h
#ifndef FACTORY_H
#define FACTORY_H
#include <vector>
#include "AString.h"
class Factory {
    friend class Menu;
public:
    Factory() {}
    void AddObject();
    void DeleteObject();
private:
    std::vector<AString*> pObj;
};
#endif //FACTORY_H
// //////////////////////////////////////// Factory.cpp
#include <iostream>
#include "Factory.h"
#include "Menu.h"
#include "SymbString.h"
#include "HexString.h"
using namespace std;
#define MAX_LEN_STR 100
void Factory::AddObject() {
    cout << "-----\n";
    cout << "Select object type:\n";

```

продолжение ➤

Листинг 11.5 (продолжение)

```

    cout << "1. Symbolic string" << endl;
    cout << "2. Hexadecimal string" << endl;
    int item = Menu::SelectItem(2);
    string name;
    cout << "Enter object name: ";
    cin >> name; cin.get();
    cout << "Enter object value: ";
    char buf[MAX_LEN_STR];
    cin.getline(buf, MAX_LEN_STR);
    string value = buf;
    AString* pNewObj;
    switch (item) {
        case 1: pNewObj = new SymbString(name, value); break;
        case 2: if (!IsHexStrVal(value)) {
            cout << "Error!" << endl;
            return;
        }
        pNewObj = new HexString(name, value); break;
    }
    pObj.push_back(pNewObj);
    cout << "Object added." << endl;
}

void Factory::DeleteObject() {
    int nItem = pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl; cin.get();
        return;
    }
    cout << ".....\n";
    cout << "Delete one of the following Object:\n";
    for (int i = 0; i < nItem; ++i)
        cout << i + 1 << ". " << pObj[i]->GetName() << endl;
    int item = Menu::SelectItem(nItem);
    string objName = pObj[item - 1]->GetName();
    pObj.erase(pObj.begin() + item - 1);
    cout << "Object " << objName << " deleted." << endl;
    cin.get();
}

// //////////////////////////////////////////// Menu.h
#include <vector>
#include "AString.h"
#include "Action.h"
#include "Factory.h"
typedef enum { AddObj, DelObj, WorkWithObj, Exit } JobMode;
class Menu {
public:
    Menu(std::vector<Action*>);
    JobMode SelectJob() const;
    AString* SelectObject(const Factory&) const;

```

```

        Action*    SelectAction(const AString*) const;
        static int SelectItem(int);
private:
        std::vector<Action*> pAct;
};
// ////////////////////////////////////// Menu.cpp
#include <iostream>
#include "AString.h"
#include "SymbString.h"
#include "HexString.h"
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Action*> _pAct) : pAct(_pAct) {}
JobMode Menu::SelectJob() const {
    cout << "=====\n";
    cout << "Select one of the following job mode:\n";
    cout << "1. Add object" << endl;
    cout << "2. Delete object" << endl;
    cout << "3. Work with object" << endl;
    cout << "4. Exit" << endl;
    int item = SelectItem(4);
    return (JobMode)(item - 1);
}
AString* Menu::SelectObject(const Factory& fctry) const {
    int nItem = fctry.pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl; cin.get();
        return 0;
    }
    cout << ".....\n";
    cout << "Select one of the following Object:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << fctry.pObj[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return fctry.pObj[item - 1];
}
Action* Menu::SelectAction(const AString* pObj) const {
    if (!pObj) return 0;
    int nItem = pAct.size();
    cout << ".....\n";
    cout << "Select one of the following Action:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}
int Menu::SelectItem(int nItem) {
    // Возьмите код аналогичной функции из листинга 11.3
}

```

продолжение ➤

Листинг 11.5 (продолжение)

```
// ////////////////////////////////////// Main.cpp
#include <iostream>
#include "AString.h"
#include "SymbString.h"
#include "HexString.h"
#include "Action.h"
#include "ShowStr.h"
#include "ShowDec.h"
#include "ShowBin.h"
#include "Factory.h"
#include "Menu.h"
using namespace std;
Action* pActs[] = { &show_str, &show_dec, &show_bin };
vector<Action*> actionList(pActs, pActs + sizeof(pActs)/sizeof(Action*));
int main() {
    Factory factory;
    Menu menu(actionList);
    JobMode jobMode;
    while ( (jobMode = menu.SelectJob()) != Exit ) {
        switch (jobMode) {
            case AddObj: factory.AddObject(); break;
            case DelObj: factory.DeleteObject(); break;
            case WorkWithObj:
                AString* pObj = menu.SelectObject(factory);
                Action* pAct = menu.SelectAction(pObj);
                if (pAct) pAct->Operate(pObj); break;
        }
        cin.get();
    }
    cout << "Bye!\n";
}
```

Обратим внимание на наиболее интересные моменты реализации.

В модуле HexString.cpp размещена глобальная функция IsHexStrVal, проверяющая, соответствует ли аргумент _str изображению шестнадцатеричного числа. Проверка выполняется с помощью метода find_first_of класса string, который возвращает индекс вхождения символа, заданного аргументом, в строку alph. Последняя объявлена в файле HexString.h и содержит набор допустимых символов для изображения шестнадцатеричного числа. Если символ не найден, метод find_first_of возвращает значение -1. Функция IsHexStrVal используется в конструкторе класса HexString, предотвращая присваивание полю val некорректного значения, а также в методе AddObject класса Factory, блокируя ошибочный ввод информации пользователем.

В базовом классе Action реализован метод GetDecimal(AString* pObj), возвращающий значение десятичного числа для передаваемого через указатель pObj изображения шестнадцатеричного числа. Почему в базовом классе? — Потому, что это значение необходимо получать как в методе ShowDec::Operate, так и в методе ShowBin::GetBinary.

В методе `Action::GetDecimal` применена технология использования *информации о типе на этапе выполнения*, сокращенно *RTTI (Run-Time Type Information)*. Дело в том, что заранее (на этапе компиляции) неизвестно, объекты каких типов будут передаваться данному методу как аргумент. Возможно появление объекта любого производного класса иерархии `AString`. В то же время из условия задачи известно, что получение десятичного и двоичного представлений корректно только для шестнадцатеричных строк. Поэтому мы используем операцию динамического приведения типов `dynamic_cast`, выполняющую *понижающее преобразование* из типа базового класса к типу производного класса. Операция возвращает адрес объекта производного класса, указанного в угловых скобках: `<HexString*>`, если преобразование возможно (если `pObj` действительно является адресом объекта класса `HexString`), либо `0` в противном случае.

В методе `ShowBin::GetBinary` сначала вычисляется десятичное значение `decVal` для шестнадцатеричной строки, а затем в цикле `while` формируется символьный массив `binStr`, содержащий символы '0' и '1' для двоичного изображения числа. Очередной символ вычисляется извлечением младшего бита из `decVal` операцией `decVal % 2` и последующим его преобразованием к коду ASCII, для чего к значению бита (0 или 1) прибавляется 48. После извлечения двоичное представление `decVal` сдвигается на один разряд вправо.

В классе `Menu` метод `SelectItem` объявлен как статический. Дело в том, что подзадачу ввода целого числа для выбора пункта меню нужно решать не только в классе `Menu`, но и в классе `Factory`. Чтобы не дублировать код, мы сделали метод `SelectItem` статическим и теперь можем вызывать его из другого класса, разумеется, предваряя операцией доступа к области видимости `Menu::`.

В методе `Factory::AddObject` добавление нового объекта к коллекции `pObj` производится вызовом метода `push_back` класса `vector`.

В методе `Factory::DeleteObject` удаление объекта с заданным адресом осуществляется с помощью метода `erase` класса `vector`. Так как значение `item` на единицу больше, чем индекс элемента в контейнере `vector`, то адрес удаляемого объекта вычисляется выражением `pObj.begin() + item - 1`, где `begin` — метод, возвращающий адрес начального элемента в контейнере.

Итоги

1. Наследование и полиморфизм — важнейшие механизмы ООП. Наследование позволяет объединять классы в семейства связанных типов, что дает им возможность совместно использовать общие поля и методы.
2. В языке C++ родительский класс называется *базовым*, а дочерний класс — *производным*. Отношения между родительскими классами и их потомками называются *иерархией наследования*.
3. Полиморфное использование объектов из одной иерархии базируется на двух механизмах: а) возможности использования указателя или ссылки на базовый класс для работы с объектами любого из производных классов, б) технологии *динамического связывания* при выборе виртуального метода через указатель на

базовый класс. Фактический выбор операции, которая будет вызвана, происходит во время выполнения программы в зависимости от типа выбранного объекта.

4. Наряду с отношением наследования, классы могут находиться и в других отношениях: *ассоциации, агрегации, композиции и зависимости (использования)*. Удобным средством отображения взаимоотношений классов является *диаграмма классов* на языке UML.
5. Современное программирование базируется не только на идеях ООП, но и на применении паттернов проектирования, аккумулирующих наиболее удачные решения типичных проблем, неоднократно возникавших при разработке ПО.

Задания

Общая часть заданий для вариантов 1–20

Написать программу, демонстрирующую работу с объектами двух типов, T1 и T2, для чего создать систему соответствующих классов. Каждый объект должен иметь идентификатор (в виде произвольной строки символов) и одно или несколько полей для хранения состояния (текущего значения) объекта.

Клиенту (функции main) должны быть доступны следующие основные операции (методы): создать объект, удалить объект, показать значение объекта и прочие дополнительные операции (зависят от варианта). Операции по созданию и удалению объектов инкапсулировать в классе Factory. Предусмотреть меню, позволяющее продемонстрировать заданные операции.

При необходимости в разрабатываемые классы добавляются дополнительные методы (например, конструктор копирования, операция присваивания и т.п.) для обеспечения надлежащего функционирования этих классов.

Варианты 1–10

В табл. 11.2 и 11.3 перечислены возможные типы объектов и возможные дополнительные операции над ними. Рассматриваются только целые положительные числа.

Таблица 11.2. Перечень типов объектов

Класс	Объект
SymbString	Символьная строка (произвольная строка символов)
BinString	Двоичная строка (изображение двоичного числа)
OctString	Восьмеричная строка (изображение восьмеричного числа)
DecString	Десятичная строка (изображение десятичного числа)
HexString	Шестнадцатеричная строка (изображение шестнадцатеричного числа)

Таблица 11.3. Перечень дополнительных операций (методов)

Операция (метод)	Описание
ShowBin()	Показать изображение двоичного значения объекта
ShowOct()	Показать изображение восьмеричного значения объекта
ShowDec()	Показать изображение десятичного значения объекта
ShowHex()	Показать изображение шестнадцатеричного значения объекта
operator +(T& s1, T& s2) ¹	для объектов SymbString — конкатенация строк s1 и s2; для объектов прочих классов — сложение соответствующих численных значений с последующим преобразованием к типу T
operator -(T& s1, T& s2)	для объектов SymbString — если s2 содержится как подстрока в s1, результатом является строка, полученная из s1 удалением подстроки s2; в противном случае возвращается значение s1; для объектов прочих классов — вычитание соответствующих численных значений с последующим преобразованием к типу T

Примечание. Первые четыре операции могут применяться к объектам любых классов, за исключением класса SymbString.

Таблица 11.4 содержит спецификации вариантов 1–10.

Таблица 11.4. Спецификации вариантов 1–10

Вариант	T1	T2	Операции (методы)
1	SymbString	BinString	ShowOct(), ShowDec(), ShowHex()
2	SymbString	BinString	operator +(T&, T&)
3	SymbString	BinString	operator -(T&, T&)
4	SymbString	OctString	operator +(T&, T&)
5	SymbString	OctString	operator -(T&, T&)
6	SymbString	DecString	ShowBin(), ShowOct(), ShowHex()
7	SymbString	DecString	operator +(T&, T&)
8	SymbString	DecString	operator -(T&, T&)
9	SymbString	HexString	operator +(T&, T&)
10	SymbString	HexString	operator -(T&, T&)

¹ Здесь T — любой из типов, T1 или T2.

Варианты 11–20

В табл. 11.5 и 11.6 перечислены возможные типы объектов и дополнительные операции над ними. Таблица 11.7 содержит спецификации вариантов 11–20.

Таблица 11.5. Перечень типов объектов

Класс	Объект	Класс	Объект
Triangle	Треугольник	Tetragon	Четырехугольник
Quadrante	Квадрат	Pentagon	Пятиугольник
Rectangle	Прямоугольник		

Таблица 11.6. Перечень дополнительных операций (методов)

Операция (метод)	Описание
Move()	Переместить объект на плоскости
Compare(T& ob1, T& ob2)	Сравнить объекты ob1 и ob2 по площади
IsIntersect(T& ob1, T& ob2)	Определить факт пересечения объектов ob1 и ob2 (есть пересечение или нет)
IsInclude(T& ob1, T& ob2)	Определить факт включения объекта ob2 в ob1

Таблица 11.7. Спецификации вариантов 11–20

Вариант	T1	T2	Операции (методы)
11	Triangle	Quadrante	Move(), Compare(T&, T&)
12	Quadrante	Pentagon	Move(), IsIntersect(T&, T&)
13	Triangle	Rectangle	Move(), Compare(T&, T&)
14	Triangle	Rectangle	Move(), IsIntersect(T&, T&)
15	Rectangle	Pentagon	Move(), IsInclude(T&, T&)
16	Triangle	Tetragon	Move(), Compare(T&, T&)
17	Triangle	Tetragon	Move(), IsIntersect(T&, T&)
18	Triangle	Tetragon	Move(), IsInclude(T&, T&)
19	Triangle	Pentagon	Move(), Compare(T&, T&)
20	Triangle	Pentagon	Move(), IsIntersect(T&, T&)

Семинар 12. Шаблоны классов. Обработка исключительных ситуаций

Теоретический материал: с. 211–230.

Шаблоны классов

Шаблоны классов наряду с шаблонами функций поддерживают парадигму *обобщенного программирования*, то есть программирования с использованием типов в качестве параметров. Механизм шаблонов в C++ допускает применение абстрактного типа в качестве параметра при определении класса или функции. После того как шаблон класса определен, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону и его аргументам называется *инстанцированием шаблона* (template instantiation).

Рассмотрим, например, точку на плоскости. Для ее представления в задаче 10.2 мы разработали класс Point, в котором положение точки задавалось координатами *x* и *y* — полями типа double. Представим, что в другом приложении требуется задавать точки для целочисленной системы координат, то есть использовать поля типа int. Можно вообразить себе системы, в которых координаты точки имеют тип short или unsigned char. Так что же — определять для каждой из этих задач новый класс Point? Бьерна Страуструпа очень раздражала такая перспектива, и он добавил в C++ поддержку того, чем мы будем заниматься на этом семинаре.

Определение шаблона класса

Определение шаблонного (обобщенного, родового) класса имеет вид

```
template <параметры_шаблона> class имя_класса { /* ... */ };
```

Например, определение шаблонного класса Point будет выглядеть так:

```
template <class T> class Point {
public:
    Point( T _x = 0, T _y = 0 ) : x( _x ), y( _y ) {}
    void Show() const { cout << "  (" << x << ", " << y << ")" << endl; };
private:
    T x, y;
};
```

Вы заметили, чем отличается это определение от определения обычного класса? Префикс `template <class T>` указывает, что объявлен шаблон класса, в котором `T` — некоторый абстрактный тип. То есть ключевое слово `class` в этом контексте задает вовсе не класс, а означает лишь то, что `T` — это параметр шаблона. Вместо `T` может использоваться любое имя. После объявления `T` используется внутри шаблона точно так же, как имена других типов. Язык позволяет вместо ключевого слова `class` перед параметром шаблона использовать другое — `typename`:

```
template < typename T> class Point { /* ... */};
```

В литературе встречаются оба стиля, но первый более распространен, так как ключевое слово `typename` появилось в языке C++ сравнительно недавно.

Определение встроенных методов внутри шаблона класса практически не отличается от записи в обычном классе. Но если определение метода выносится за пределы класса, синтаксис его заголовка усложняется. Покажем это на примере метода `Show`:

```
template <class T> class Point { // --- Версия с внешним определением метода Show
public:
    Point( T _x = 0, T _y = 0 ) : x( _x ), y( _y ) {}
    void Show() const;
private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
    cout << " (" << x << ", " << y << ")" << endl;
}
```

Обратите внимание на появление того же префикса `template <class T>`, который предварял объявление шаблона класса, а также на более сложную запись операции квалификации области видимости для имени `Show`: если раньше мы писали `Point::`, то теперь пишем `Point<T>::`, добавляя к имени класса список параметров шаблона в угловых скобках (в данном случае — один параметр `T`). На первых порах это сложно запомнить (что является причиной постоянных ошибок компиляции), но, написав пару десятков шаблонных классов, вы привыкнете...

Использование шаблона класса

При включении шаблона класса в программу никакие классы на самом деле не генерируются до тех пор, пока не будет создан экземпляр шаблонного класса, в котором вместо абстрактного типа `T` указывается некоторый конкретный тип. Такая подстановка приводит к *актуализации*, или *инстанцированию*, шаблона. Как и для обычного класса, экземпляр создается либо объявлением объекта, например:

```
Point<int> aPoint( 13, -5 );
```

либо объявлением указателя на актуализированный шаблонный тип с присваиванием ему адреса, возвращаемого операцией `new`, например:

```
Point<double>* pPoint = new Point<double>( 9.99, 3.33 );
```

Встретив такие объявления, компилятор генерирует код соответствующего класса.

Организация исходного кода

В многофайловом проекте определение шаблона класса обычно выносится в отдельный файл. В то же время для инстанцирования компилятором конкретного экземпляра шаблона класса необходимо, чтобы определение шаблона находилось *в одной единице трансляции* с данным экземпляром.

В связи с этим принято размещать *все определение* шаблонного класса в некотором заголовочном файле, например `Point.h`, и подключать его к нужным файлам с помощью директивы `#include`. Для предотвращения повторного включения этого файла, которое может возникнуть в многофайловом проекте, обязательно используйте «стражи включения» (см. с. 115).

Продemonстрируем эту технику на примере нашего класса `Point` (листинг 12.1).

Листинг 12.1. Шаблонный класс `Point`

```

//////////////////////////////////// Point.h ///////////////////////////////////
#ifndef POINT_H
#define POINT_H
template <class T> class Point {
public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const;
private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
    cout << " (" << x << ", " << y << ")" << endl;
}
#endif /* POINT_H */
//////////////////////////////////// Main.cpp ///////////////////////////////////
#include <iostream>
#include "Point.h"
using namespace std;
int main() {
    Point<double> p1;                // 1
    Point<double> p2(7.32, -2.6);    // 2
    p1.Show(); p2.Show();
    Point<int> p3(13, 15);           // 3
    Point<short> p4(17, 21);         // 4
    p3.Show(); p4.Show();
}

```

Обратите внимание на использование шаблонного класса `Point` клиентом `main`: в *строках 1 и 2* шаблон инстанцируется в конкретный класс `Point` с подстановкой вместо `T` типа `double`, в *строке 3* — в класс `Point` с подстановкой типа `int`, в *строке 4* — с подстановкой типа `short`.

Заметим, что наиболее широкое применение шаблоны классов нашли при создании контейнерных классов стандартной библиотеки шаблонов (STL), предназначенных для работы с такими стандартными структурами, как вектор, список, очередь,

множество и т. д. Кстати, на семинаре 11 мы уже воспользовались одним из этих классов, а именно классом `vector`. Так что сейчас вы можете бросить беглый ретроспективный взгляд на пройденный материал, испытывая чувство глубокого удовлетворения от более тонкого понимания новой материи — шаблонных классов.

Параметры шаблонов

Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов, таких как `int`. Стандарт C++ допускает также использование параметров-шаблонов, но это реализовано далеко не во всех компиляторах. *Абстрактные типы* в качестве параметров мы уже рассмотрели: при инстанцировании на их место подставляются аргументы либо встроенных, либо описанных программистом типов.

Переменные встроенных типов используются как параметры шаблонов, когда для шаблона предусматривается его настройка некоторой константой. Например, можно создать шаблон для массивов, содержащих `n` элементов типа `T`:

```
template <class T, int n> class Array { /*...*/ };
Array<Point, 20> ap;                      // Создание массива из 20 элементов типа Point
```

Приведем менее тривиальный пример использования параметров второго вида:

```
void f1() { cout << "I am f1()." << endl; }           // Директивы препроцессора для
void f2() { cout << "I am f2()." << endl; }           // краткости опущены
template<void (*pf)()> struct A { void Show() { pf(); } };
int main() {
    A<&f1> aa;
    aa.Show();    // вывод: I am f1().
    A<&f2> ab;
    ab.Show();    // вывод: I am f2().
}
```

Здесь параметр шаблона имеет тип указателя на функцию. При инстанцировании класса в качестве аргумента подставляется адрес соответствующей функции (адрес функции также является константой, создаваемой компилятором).

Естественно, у шаблона может быть несколько параметров. Например, ассоциативный контейнер `map` из библиотеки STL имеет следующее определение:

```
template <class Key, class T, class Compare = less<Key> >
class map { /*...*/ };
```

Из последнего примера видно, что параметры шаблона так же, как и параметры обычных функций, могут иметь значения по умолчанию. Обратите внимание на наличие пробела между двумя последними символами `>`. Если не поставить пробел, компилятор воспримет последовательность `>>` как операцию сдвига вправо и выдаст сообщение об ошибке, текст которого слабо проясняет ее истинную причину.

Специализация

Иногда возникает необходимость определить специализированную версию шаблона для некоторого конкретного типа одного из его параметров. Например, невозможно создать обобщенный алгоритм, проверяющий отношение `<` (меньше) для

двух аргументов типа T, который одновременно подходил бы и для числовых типов, и для традиционных C-строк, завершающихся нулевым байтом. В таких случаях применяется специализация шаблона, имеющая следующую общую форму:

```
template <> class имя_класса <имя_специализированного_типа> { /* ... */ };
```

Например:

```
template <class T> class Sample { // Общий шаблон
    bool Less(T) const; /*...*/ };
template <> class Sample<char*> { // Специализация для char*
    bool Less(char*) const; /*...*/ };
```

Если у класса-шаблона несколько параметров, возможна *частичная специализация*:

```
template <class T1, class T2> class Pair { /*...*/ }; // Общий шаблон
// специализация, где для T2 установлен тип int:
template <class T1> class Pair <T1, int> { /*...*/ };
```

В любом случае общий шаблон должен быть объявлен прежде любой специализации.

Иногда есть смысл специализировать не весь класс, а какой-либо из его методов:

```
// обобщенный метод:
template <class T> bool Sample<T>::Less(T ob) const { /*...*/ };
// специализированный метод:
void Sample<char*>::Less(char* ob) const { /*...*/ };
```

Использование функциональных объектов для настройки шаблонных классов

Напомним, что *функциональным объектом* называется объект, для которого определена операция вызова функции `operator()`. Соответственно, класс, экземпляром которого является этот объект, называется *классом функционального объекта* или просто *функциональным классом*. Приведем пример (листинг 12.2).

Листинг 12.2. Использование функционального класса

```
#include <iostream>
using namespace std;
struct LessThan { // функциональный класс с единственной операцией operator()
    bool operator() ( const int x, const int y ) { return x < y; }
};
int main() {
    LessThan lt; // объявлен объект lt функционального класса LessThan
    int a = 5, b = 9;
    if ( lt( a, b ) ) cout << a << " less than " << b << endl; // 2
    if ( LessThan()( a, b ) ) cout << a << " less than " << b << endl; // 3
}
```

Как вы помните, `struct` — это вид класса, в котором все элементы по умолчанию открыты. Объект `lt` используется в операторе `if` (*оператор 2*) для вызова функции `operator()`, передавая ей аргументы `a` и `b`.

В *операторе 3* демонстрируется способ использования функционального объекта без его предварительного объявления. Запись `LessThan()` означает создание анонимного экземпляра класса `LessThan`, то есть функционального объекта. Запись `LessThan()(a, b)` означает вызов функции `operator()` с передачей ей аргументов `a` и `b`. Результаты выполнения обоих операторов `if` одинаковы.

Возможно, вы удивитесь, какой смысл использовать здесь класс `LessThan`, вместо того чтобы просто написать `if (a < b)`, и будете абсолютно правы. Но ситуация резко меняется, когда один из параметров шаблонного класса используется для настройки класса на некоторую стратегию.

Рассмотрим пример. Вас вызвал шеф и дал задание создать шаблонный класс `PairSelect`, предназначенный для выбора одного значения из пары значений, хранящихся в этом классе. Выбор выполняется в соответствии с критерием (стратегией), который передается как параметр шаблона. Параметр должен называться `class Compare`. Для начала нужно реализовать две стратегии: `LessThan` (первое значение меньше второго) и `GreaterThan` (первое значение больше второго).

Анализируя постановку задачи, вы приходите к заключению, что реализация стратегий в виде глобальных функций `LessThan()` и `GreaterThan()` быстро заведет в тупик, так как компилятор не позволит передать адреса функций на место параметра `class Compare`. Остается единственное решение — использовать функциональные классы (листинг 12.3).

Листинг 12.3. Шаблонный класс `PairSelect`

```
template <class T> struct LessThan {
    bool operator() ( const T& x, const T& y ) { return x < y; }
};
template <class T> struct GreaterThan {
    bool operator() ( const T& x, const T& y ) { return x > y; }
};
template <class T, class Compare>
class PairSelect {
public:
    PairSelect( const T& x, const T& y ) : a( x ), b( y ) {}
    void OutSelect() const {
        cout << ( Compare()( a, b ) ? a : b ) << endl;
    }
private:
    T a, b;
};
int main() {
    PairSelect<int, LessThan<int> > ps1( 13, 9 );
    ps1.OutSelect(); // вывод: 9
    PairSelect<double, GreaterThan<double> > ps2( 13.8, 9.2 );
    ps2.OutSelect(); // вывод: 13.8
}
```

Обратите внимание на использование функционального объекта `Compare()(a, b)` в методе `OutSelect`, а также на настройку шаблонного класса `PairSelect` при его

инстанцировании. При первом инстанцировании (объявление объекта `ps1`) второму аргументу класса `PairSelect` передается функциональный класс `LessThan<int>`. При втором инстанцировании (объявление `ps2`) передается `GreaterThan<double>`.

Разработка шаблонного класса для представления разреженных массивов

Для закрепления материала разработаем шаблонный класс, предназначенный для представления *разреженных массивов* — массивов, в которых не все элементы фактически используются, присутствуют на своих местах или нужны.

Эта структура данных появилась в процессе решения научных и инженерных задач, в которых нужны многомерные массивы очень большого объема, но в то же время на каждом этапе обработки информации фактически используется только незначительная часть элементов массива. Например, это приложения, связанные с анализом матриц, или *электронные таблицы*, использующие матрицу для хранения формул, значений и строк, ассоциируемых с каждой из ячеек. Благодаря использованию разреженных массивов память для хранения каждого из элементов выделяется из пула свободной памяти только по мере надобности.

Для простоты изложения будем рассматривать одномерные разреженные массивы, так как все обсуждаемые идеи несложно применить и для реализации многомерных массивов. В среде приличных людей, так или иначе связанных с разреженными массивами, в ходу два термина: «*логический массив*» и «*физический массив*». Логический массив является воображаемым, а физический массив — это массив, фактически существующий в системе. Например, если вы определите разреженный массив:

```
SparseArr sa(1000000);
```

логический массив будет состоять из 1 000 000 элементов даже в случае, если этот массив в системе физически не существует. Если же фактически используется 100 элементов массива, то только они и будут занимать физическую память компьютера.

Обычно каждый элемент физического разреженного массива содержит как минимум два поля: логический индекс элемента и его значение. Для хранения физического массива, как правило, используют одну из динамических структур данных.

Задача 12.1. Шаблонный класс для разреженных массивов

Разработать шаблонный класс для представления разреженных одномерных массивов. Размер логического массива передавать через аргумент конструктора.

Класс должен обеспечивать хранение данных любого типа T , для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать операцию индексирования, возвращающую ссылку на найденный элемент. Если элемент с заданным индексом не найден, операция должна создать новый элемент с этим индексом и поместить его в массив. При необходимости добавить в класс другие методы. В клиенте `main` продемонстрировать использование этого класса.

Заметим, что согласно условию задачи нужно разработать очень примитивный класс с минимальной функциональностью. В реальных приложениях такой класс будет, конечно, содержать и другие методы, например, удаление из физического массива элемента с заданным индексом.

Выше было сказано, что каждый элемент физического массива должен содержать два поля: логический индекс элемента и его значение. Поэтому начнем с разработки серверного класса для представления одного элемента физического массива. Этот класс также должен быть шаблонным, иначе как же он же будет использоваться клиентом — шаблонным же классом разреженного массива? Результатом проработки этого вопроса может быть, например, следующий класс:

```
template <class DataT> class SA_item {
public:
    SA_item( long i, DataT d ) : index( i ), info( d ) {}
    long index;
    DataT info;
};
```

Второй вопрос, который нужно решить до начала кодирования, — какую структуру данных мы выберем для хранения физического массива и какими средствами ее реализуем? Чаще всего используются линейные списки, бинарные деревья или структуры данных с хешированием индексов.

Линейный список имеет наихудшие показатели по времени поиска информации с заданным ключом (индексом), что может иметь существенное значение, если количество элементов в физическом массиве достаточно велико, но в то же время он наиболее прост для программирования. Более того, чтобы не отвлекаться на детали реализации и написать компактный код, мы воспользуемся контейнерным классом `list` из STL. Вообще-то контейнерным классам STL будет посвящен семинар 15, но у нас уже есть опыт использования класса `vector` на семинаре 11; так же и здесь, приведя минимально необходимые сведения о классе `list`, мы сможем воспользоваться многими его преимуществами.

Контейнерный класс `list` является шаблонным классом и реализован в STL в виде двусвязного списка, каждый узел которого содержит ссылки на последующий и предыдущий элементы. Для использования класса необходимо подключить заголовочный файл `<list>`. В классе есть конструктор по умолчанию, создающий список нулевой длины. Можно добавить в конец имеющегося списка новый элемент с помощью метода `push_back`. Доступ к любому элементу списка осуществляется через *итератор* — переменную типа `list<T>::iterator`. Поскольку с понятием итератора вы еще не знакомы, скажем о нем несколько слов.

Проще всего рассматривать итератор как указатель на элемент списка. Он используется для просмотра списка в прямом или обратном направлении. В первом случае к итератору применяется операция инкремента, во втором — декремента.

В классе `list` есть два метода, позволяющие организовать просмотр всех элементов списка: `begin` возвращает указатель на первый элемент, `end` возвращает указатель на элемент, следующий за последним. Текущее значение итератора в цикле

сравнивается со значением, полученным от метода `end`, с помощью операции `!=`, так как из-за произвольного размещения в памяти соседних элементов списка операция `<` для адресов элементов теряет смысл. Поясним применение класса `list` на примере:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<char> v1;
    v1.push_back( 'A' );
    v1.push_back( 'B' );
    v1.push_back( 'C' );
    list<char>::iterator i = v1.begin();
    list<char>::iterator n = v1.end();
    for ( ; i != n; ++i )
        cout << *i << ' '; // содержимое ячейки памяти, на которую указывает i
    cout << endl;
}
```

Мы можем позволить себе не комментировать этот пример, так как уровень ваших знаний на текущий момент, безусловно, достаточен, чтобы понять все с полувзгляда. Теперь все готово, чтобы привести возможное решение задачи (листинг 12.4).

Листинг 12.4. Шаботонный класс для разреженных массивов

```
//////////////////////////////////// SparseArr.h //////////////////////////////////
#ifndef SPARSE_ARR_H
#define SPARSE_ARR_H
#include <list>
template <class DataT> class SA_item {
public:
    SA_item() : index( -1 ), info( DataT() ) {}
    SA_item( long i, DataT d ) : index( i ), info( d ) {}
    long index;
    DataT info;
};
template <class T> class SparseArr {
public:
    SparseArr( long len ) : length( len ) {}
    T& operator [] ( long ind );
    void Show( const char* );
private:
    std::list<SA_item<T> > arr; // 1
    long length;
};
template <class T>
void SparseArr<T>::Show( const char* title ) {
    cout << "==== " << title << "====\n";
    list<SA_item<T> >::iterator i = arr.begin();
```

продолжение ➤

Листинг 12.4 (продолжение)

```

    list<SA_item<T> >::iterator n = arr.end();
    for ( ; i != n; ++i )
        cout << i->index << "\t" << i->info << endl;
}
template <class T>
T& SparseArr<T>::operator[]( long ind ) {
    if ( ( ind < 0 ) || ( ind > length-1 ) ) {
        cerr << "Error of index: " << ind << endl;
        SA_item<T> temp;
        return temp.info;
    }
    list<SA_item<T> >::iterator i = arr.begin();
    list<SA_item<T> >::iterator n = arr.end();
    for ( ; i != n; ++i )
        if ( ind == i->index ) return i->info;                // элемент найден
                                                                // Элемент не найден, создаем новый элемент
    arr.push_back( SA_item<T>( ind, T() ) );                  // 2
    i = arr.end();
    return (--i)->info;
}
#endif    /* SPARSE_ARR_H */
// Main.cpp
#include <iostream>
#include <string>
#include "SparseArr.h"
using namespace std;
int main() {
    SparseArr<double> sal( 2000000 );                          // 3
    sal[127649] = 1.1;                                         // 4
    sal[38225] = 1.2;                                          // 5
    sal[2004056] = 1.3;                                        // 6
    sal[1999999] = 1.4;                                        // 7
    sal.Show("sal");                                          // 8
    cout << "sal[38225] = " << sal[38225] << endl;           // 9
    sal[38225] = sal[93];                                     // 10
    cout << "After the modification of sal:\n";
    sal.Show( "sal" );
    SparseArr<string> sa2( 1000 );                            // 11
    sa2[333] = "Nick";
    sa2[222] = "Peter";
    sa2[444] = "Ann";
    sa2.Show( "sa2" );
    sa2[222] = sa2[555];
    sa2.Show( "sa2" );
}

```

Обратите внимание на следующие моменты. Поле *arr* (*оператор 1*) в шаблонном классе *SparseArr*, объявленное как объект шаблонного класса *list*, актуализированного шаблонным же параметром *SA_item<T>*, предназначено для хранения

физического массива. Аргумент `T` здесь совпадает с параметром шаблонного класса `SparseArr`.

В операции индексирования `operator[]()` прежде всего проверяется, не выходит ли значение индекса `ind` за допустимые границы. Если выходит, формируется сообщение об ошибке, выводимое в поток `cerr`¹, после чего нужно либо прервать выполнение программы, либо вернуть некоторое приемлемое значение. В данном случае выбран второй вариант: создается временный объект `temp` с вызовом конструктора по умолчанию `SA_item()` и возвращается ссылка на его поле `info`.

Теперь обратим наши взоры на конструктор по умолчанию в классе `SA_item` — в первоначальном варианте класса, рассмотренном выше, этого конструктора не было. Здесь же он добавлен именно для использования в нестандартных ситуациях: конструктор создает клон объекта физического массива со значением индекса, равным `-1`. Этим гарантируется, что созданный элемент нигде и никогда не будет использован. Таким образом, после вывода сообщения об ошибке программа продолжит свое выполнение.

Вряд ли стоит считать такое решение идеальным: ведь память будет замусориваться неиспользуемыми объектами. С другой стороны, терминальное прерывание работы программы вызовом `exit` тоже не назовешь эстетичным решением. Короче говоря, перед нами — классическая ситуация, требующая генерации и последующей обработки исключения, но эту тему мы рассмотрим чуть ниже. А пока вернемся к анализу работы операции индексирования.

Если с индексом все в порядке, то далее в цикле `for` ищется элемент физического массива с заданным индексом. В случае успеха возвращается ссылка на поле `info` найденного элемента. В случае неуспеха создается новый элемент физического массива и добавляется к списку `arr` (*оператор 2*). При этом полю `info` присваивается значение, сформированное конструктором по умолчанию `T()`. Затем вызывается метод `arr.end`, возвращающий указатель (итератор) на элемент, следующий за последним элементом списка. Чтобы получить доступ к последнему элементу, применяется операция *префиксного* декремента `--i`. Полученное значение итератора позволяет вернуть из функции ссылку на поле `info` нового элемента.

В класс добавлен метод `Show`, который просто выводит в поток `cout` перечень элементов (индексы и значения) физического массива.

В клиенте `main` показано два варианта инстанцирования класса `SparseArr`. В первом случае (*оператор 3*) создается логический массив `sal` из 2 000 000 элементов типа `double`. Поначалу он не содержит физических элементов. Выполнение *оператора 4* начнется с вызова операции `sal.operator[](127649)`. Так как элемента с указанным индексом в массиве нет, будет создан элемент, имеющий индекс 127649 и значение 0.0 (значение для типа `double` по умолчанию), и добавлен в конец массива. Операция индексирования вернет ссылку на поле `info` со значением 0.0, выполняемая

¹ `cerr` — объект класса `ostream`, представляющий стандартное устройство для вывода сообщений об ошибках, который аналогично объекту `cout` направляет выводимые данные на терминал пользователя. Различие состоит в том, что вывод объекта `cerr` не буферизируется и отображается немедленно.

следом операция присваивания изменит это значение на 1.1. Выполнение *операторов 5 и 7* аналогично, а вот при выполнении *оператора 6* будет обнаружена ошибка индексирования.

В *операторе 9* проверяется обращение к существующему элементу массива для его вывода в поток cout. Интересным является *оператор 10* — в результате его выполнения элемент `sal[38225]` получит значение 0. Если вы внимательно читали предыдущие пояснения, такой результат не вызовет у вас вопросов. В итоге вывод на экран первой части программы (работа с массивом `sal`) будет следующим:

```
Error of index: 2004056
===== sal =====
127649  1.1
38225   1.2
1999999 1.4
sal[38225] = 1.2
After the modification of sal:
===== sal =====
127649  1.1
38225   0
1999999 1.4
93      0
```

Во втором случае (*оператор 11*) создается логический массив `sa2` из 1000 элементов типа `string`. Действия по проверке его использования аналогичны предыдущим. Вывод на экран второй части программы будет следующим:

```
===== sa2 =====
333 Nick
222 Peter
444 Ann
===== sa2 =====
333 Nick
222
444 Ann
555
```

Обработка исключительных ситуаций

При решении предыдущей задачи мы столкнулись с проблемой — что делать методу класса (или, например, операции индексирования), если он обнаруживает некоторую ошибку, вызванную некорректным обращением клиента к методу (недопустимое значение индекса)? Инструментарий, которым мы пользовались до сих пор, начиная с семинара 7, позволял предпринять одно из следующих решений:

- ☐ прервать выполнение программы;
- ☐ вернуть значение, означающее «ошибка»;
- ☐ вывести сообщение об ошибке в поток `cerr` и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжить работу.

Первый вариант решения не понравится пользователю программы, так как в самый неподходящий момент она будет «ломаться» без всякого уведомления о причине своей гибели. Второй вариант возможен лишь тогда, когда возвращаемое значение функции предназначено именно для кодирования статуса ее завершения. Так бывает далеко не всегда: например, в операции индексирования `operator[]()` класса `SparseArr` возвращаемое значение есть ссылка на поле `info` объекта. Ну и наконец, третий вариант решения тоже часто связан с труднорешаемыми вопросами: что есть «приемлемое значение» и почему программа продолжает свою работу независимо от обнаруженной ошибки?

С подобными проблемами часто сталкиваются разработчики промышленных библиотек классов. Автор библиотечного класса может обнаружить ошибки времени выполнения, но в общем случае не знает, как должен на них реагировать клиент. Для решения подобных проблем в C++ были введены средства генерации и обработки *исключений* (*exception*). Заметим, что такими средствами пользуются не только при разработке библиотек. Например, в процессе выполнения конструктора класса может возникнуть какая-то нештатная ситуация (скажем, нехватка памяти). Поскольку конструктор не имеет возвращаемого значения, единственным способом для него уведомить об этом клиента также является генерация исключения.

Определение исключений

Для того чтобы работать с исключениями, необходимо:

- ❑ выделить *контролируемый блок* — составной оператор, перед которым записано ключевое слово `try`:

```
try {  
    // фрагмент кода  
}
```

- ❑ предусмотреть генерацию одного или нескольких исключений операторами `throw` внутри блока `try` или внутри функций, вызываемых из этого блока;
- ❑ разместить сразу за блоком `try` один или несколько обработчиков `catch`.

Оператор `throw`, предназначенный для *генерации исключения*, имеет вид `throw выражение`. Тип выражения определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, происходит поиск соответствующего обработчика и передача ему управления.

Синтаксис *обработчиков исключений* напоминает определение функции с одним параметром и именем `catch`:

```
catch (/* ... */) {  
    // действия по обработке исключения  
}
```

Объявление параметра обработчика возможно в одной из трех форм:

```
catch (Type) {  
} // Форма 1: обработка исключения типа Type  
catch (Type info) {  
} // Форма 2: обработка исключения типа Type
```

продолжение ➤

```

// с использованием значения info
}
catch (...) { // Форма 3: обработка исключений всех типов
}

```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками. Туда же, минуя все обработчики, передается управление, если исключение в try-блоке не было сгенерировано.

Если обработчик не в состоянии полностью обработать ошибку, он может сгенерировать исключение повторно с помощью оператора `throw` без параметров. В этом случае предполагается наличие внешних объемлющих блоков, в которых может находиться другой обработчик для этого типа исключения.

Перехват исключений

Когда с помощью `throw` генерируется исключение, функции исполнительной библиотеки C++ выполняют следующие действия:

- ❑ создают копию параметра `throw` в виде статического объекта, который сохраняется до тех пор, пока исключение не будет обработано;
- ❑ в поисках подходящего обработчика *раскручивают стек* (об этом чуть ниже);
- ❑ передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

А что такое подходящий разработчик? Рассмотрим такой пример:

```

try {
    throw E();
}
catch ( N ) {
    // когда мы сюда попадем?
}

```

Обработчик будет вызван, если

- ❑ `N` и `E` — одного типа;
- ❑ `N` является открытым базовым классом для `E`;
- ❑ `N` и `E` — указатели либо ссылки и для них справедливо одно из предыдущих утверждений.

Следующая маленькая программка демонстрирует перехват исключений:

```

class A {
public:
    A() { cout << "Constructor of A\n"; }
    ~A() { cout << "Destructor of A\n"; }
};
class Error {};
class ErrorOfA : public Error {};
void foo() {
    A a;
    throw 1;
    cout << "This message is never printed" << endl;
}

```



```
int main() {
    try {
        foo();
        throw ErrorOfA();
    }
    catch( int )      { cerr << "Catch of int\n"; }
    catch( ErrorOfA ) { cerr << "Catch of ErrorOfA\n"; }
    catch( Error )    { cerr << "Catch of Error\n"; }
}
```

Она выдаст следующий результат:

```
Constructor of A
Destructor of A
Catch of int
```

Первым генерируется исключение `throw 1` внутри функции `foo`. Исполнительная система создает копию объекта «целая константа 1» и начинает поиск подходящего обработчика. Поскольку внутри `foo` ничего подходящего нет, поиск переносится вовне — в клиент `main`. Но (важная деталь!) перед тем, как покинуть тело функции `foo`, система вызывает деструкторы всех ее локальных объектов! Это и есть *раскрутка стека*.

В функции `main` нужный обработчик обнаруживается и дает о себе знать сообщением «Catch of int». Таким образом, генерация второго исключения `throw ErrorOfA()` здесь невозможна.

Поменяйте местами строки в блоке `try` и запустите программу снова. Теперь исключение `throw ErrorOfA()` будет сгенерировано и вы получите результат:

```
Catch of ErrorOfA
```

Интересно, что класс `ErrorOfA` совершенно пустой. Но этого достаточно, чтобы использовать его экземпляр (в данном случае анонимный) для генерации исключения. А теперь закомментируйте строку с обработчиком `catch(ErrorOfA)`. Программа выдаст:

```
Catch of Error
```

Неперехваченные исключения

Если исключение сгенерировано, но не перехвачено, вызывается стандартная функция `std::terminate`. Она будет вызвана и в том случае, если механизм обработки исключения обнаружит, что стек разрушен, или если деструктор, вызванный во время раскрутки стека, пытается завершить свою работу при помощи исключения. По умолчанию `terminate` вызывает функцию `abort`.

Чтобы увидеть, что происходит при вызове этих функций, скомпилируйте и выполните следующую программу:

```
int main() {
    throw 5;    // произвольное значение
    return 0;
}
```

На нашем компьютере (Windows XP и Visual Studio 2005) на экран вываливается окно с сообщением об аварийном завершении программы:

Debug Error!... This application has requested the Runtime to terminate it in an unusual way.

Вы можете заменить, если захотите, вызов функции `abort` вызовом своего обработчика. Это делается с помощью функции `set_terminate`. Например:

```
void SoftAbort() {
    cerr << "Program is terminated." << endl;
    exit( 1 );
}
int main() {
    set_terminate( SoftAbort );
    throw 5;
    return 0;
}
```

Проверьте, что завершение программы теперь действительно окажется более мягким.

Классы исключений. Иерархии исключений

Отметим, что, хотя язык позволяет генерировать исключения любого встроенного типа (как, например, типа `int` в рассмотренных выше примерах), в реальных программных системах такие исключения используются редко. Гораздо удобнее создавать специальные классы исключений (такие, как `ErrorOfA`) и использовать в операторе `throw` либо объекты этих классов, либо анонимные экземпляры (через вызов конструктора класса).

При необходимости в этих классах можно передавать через параметры конструктора и сохранять для последующей обработки любую информацию о состоянии программы в момент генерации исключения. Другое преимущество этого подхода — возможность создания иерархии исключений. Например, исключения для математической библиотеки можно организовать следующим образом:

```
class MathError { /* ... */ }; // Базовый класс обработки ошибок
class Overflow : public MathError { /* ... */ }; // Класс ошибки переполнения
class ZeroDivide : public MathError { /* ... */ }; // Класс ошибки "деление на 0"
```

Если не лениться и добавить в базовый класс виртуальный метод `ErrProcess`, заместив его в производных классах версиями `ErrProcess`, ориентированными на обработку конкретного типа ошибки, это позволит задавать после блока `try` единственный обработчик `catch`, принимающий объект базового класса:

```
try {
    // генерация любых исключений MathError, Overflow, ZeroDivide, ...
}
catch (MathError& me) {
    me.ErrProcess() // Обработка любого исключения
}
```

На этапе выполнения такой обработчик `catch` будет обрабатывать как исключения типа `MathError`, так и исключения любого производного типа, причем благодаря полиморфизму каждый раз будет вызываться версия метода `ErrProcess`, соответствующая именно данному типу исключения.

Организация исключений в виде иерархий исключительно важна, если ставится цель разработать легко модифицируемое программное обеспечение. Ведь при структуре кода, которую мы рассмотрели, добавление в систему нового вида исключения, входящего в существующую иерархию, вообще не потребует изменять написанные ранее фрагменты кода, предназначенные для перехвата и обработки исключений.

Спецификации исключений

В заголовке функции можно задать список типов исключений, которые она может прямо или косвенно порождать. Этот список приводится в конце заголовка и предваряется ключевым словом `throw`. Например:

```
void Func(int a) throw (Foo1, Foo2);
```

Такое объявление означает, что `Func` может сгенерировать только исключения `Foo1`, `Foo2` и исключения, являющиеся производными от этих типов, но не другие. Заголовок является интерфейсом функции, поэтому такое объявление дает пользователям функции определенные гарантии. Это очень важно при использовании библиотечных функций, так как определения таких функций не всегда доступны.

Что произойдет, если функция вдруг нарушит взятые на себя обязательства и в ее недрах будет возбуждено исключение, не соответствующее списку спецификации исключений? Тогда система вызовет обработчик `std::unexpected`, который может попытаться сгенерировать свое исключение (это зависит от реализации), и если оно не будет противоречить спецификации, то продолжится поиск подходящего обработчика. В противном случае вызывается `std::terminate`.

Если вас не устраивает поведение `unexpected` по умолчанию, вы можете установить собственную функцию, которую он будет вызывать. Для этого нужно воспользоваться функцией `set_unexpected`.

Если спецификация исключений задана в виде `throw()`, это означает, что функция вообще не генерирует исключений. Отсутствие спецификации исключений в заголовке функции, то есть запись заголовка в привычном нам виде, означает, что функция может сгенерировать любое исключение.

Если заголовок функции содержит спецификацию исключений, то каждое объявление этой функции (включая определение) должно иметь спецификацию исключений с точно таким же набором типов исключений. Виртуальная функция может быть замещена в производном классе функцией с не менее ограничительной спецификацией исключений, чем ее собственная.

Спецификация исключений не является частью типа функции, поэтому `typedef` не может ее содержать. Например:

```
// typedef int (*PF)() throw( A );    // ошибка!
```

Исключения в конструкторах

Исключения предоставляют единственную возможность передать информацию об обломе (срыве, крахе, фиаско), случившемся в процессе создания нового объекта. Рассмотрим пример кода (листинг 12.5), написанного, мягко говоря, некорректно, но именно это позволяет смоделировать ситуации, которые могут иметь место в реальной программе.

Листинг 12.5. Исключения в конструкторе

```
#include <iostream>
using namespace std;
class Vect {
public:
    Vect(char);
    ~Vect() { delete [] p; }
    int& operator [] (int i) { return p[i]; }
    void Print();
private:
    int* p;
    char size;
};
Vect::Vect(char n) : size(n) {
    p = new int[size]; // 1
    for (int i = 0; i < size; ++i) p[i] = int();
}
void Vect::Print() {
    for (int i = 0; i < size; ++i) cout << p[i] << " ";
    cout << endl;
}
int main() {
    int x = -1;
    cout << "x = " << hex << x << endl; // вывод в шестнадцатеричном
                                           // формате
    Vect a(3);
    a[0] = 0; a[1] = 1; a[2] = 2; a.Print();
    Vect a1(200); // 2
    a1[10] = 5; a1.Print();
}
```

В программе реализован класс Vect, предназначенный для создания и использования одномерных массивов типа int произвольного размера (размер передается через параметр конструктора класса).

Сначала несколько слов о возможной предыстории появления этого кода на свет. Руководитель проекта, для которого предназначен класс Vect, заявил, что размер массива никогда не превысит число 256. Проект (система) разрабатывается для «железа» с жуткими ограничениями на размер оперативной памяти. Программисты ведут борьбу за каждый байт! Человек, которому поручили реализацию класса Vect, решил хранить информацию о размере массива в поле char size. Ведь для типа

char выделяется один байт, а диапазон возможных значений для восьмиразрядного двоичного числа составляет 0 ... 255.

В функции main созданный класс тестируется. Предварительно листинг 12.5 должен быть откомпилирован в отладочной конфигурации (Debug) проекта. Если запустить программу под отладчиком (клавиша F5), то программа выводит

```
x = ffffffff
0 1 2
```

после чего серьезно «ломается» — среда выполнения сообщает о недопустимом размере запрашиваемой памяти: 4 294 967 295 байт.

Выполнение по шагам показывает, что крах происходит при создании объекта a1 (*оператор 2*). Если выполнить трассировку этого оператора, нажав клавишу F11, то можно достичь *оператора 1* в конструкторе Vect и увидеть, что значение size, задающее размер запрашиваемой памяти, равно -56.

Откуда такое странное значение? Обычно для начинающих программистов непросто выяснить причину... Наш герой, видимо, тоже был начинающим программистом. Более того, скорее всего, он невнимательно изучил Учебник [21], где на страницах 24–25 объясняется, что char – сокращенное обозначение типа signed char, в котором старший бит используется для представления знака числа. Поэтому диапазон представимых чисел для типа char составляет -128 ... +127.

Как же будет интерпретироваться в этом случае десятичное число 200? Преобразовав его в двоичный эквивалент, получаем 11001000. Вы видите, что старший разряд равен единице?... Это означает, что компьютер воспримет его как -1001000.

Обратный перевод в десятичную систему будет несколько сложнее. Напомним, что отрицательные целые числа хранятся в памяти компьютера в *дополнительном коде*, который получается инверсией всех разрядов с последующим прибавлением единицы. Значит, для обратного перевода нужно сначала вычесть единицу, а потом инвертировать все разряды. В итоге получим число -56.

Если теперь выполнить трассировку *оператора 1*, нажав клавишу F11, то отладчик остановится перед заголовком библиотечной реализации операции new:

```
void *__CRTDECL operator new[](size_t count)
```

а значение аргумента count будет равно уже не -56, а 4 294 967 295. Если перевести это десятичное число в шестнадцатеричный эквивалент, получим fff ffff. Для типа size_t, эквивалентного типу unsigned int, это максимально возможное значение 32-разрядного двоичного числа. Для типа signed int это значение представляет -1 (см. результат вывода программы для переменной x).

Далее операция new завершается аварийно, будучи не в состоянии выделить 4 294 967 295 байт.

Итак, мы нашли ошибку в реализации класса. Исправить ошибку несложно — нужно все char заменить на unsigned char (проверьте это). Но не будем спешить. Дело в том, что мы имеем прекрасную модель ненормативного поведения операции new и для дальнейших экспериментов она нам еще пригодится.

Приведенный пример показал, что, если вызываемые в теле конструктора класса функции или операции могут завершиться аварийно, необходимо информировать об этом клиента (функцию, в теле которой создается объект класса). Использование механизма исключений дает элегантное решение проблемы. Изменим код конструктора класса на следующий:

```
Vect::Vect(char n) : size( n ) {
    try { p = new int[size]; }
    catch(...) {
        throw "ErrorVectConstr";
        for ( int i = 0; i < size; ++i ) p[i] = int();
    }
}
```

Внесите также изменения в функцию main, использующую объекты класса Vect:

```
int main() {
    try {
        int x = -1;
        cout << "x = " << hex << x << endl;
        Vect a(3);
        a[0] = 0; a[1] = 1; a[2] = 2; a.Print();
        Vect al(200);
        al[10] = 5; al.Print();
    }
    catch( const char* except_name ) {      cout << "Error: " << except_name <<
endl; }
}
```

Для проверки работы новой версии программы необходимо откомпилировать листинг 12.5 в выпускной конфигурации (Release) проекта. Дело в том, что по непонятным причинам операция new в среде Visual Studio 2005 генерирует исключения только в выпускной конфигурации проекта.

Исключения в деструкторах

Если деструктор, вызванный во время раскрутки стека, попытается завершить свою работу при помощи исключения, система вызовет функцию terminate. На этапе отладки программы это допустимо, но в готовом продукте появление окон сообщений об ошибках должно быть сведено к минимуму.

Отсюда наиважнейшее требование к деструктору: *ни одно из исключений, которое могло бы появиться в процессе работы деструктора, не должно покинуть его пределы!* Чтобы выполнить это требование, придерживайтесь двух правил.

1. Никогда не генерируйте исключения в теле деструктора с помощью throw.
2. Если финальные действия в деструкторе связаны с вызовом других функций, относительно которых у вас нет гарантий отсутствия генерации исключений, инкапсулируйте эти действия в одном методе, например Destroy, и вызывайте его в блоке try:

```
T::~Destroy() {
    // код, который может генерировать исключения
}
```

```
T::~~T() {                                     // деструктор
    try { Destroy(); }
    catch(...) { /* ... */ }
}
```

Мы завершили рассмотрение вопросов, связанных с обработкой исключений. Рекомендуем вам доработать листинг 12.4, предусмотрев в реализации операции индексирования генерацию исключения для ошибочного значения индекса.

Теперь рассмотрим решение задачи, в которой требуется использовать и шаблонные классы, и обработку исключений.

Задача 12.2. Шаблонный класс векторов

Разработать шаблонный класс `Vect` для представления динамических одномерных массивов (векторов). Класс должен обеспечивать хранение данных любого типа `T`, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать:

- ☐ конструктор по умолчанию, создающий вектор нулевого размера;
- ☐ конструктор, создающий вектор заданного размера;
- ☐ операцию индексирования, возвращающую ссылку на соответствующий элемент вектора;
- ☐ метод, добавляющий элемент в произвольную позицию вектора;
- ☐ метод, добавляющий элемент в конец вектора;
- ☐ метод, удаляющий элемент из конца вектора.

При необходимости добавить в класс другие методы. Предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. В клиенте `main` продемонстрировать использование этого класса.

Одним из принципиальных вопросов при разработке контейнерного класса является вопрос реализации хранения элементов контейнера, или, другими словами, выбор подходящей структуры данных (массив, список, бинарное дерево и т. п.).

В данном случае мы остановим наш выбор на динамическом массиве, размещаемом в памяти посредством операции `new`, поскольку это наиболее простое решение. После размещения адрес первого элемента вектора будет запоминаться в поле `T* first`, а адрес элемента, следующего за последним, — в поле `T* last`. С учетом того, что используемый в классе метод `size` возвращает количество элементов в векторе, размещение контейнера в памяти поясняется на рис. 12.1.

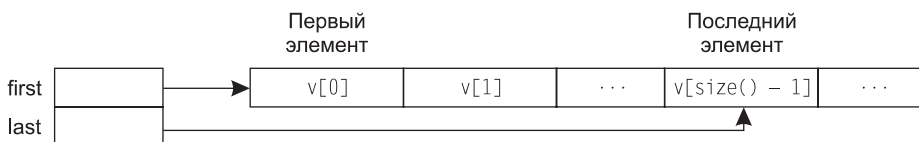


Рис. 12.1. Размещение вектора `v` в памяти

При решении этой задачи мы постараемся сделать класс `Vect` как можно более похожим (с точки зрения его интерфейса и, частично, реализации) на контейнерный класс `std::vector` из библиотеки STL (с учетом ограничений, вытекающих из постановки задачи). Поэтому в состав интерфейса будут включены методы:

- ❑ `void insert(T* _P, const T& _x)` — вставка элемента в позицию `_P`;
- ❑ `void push_back(const T& _x)` — вставка элемента в конец вектора;
- ❑ `void pop_back()` — удаление элемента из конца вектора;
- ❑ `T* begin()` — получение указателя на первый элемент;
- ❑ `T* end()` — получение указателя на элемент, следующий за последним;
- ❑ `size_t size()` — получение размера вектора (тип `std::size_t` является синонимом типа `unsigned int`).

Эти методы имитируют интерфейс класса `std::vector`. Уточним, что вместо типа `iterator`, имеющегося в `std::vector`, здесь используется указатель `T*`, но концептуально это одно и то же. Целью такого подражания является психологическая подготовка наших читателей к более легкому восприятию контейнерных классов STL (семинар 15).

К двум конструкторам, требующимся по заданию, добавим конструктор копирования, чтобы обеспечить возможность передачи объектов класса в качестве аргументов для любой внешней функции.

По некоторым параметрам наш класс `Vect`, однако, будет превосходить класс `std::vector`. Для целей отладки и обучения мы хотим визуализировать работу таких методов класса, как конструктор копирования и деструктор, путем вывода на терминал соответствующих сообщений. Чтобы эти сообщения были более информативны, мы снабдим каждый объект класса так называемым отладочным именем (поле `markName`), которое позволит распознавать источник сообщения.

Трудно переоценить, какую пользу приносят эти сообщения начинающим программистам, помогая прочувствовать скрытые механизмы функционирования класса. Для более опытных читателей эти средства могут оказаться полезными при поиске неочевидных ошибок в программе.

Задание требует также *предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций. Наиболее очевидными являются: ошибка индексирования, когда клиент пытается получить доступ к несуществующему элементу вектора, и ошибка удаления несуществующего элемента из конца вектора — когда вектор пуст.* В принципе возможна еще одна нештатная ситуация, связанная с нехваткой памяти, когда операция `new` завершается аварийно. Но мы рассмотрели выше, как решать эту проблему, и сейчас ради краткости изложения будем полагать, что обладаем неограниченной памятью.

Для обработки исключительных ситуаций создадим иерархию классов во главе с базовым классом `VectError`. Производный класс `VectRangeErr` будет предназначен для обработки ошибок индексирования, а производный класс `VectPopErr` — для обработки ошибки удаления несуществующего элемента. В листинге 12.6 приводится код предлагаемого решения задачи.

Листинг 12.6. Шаботонный класс векторов

```

//////////////////////////////////// VectError.h //////////////////////////////////
#ifndef _VECT_ERROR_
#define _VECT_ERROR_
#include <iostream>
#define DEBUG
class VectError {
public:
    VectError() {}
    virtual void ErrMsg() const { std::cerr << "Error with Vect object.\n"; }
    void Continue() const {
#ifdef DEBUG
        std::cerr << "Debug: program is being continued.\n";
#else
        throw;
#endif
    }
};
class VectRangeErr : public VectError {
public:
    VectRangeErr(int _min, int _max, int _actual) :
        min(_min), max(_max), actual(_actual) {}
    void ErrMsg() const {
        std::cerr << "Error of index: ";
        std::cerr << "possible range: " << min << " - " << max << ", ";
        std::cerr << "actual index: " << actual << std::endl;
        Continue();
    }
private:
    int min, max;
    int actual;
};
class VectPopErr : public VectError {
public:
    void ErrMsg() const { std::cerr << "Error of pop\n"; Continue(); }
};
#endif /* _VECT_ERROR_ */
//////////////////////////////////// Vect.h //////////////////////////////////
#ifndef _VECT_
#define _VECT_
#include <iostream>
#include <string>
#include "VectError.h"
template<class T> class Vect {      // ----- Template class Vect
public:
    explicit Vect() : first(0), last(0) {}
    explicit Vect(size_t _n, const T& _v = T()) {
        Allocate(_n);
        for (size_t i = 0; i < _n; ++i) *(first + i) = _v;
    }

```

продолжение ➤

Листинг 12.6 (продолжение)

```

    Vect( const Vect& );                                // конструктор копирования
    Vect& operator =(const Vect&);                      // операция присваивания
    ~Vect() {
#ifdef DEBUG
        cout << "Destructor of " << markName << endl;
#endif
        Destroy();
        first = 0, last = 0;
    }
    void mark(std::string& name) { markName = name; } // установить отладочное имя
    std::string mark() const { return markName; }     // получить отладочное имя
    size_t size() const;                               // получить размер вектора
    T* begin() const { return first; }                // получить указатель на 1-й элемент
    T* end() const { return last; }                   // получить указатель на элемент,
                                                    // следующий за последним
    T& operator[](size_t i);                           // операция индексирования
    void insert(T* _P, const T& _x);                   // вставка элемента в позицию _P
    void push_back(const T& _x);                       // вставка элемента в конец вектора
    void pop_back();                                   // удаление элемента из конца вектора
    void show() const;                                 // вывод в cout содержимого вектора
protected:
    void Allocate( size_t _n ) {
        first = new T [_n * sizeof(T)];
        last = first + _n;
    }
    void Destroy() {
        for ( T* p = first; p != last; ++p ) p->~T();
        delete [] first;
    }
    T* first;                                           // указатель на 1-й элемент
    T* last;                                           // указатель на элемент, следующий за последним
    std::string markName;
};
template<class T>
Vect<T>::Vect(const Vect& other) { // ----- Конструктор копирования
    size_t n = other.size();
    Allocate( n );
    for (size_t i = 0; i < n; ++i) *(first + i) = *(other.first + i);
    markName = string("Copy of ") + other.markName;
#ifdef DEBUG
    cout << "Copy constructor: " << markName << endl;
#endif
}
template<class T>
Vect<T>& Vect<T>::operator =(const Vect& other) { // ----- Операция присваивания
    if (this == &other) return *this;
    Destroy();
    size_t n = other.size();

```

```

    Allocate(n);
    for (size_t i = 0; i < n; ++i) *(first + i) = *(other.first + i);
    return *this;
}
template<class T>
size_t Vect<T>::size() const { // ----- Получение размера вектора
    if (first > last) throw VectError();
    return (0 == first ? 0 : last - first);
}
template<class T>
T& Vect<T>::operator[](size_t i) { // ----- Операция доступа по индексу
    if(i < 0 || i > ( size() - 1 ))
        throw VectRangeErr(0, last - first - 1, i);
    return ( *( first + i ) );
}
template<class T> // ----- Вставка элемента со значением _x в позицию _P
void Vect<T>::insert(T* _P, const T& _x) {
    size_t n = size() + 1; // новый размер
    T* new_first = new T [n * sizeof(T)];
    T* new_last = new_first + n;
    size_t offset = _P - first;
    for (size_t i = 0; i < offset; ++i) *(new_first + i) = *(first + i);
    *( new_first + offset ) = _x;
    for (size_t i = offset; i < n; ++i) *(new_first + i + 1) = *(first + i);
    Destroy();
    first = new_first;
    last = new_last;
}
template<class T>
void Vect<T>::push_back(const T& _x) { // ----- Вставка элемента в конец вектора
    if (!size()) { Allocate(1); *first = _x; }
    else insert( end(), _x );
}
template<class T>
void Vect<T>::pop_back() { // ----- Удаление элемента из конца вектора
    if(last == first) throw VectPopErr();
    T* p = end() - 1;
    p->~T();
    last--;
}
template<class T>
void Vect<T>::show() const { // ----- Вывод в cout содержимого вектора
    cout << "\n==== Contents of " << markName << "====" << endl;
    size_t n = size();
    for (size_t i = 0; i < n; ++i) cout << *(first + i) << " ";
    cout << endl;
}
#endif /* _VECT */
///////// Main.cpp //////////
#include "Vect.h"

```

продолжение ➔

Листинг 12.6 (продолжение)

```

#include <iostream>
#include <string>
using namespace std;
template<class T> void SomeFunction( Vect<T> v ) {
    std::cout << "Reversive output for " << v.mark() << ":" << endl;
    size_t n = v.size();
    for (int i = n - 1; i >= 0; --i) std::cout << v[i] << " ";
    std::cout << endl;
}
int main() {
    try {
        string initStr[5] = { "first", "second", "third", "fourth", "fifth" };
        Vect<int> v1(10); v1.mark(string("v1"));
        size_t n = v1.size();
        for (int i = 0; i < n; ++i) v1[i] = i + 1;
        v1.show();
        SomeFunction(v1);
        try {
            Vect<string> v2(5);
            v2.mark(string("v2"));
            size_t n = v2.size();
            for (int i = 0; i < n; ++i) v2[i] = initStr[i];
            v2.show();
            v2.insert(v2.begin() + 3, "After_third");
            v2.show();
            cout << v2[6] << endl;
            v2.push_back("Add_1"); v2.push_back("Add_2"); v2.push_back("Add_3");
            v2.show();
            v2.pop_back(); v2.pop_back();
            v2.show();
        }
        catch (VectError& vre) { vre.ErrMsg(); }
        try {
            Vect<int> v3;
            v3.mark(string("v3"));
            v3.push_back(41); v3.push_back(42); v3.push_back(43);
            v3.show();
            Vect<int> v4;
            v4.mark(string("v4"));
            v4 = v3;
            v4.show();
            v3.pop_back(); v3.pop_back();
            v3.pop_back(); v3.pop_back();
            v3.show();
        }
        catch (VectError& vre) { vre.ErrMsg(); }
    }
    catch (...) { cerr << "Epilogue: error of Main().\n"; }
}

```

Наши комментарии будут следовать в порядке размещения текста модулей.

Модуль `VectError.h` содержит иерархию классов исключений. В базовом классе `VectError` определены два метода. Виртуальный метод `ErrMsg` обеспечивает вывод по умолчанию сообщения об ошибке (в производных классах этот метод замещается конкретными методами). Метод `Continue` определяет стратегию продолжения работы программы после обнаружения ошибки. Стратегия зависит от конфигурации программы. Информация о конфигурации кодируется наличием или отсутствием единственной директивы в начале файла: `#define DEBUG`.

Если лексема `DEBUG` определена, программа компилируется в *отладочной конфигурации*, если не определена — в *выпускной конфигурации*. Для метода `Continue` это означает, что

- ❑ в отладочной конфигурации выводится сообщение «Debug: program is being continued», после чего работа программы продолжается;
- ❑ в выпускной конфигурации повторно возбуждается (оператор `throw`) исключение, которое было первопричиной цепочки событий, завершившихся вызовом данного метода.

В производных классах `VectRangeErr` и `VectPopErr` метод `ErrMsg` переопределяется с учетом вывода информации о конкретной ошибке. После вывода вызывается метод `Continue` базового класса.

Модуль `Vect.h` содержит определение шаблонного класса `Vect`. Отметим наиболее интересные моменты.

Для управления выделением и освобождением ресурсов класс снабжен методами `Allocate` и `Destroy`. Они размещены в защищенной части класса, так как относятся к его реализации. Напомним, что операция `new T [n]` (n — количество элементов, которое мы хотим поместить в массив) не только выделяет память, но и инициализирует элементы путем вызова `T()` — конструктора по умолчанию для типа `T`. Поэтому в методе `Destroy` сначала в цикле вызываются деструкторы `~T()` всех элементов массива, а потом операцией `delete` освобождается память.

Обратите внимание, что в заголовке цикла `for` условие его продолжения записано в виде `p != last`, а не `p < last`. Здесь мы следуем традиции STL. Подобный способ проверки обусловлен тем, что для произвольной структуры данных (например, для списка) соседние элементы не обязаны занимать подряд идущие ячейки памяти, а могут быть разбросаны в памяти как угодно.

Деструктор и конструктор копирования содержат вывод сообщений типа «Здесь был я!», дополненных печатью содержимого `markName`. Весь этот вывод осуществляется *только в отладочной конфигурации* программы. Операция присваивания (верная спутница конструктора копирования) реализована стандартным способом.

В операции доступа по индексу `operator[]()` служба внутренней охраны (оператор `if`) проверяет, не несет ли индекс с собой что-либо взрывоопасное (значение, лежащее вне пределов допустимого диапазона). Если индекс оказывается неблагонадежным, служба бьет тревогу, вызывая исключение типа `VectRangeErr`. Вызов происходит через анонимный экземпляр класса с передачей конструктору трех аргументов: минимальной и максимальной границы и текущего значения индекса.

В методе получения размера вектора `size` предусмотрена проверка на лояльность взаимоотношений между `first` и `last`. Вообще-то непонятно, с каких это дел `first` может оказаться больше, чем `last`? Да, в нормально функционирующем классе это невозможно. Но в случае каких-либо ошибок или сбоях указанная проверка весьма полезна. При несоблюдении указанного условия генерируется исключение типа `VectError`.

Метод `insert` (вставка элемента со значением `_x` в произвольную позицию `_P`) работает следующим образом. Создается новый вектор размером на единицу больше существующего вектора; адреса размещения нового вектора сохраняются в локальных переменных `new_first`, `new_last`. Затем в новый вектор копируется первая часть существующего вектора, начиная с первого элемента и заканчивая элементом, предшествующим элементу с адресом `_P`.

Поскольку доступ к элементам осуществляется через смещение относительно указателя `first`, перед копированием требуется вычислить `offset` — смещение, соответствующее адресу `_P`¹. Затем элементу с адресом `new_first + offset` присваивается значение `_x`. После этого оставшаяся часть существующего вектора копируется в новый вектор, начиная с позиции `new_first + offset + 1`. Вот и все. Осталось освободить ресурсы, занятые существующим вектором (`Destroy`), и назначить полям `first` и `last` новые значения.

Метод `push_back` (вставка элемента со значением `_x` в конец вектора) получился очень простым благодаря использованию серверной функции `insert`. Если вектор пуст, то выделяются ресурсы для хранения одного элемента и этому элементу присваивается значение `_x`. В противном случае вызывается `insert` для вставки элемента в позицию, определенную с помощью `end`.

В методе `pop_back` (удаление элемента из конца вектора) проверяется, не пуст ли вектор. В пустом векторе `first = last = 0`. Если это так, удалять нечего и, значит, клиент ошибся, затребовав такую операцию. Следовательно, надо ему об этом сообщить: генерируется исключение типа `VectPopErr`. В противном случае определяется указатель на последний элемент вектора, для него вызывается деструктор `~T()` и значение `last` уменьшается на единицу (учитывая арифметику указателей).

Модуль `Main.cpp` содержит определение глобальной шаблонной функции `SomeFunction` и определение функции `main`.

Функция `SomeFunction` обеспечивает реверсивный вывод содержимого вектора в поток `cout`. Попутно она позволяет проверить передачу объектов конкретного класса `Vect` в качестве аргументов функции.

Функция `main` предназначена для тестирования класса `Vect`. Советуем внимательно изучить ее многослойную архитектуру: внешний блок `try` содержит в себе два внутренних блока `try`. И это не случайно! Первый внутренний блок `try` предназначен для тестирования операций с объектом `v2`, среди которых есть операция,

¹ Напомним, что разность двух указателей — это разность их значений, деленная на размер типа в байтах. Например, в применении к массивам разность указателей на второй и седьмой элементы равна 5.

вызывающая ошибку индексирования. Второй блок `try` работает с объектом `v3`, причем одна из операций вызывает ошибку удаления несуществующего элемента.

Обратите внимание, что обработчики `catch` в обоих случаях имеют параметром объект базового класса `VectError`, но благодаря полиморфизму они будут перехватывать и исключения производных классов `VectRangeErr` и `VectPopErr`, так что в результате будет вызываться конкретный метод `ErrMsg` для данного типа ошибки! Вспомним, что после вывода сообщения об ошибке метод `ErrMsg` вызывает метод `Continue`, работа которого зависит от конфигурации программы.

В отладочной конфигурации `Continue` сообщает о продолжении работы программы и возвращает управление клиенту. В выпускной конфигурации `Continue` возбуждает исключение повторно. Именно для того, чтобы его поймать, и служит внешний блок `try` с обработчиком `catch(...)` в основной программе.

Откомпилируйте и выполните программу, чтобы увидеть, как она работает. Исходный текст настроен на работу в отладочной конфигурации. Вспомните внимательно в отладочные сообщения конструктора копирования и деструктора. А теперь закомментируйте директиву `#define DEBUG` и выполните программу после компиляции заново. Почувствуйте разницу!

Итоги

1. Шаблоны классов поддерживают парадигму обобщенного программирования — программирования с использованием типов в качестве параметров. Определение конкретного класса по шаблону класса и аргументам шаблона называется инстанцированием (актуализацией) шаблона.
2. Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов.
3. Для настройки шаблонного класса на некоторую стратегию возможно использование классов функциональных объектов.
4. Обработка исключительных ситуаций позволяет разделить проблему обработки ошибок на две фазы: генерацию исключения в случае нарушения каких-то заданных условий и последующую обработку сгенерированного исключения.
5. Фазы генерации и обработки исключений обычно разнесены в программе по разным компонентам (модулям, функциям). Это дает существенные преимущества, так как в месте возникновения ошибки (серверная функция) часто нет возможности корректно ее обработать, а клиентская функция такими возможностями обычно располагает.
6. Для конструктора класса использование исключений является единственным способом сообщить клиенту об ошибках или сбоях, случившихся в процессе конструирования объекта.
7. Для работы с исключениями необходимо: а) описать контролируемый блок; б) предусмотреть генерацию одного или нескольких исключений внутри блока или внутри функций, вызываемых из этого блока; в) разместить сразу за блоком один или несколько обработчиков исключений.

- 8. В качестве типов исключений, генерируемых оператором `throw`, целесообразно использовать определенные программистом классы исключений. Эти классы полезно выстраивать в иерархию, что позволяет создавать программы, для которых модификация в связи с обработкой новых типов ошибок оказывается менее трудоемкой.
- 9. Не генерируйте исключения в теле деструктора. Если это могут сделать другие вызываемые функции, примите все меры, чтобы исключение не покинуло деструктор.

Задания

Требуется создать шаблон некоторого целевого класса A. В каждом варианте уточняются требования к реализации — указанием на применение некоторого серверного класса B. Это означает, что объект класса B используется как элемент класса A. В качестве серверного класса может быть указан либо класс, созданный программистом в рамках того же задания, либо класс стандартной библиотеки.

Варианты целевых и серверных классов, создаваемых программистом, приведены в табл. 12.1. Информацию о работе с динамическими структурами данных см. в Учебнике (с. 114–127) и в семинаре 9.

Таблица 12.1. Варианты целевых или серверных классов

Имя класса	Назначение
Vect	Одномерный динамический массив
List	Двунаправленный список
Stack	Стек
BinaryTree	Бинарное дерево
Queue	Односторонняя очередь
Deque	Двусторонняя очередь (допускает вставку и удаление из обоих концов)
Set	Множество (повторяющиеся элементы в множество не заносятся; элементы в множестве хранятся отсортированными)
SparseArray	Разреженный массив

Если вместо серверного класса указан динамический массив, это означает, что для хранения элементов в целевом классе используется массив, размещаемый с помощью операции `new`. Во всех вариантах необходимо предусмотреть генерацию и обработку исключений для возможных ошибочных ситуаций.

Во всех вариантах показать в клиенте `main` использование созданного класса, включая ситуации, приводящие к генерации исключений. Показать инстанцирование шаблона для типов `int`, `double`, `std::string`.

Варианты заданий приведены в табл. 12.2.

Таблица 12.2. Варианты заданий

Вариант	Целевой шаблонный класс	Реализация с применением
1	Vect	std::list
2	List	—
3	Stack	Динамический массив
4	Stack	Vect
5	Stack	List
6	Stack	std::vector
7	Stack	std::list
8	BinaryTree	—
9	Queue	Vect
10	Queue	List
11	Queue	std::list
12	Deque	Vect
13	Deque	List
14	Deque	std::list
15	Set	Динамический массив
16	Set	Vect
17	Set	List
18	Set	std::vector
19	Set	std::list
20	SparseArray	List

Семинар 13. Стандартные потоки

Теоретический материал: с. 265–280.

Потоковые классы

Программа на C++ представляет ввод и вывод как поток байтов. При вводе она читает байты из потока ввода, при выводе вставляет байты в поток вывода. Понятие потока позволяет абстрагироваться от того, с каким устройством ввода-вывода идет работа. Например, байты потока ввода могут поступать либо с клавиатуры, либо из файла на диске, либо из другой программы. Аналогично, байты потока вывода могут выводиться на экран, в файл на диске или на вход другой программы.

Обычно ввод-вывод осуществляется через *буфер* — область оперативной памяти, накапливающую какое-то количество байтов, прежде чем они пересылаются по назначению. При обмене, например, с диском это значительно повышает скорость передачи информации. При вводе с клавиатуры буферизация обеспечивает другое удобство для пользователя — возможность исправления ошибок набора символов, пока они не отправлены из буфера в программу.

Буфер ввода обычно очищается при нажатии клавиши Enter. *Буфер вывода* на экран очищается либо при появлении в выходном потоке символа новой строки '\n', либо при выполнении программой очередной операции ввода с клавиатуры.

Язык C++ предоставляет возможности ввода-вывода на низком уровне (*неформатированный* ввод-вывод) и на высоком уровне (*форматированный* ввод-вывод). В первом случае передача информации выполняется блоками байтов без какого-либо преобразования данных. Во втором случае байты группируются для представления таких элементов данных, как целые и вещественные числа, строки символов и так далее.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов — `ios` и `streambuf`. Класс `ios` содержит базовые средства управления потоками, являясь родительским для других классов ввода-вывода. Класс `streambuf` обеспечивает общие средства управления буферами потоков и их взаимодействие с физическими устройствами, являясь родительским для других буферных классов. Связь между этими двумя классами реализует поле `bp` в классе `ios`, являющееся указателем на `streambuf`.

Классы стандартных потоков

Потоки, связанные с консольным вводом-выводом (клавиатура–экран), называются *стандартными*. Стандартному потоку ввода соответствует класс `istream`, стандартному потоку вывода — класс `ostream`. Оба класса являются потомками класса `ios`. Следующим в иерархии является класс `iostream`, наследующий классы `istream` и `ostream` и обеспечивающий общие средства потокового ввода-вывода.

Заголовочные файлы библиотеки ввода-вывода C++

Заголовочные файлы стандартной библиотеки C++ по стандарту указываются без расширения, например `<iostream>`. Все имена стандартной библиотеки принадлежат пространству имен `std`. Для упрощения доступа к именам в программах часто используется директива `using namespace std;`.

Объекты и методы стандартных потоков ввода-вывода

По директиве `#include <iostream>` становятся доступными объекты:

- `cin` — объект класса `istream`, соответствующий стандартному потоку ввода;
- `cout` — объект класса `ostream`, соответствующий стандартному потоку вывода.

Оба потока являются буферизованными. Благодаря объектам `cin` и `cout` становятся доступными и методы соответствующих классов.

Форматированный ввод-вывод реализуется через две перегруженные операции: операция сдвига влево `'<<'` перегружена для вывода в поток и называется *операцией вставки (вывода)* в поток; операция сдвига вправо `'>>'` перегружена для ввода из потока и называется *операцией извлечения (ввода)* из потока. Например, в классе `ostream` определены операции

```
ostream& operator<<( short );  
ostream& operator<<( int );  
ostream& operator<<( double );           // и так далее
```

Рассмотрим, что произойдет, если в программе встретится оператор

```
cout << x;                               // ранее в программе:   int x = 5;
```

Сначала компилятор определит, что левый аргумент имеет тип `ostream`, а правый — тип `int`. Далее он найдет в заголовочном файле `ostream` прототип метода `ostream& operator<<(int)`. В конечном итоге будет вызван метод `cout.operator<<(x)`, в результате выполнения которого мы увидим на экране 5.

Заметим, что в случае вывода значений встроенных типов, таких как `int` или `double`, класс `ostream` обеспечивает их преобразование из внутреннего двоичного представления в строку символов. О форматировании можно не заботиться, так как класс `ostream` поддерживает форматирование вывода по умолчанию, задаваемое соответствующими полями базового класса `ios`.

Аналогично происходит работа со стандартным потоком ввода через объект `cin`, но направление движения информации противоположное: из потока — в программу.

Если форматирование по умолчанию вас не устраивает, можно воспользоваться либо соответствующими *методами* класса `ios`, либо *манипуляторами* ввода-вывода,

представляющими собой функции, которые можно включать прямо в поток. Они определены частично в файлах `istream` и `ostream` и частично в файле `iomanip`. В табл. 13.1 приведены наиболее часто употребляемые манипуляторы¹.

Таблица 13.1. Основные манипуляторы и методы управления форматом

Манипулятор	Метод класса <code>ios</code>	Ввод	Вывод	Описание
<code>endl</code>	–		+	Включить в поток символ новой строки и выгрузить буфер
<code>dec</code>	<code>flags(ios::dec)</code>	+	+	Перейти в десятичную систему счисления
<code>hex</code>	<code>flags(ios::hex)</code>	+	+	Перейти в шестнадцатеричную систему счисления
<code>oct</code>	<code>flags(ios::oct)</code>	+	+	Перейти в восьмеричную систему счисления
<code>setprecision(n)</code>	<code>precision(n)</code>		+	Установить количество отображаемых знаков
<code>setw(n)</code>	<code>width(n)</code>		+	Установить ширину поля вывода
<code>setfill(c)</code>	<code>fill(c)</code>		+	Установить символ заполнения <code>c</code>

Пример вывода целой переменной `x` в шестнадцатеричной форме:

```
cout.flags( ios::hex ); cout << x;      // способ 1: использование метода
cout << hex << x;                       // способ 2: использование манипулятора
```

Остановимся немного подробнее на манипуляторе `setprecision(n)` и соответствующем методе `precision(n)`, поскольку их воздействие на формат вывода подчиняется довольно сложным правилам. И тот, и другой изменяют значение поля `ios::x_precision` (по умолчанию это значение равно шести). Данное поле управляет точностью вывода вещественных чисел, причем его интерпретация зависит от значений других полей, управляющих форматом вывода:

- ❑ `ios::scientific` – «научный» формат (с плавающей точкой);
- ❑ `ios::fixed` – формат вывода с фиксированной точкой.

Если установлен хотя бы один из этих форматов (например, с помощью метода `flags`), то `x_precision` задает количество цифр *после десятичной точки*. Если не установлен ни тот, ни другой и вывод идет в так называемом *автоматическом формате*, то значение `x_precision` задает *общее количество значащих цифр*.

Учтите, что большинство параметров форматирования сохраняют свое состояние вплоть до следующего вызова функции, изменяющей это состояние. Исключение – манипулятор `setw(n)` и соответствующий ему метод `width(n)`: их действие распространяется только на ближайшую операцию вывода, после чего восстанавливается ширина поля вывода по умолчанию.

¹ Полный список манипуляторов и методов форматирования см. в Учебнике, с. 271.

По признаку наличия аргумента манипуляторы подразделяются на *простые* (без аргумента) и *параметризованные* (с аргументом). Для использования последних необходимо подключить заголовочный файл `iosmanip`.

С точки зрения *реализации* манипуляторы можно разделить на три группы:

- 1) манипуляторы без аргумента, выводящие в поток управляющий символ (`endl`, `ends`) или очищающие буфер потока (`flush`);
- 2) манипуляторы без аргумента, изменяющие значения полей базового класса `ios`, задающих текущую систему счисления (`dec`, `hex`, `oct`);
- 3) манипуляторы с аргументом.

Кроме рассмотренных, класс `ios` предоставляет и другие методы, обеспечивающие неформатированный ввод-вывод, а также связь с буфером потока. Наиболее часто употребляемые методы ввода-вывода приведены в табл. 13.2.

Таблица 13.2. Некоторые методы ввода-вывода класса `ios`

Метод	Описание
<code>get()</code>	Возвращает код извлеченного из потока символа или EOF, если достигнут конец файла
<code>get(c)</code>	Присваивает код извлеченного из потока символа аргументу <code>c</code>
<code>get(buf, num, lim='\n')</code>	Читает из потока символы, пока не встретится символ <code>lim</code> (по умолчанию <code>'\n'</code>) или пока не будет прочитано <code>num-1</code> символов. Извлеченные символы размещаются в символьный массив <code>buf</code> , к ним добавляется нулевой байт. Символ <code>lim</code> остается в потоке
<code>getline(buf, num, lim='\n')</code>	Выполняется аналогично предыдущему методу, но символ <code>lim</code> удаляется из потока (в массив <code>buf</code> он также не записывается)
<code>peek()</code>	Возвращает код следующего (готового для извлечения) символа в потоке, но не извлекает его; или EOF, если достигнут конец файла
<code>read(buf, num)</code>	Считывает <code>num</code> символов из потока в символьный массив <code>buf</code>
<code>gcount()</code>	Возвращает количество символов, прочитанных последним вызовом функции неформатированного ввода
<code>rddbuf()</code>	Возвращает указатель на буфер типа <code>streambuf</code> , связанный с данным потоком
<code>put(c)</code>	Выводит в поток символ <code>c</code>
<code>write(buf, num)</code>	Выводит в поток <code>num</code> символов из массива <code>buf</code>

Обработка ошибок потоков

Библиотека ввода-вывода C++ обеспечивает гораздо более надежный ввод-вывод, чем старые функции библиотеки C. Это достигается перегрузкой операций извлечения и вставки для всех встроенных типов, что исключает путаницу с типами, которая была возможна при использовании функций `scanf` и `printf` и которая приносила массу неприятностей программистам, поскольку компилятор никак не реагировал на ошибки в спецификации формата.

И все же проблема ошибок практически не волнует нас только при выводе информации в поток `cout`. При вводе никто не может запретить пользователю ввести вместо ожидаемого программой целого числа произвольную строку символов, и если не принять специальных мер, программа «сломается». Для отслеживания ошибок потоков в базовом классе `ios` определено поле `state`, отдельные биты (флаги) которого отображают состояние потока, как показано в табл. 13.3.

Таблица 13.3. Флаги состояния потока

Имя флага	Интерпретация
<code>ios::goodbit</code>	Нет ошибок
<code>ios::eofbit</code>	Достигнут конец файла
<code>ios::failbit</code>	Ошибка форматирования или преобразования
<code>ios::badbit</code>	Серьезная ошибка, после которой пользоваться потоком невозможно

Получить текущее состояние потока можно с помощью метода `rdstate`, который возвращает значение типа `int`. Есть и другие, более удобные для анализа методы:

- ❑ `int eof()` — возвращает ненулевое значение, если установлен флаг `eofbit`;
- ❑ `int fail()` — возвращает ненулевое значение, если случилась любая ошибка ввода-вывода (но не конец файла). Этому условию соответствует установка либо флага `badbit`, либо флага `failbit`;
- ❑ `int bad()` — возвращает ненулевое значение, если случилась серьезная ошибка ввода-вывода (то есть установлен флаг `badbit`);
- ❑ `int good()` — возвращает ненулевое значение, если сброшены все флаги ошибок.

Если произошла ошибка ввода, в результате которой установлен только флаг `failbit`, то после этого можно продолжать работу с потоком, но сначала нужно сбросить все флаги ошибок, для чего предназначен метод `void clear(int = 0)`. Для сброса флагов достаточно сделать вызов `clear()`, так как аргумент по умолчанию равен нулю. Этот же метод можно использовать для установки соответствующих флагов поля `state`, если передать ему ненулевой аргумент (обычно это комбинация флагов из табл. 13.3, объединенных операцией `|`).

Есть и другие приемы диагностирования ошибочных ситуаций. Мы используем их при решении задачи 13.1.

Перегрузка операций извлечения и вставки для типов, определенных программистом

Одно из удобств C++ — возможность перегрузить операции извлечения и вставки для любого класса `MyClass`, созданного программистом. После этого для любого объекта `obj` класса `MyClass` можно записывать операторы вида `cin >> obj; cout << obj`. Удобно, не правда ли?

Однако операции `>>` и `<<` не могут быть элементами класса `MyClass`. Причина в том, что у любой функции-операции класса левым операндом подразумевается объект этого класса, вызывающий данную функцию. Но для операций извлечения (вставки) левый операнд должен быть потоком ввода-вывода. Поэтому эти функции всегда объявляются *дружественными* классу, для которого они создаются.

Общая форма пользовательской *функции извлечения*:

```
istream& operator >>( istream& is, MyClass& obj )
```

где `is` – ссылка на входной поток, `obj` – ссылка на объект, принимающий ввод. В теле функции последний оператор должен возвращать объект `is`.

Общая форма пользовательской *функции вставки*:

```
ostream& operator <<( ostream& os, MyClass& obj )
```

где `os` – ссылка на выходной поток, а `obj` – ссылка на объект, к которому применяется операция. Последний оператор функции должен возвращать объект `os`. В принципе, возможна и другая сигнатура операции вставки с передачей второго аргумента по значению (`MyClass obj`). Но в этом случае, во-первых, при вызове функции в стеке будет создаваться копия объекта `obj`, а во-вторых, в классе `MyClass` должен быть предусмотрен конструктор копирования.

Рассмотрим теперь задачу, в которой используются стандартные средства ввода-вывода для потоков и, кроме этого, перегружаются операции извлечения и вставки для пользовательского типа данных.

Задача 13.1. Первичный ввод и поиск информации в базе данных

Написать программу, которая обеспечивает первичный ввод информации в базу данных отдела кадров предприятия численностью до 100 человек (без записи в файл) и поиск информации по заданному критерию. Каждая запись базы данных содержит следующие сведения о сотруднике: фамилия и инициалы; год поступления на работу; оклад. Критерий поиска: сотрудники с окладом, превышающим некоторую заданную величину.

Решение задачи начнем с выявления понятий (классов) и их фундаментальных взаимосвязей.

В данном случае первым понятием является «база данных», и, следовательно, для моделирования этого понятия понадобится класс, который мы назовем `DBase`. Объект типа `DBase` должен содержать некоторую совокупность, или коллекцию, других объектов, соответствующих записям базы данных. Для моделирования понятия «запись базы данных» введем класс `Man`. Очевидно, что взаимоотношение между указанными классами относится к типу «`DBase has a Man`».

На втором этапе необходимо уточнить классы, определив основные поля и набор операций над ними.

Начнем с класса `DBase`. Первый вопрос: какую структуру данных целесообразно использовать для хранения коллекции записей. Поскольку объем базы данных небольшой, выберем самое простое решение — массив объектов типа `Man`. Очевидно, что в конструкторе класса необходимо выполнить выделение памяти для требуемого количества объектов типа `Man`, а в деструкторе — освобождение этой памяти. Адрес начала массива объектов представим полем `Man* pMan`.

Ясно также, что в классе необходимо иметь метод `InitInput` для первичного ввода информации в базу данных и метод `SearchPayNotLess` для поиска сотрудников

с окладом, превышающим некоторую заданную величину. Для контроля правильности ввода исходных данных нам пригодится еще один метод — `Show`, выполняющий вывод на экран содержимого базы данных.

Теперь разберемся с классом `Man`. Для хранения информации об одном сотруднике потребуются следующие поля:

- ❑ `char* pName` — адрес строки, содержащей фамилию и инициалы;
- ❑ `int take_job_year` — год поступления на работу;
- ❑ `double pay` — величина оклада.

Конструктор класса должен выделять память для хранения указанной строки, а деструктор — освобождать эту память. Для решения второй подзадачи (поиск информации) добавим в класс метод доступа `GetPay`. И наконец, для класса `Man` нужно предусмотреть перегрузку операции извлечения, чтобы обеспечить первичный ввод информации с клавиатуры в методе `InitInput` класса `DBase`, и операцию вставки, которая будет использована в методе `Show` класса `DBase`. Обе операции должны быть реализованы как внешние дружественные функции.

Иногда при решении задачи удобно использовать внешние функции, не являющиеся элементами классов. Обычно они выполняют какую-то рутинную работу и могут быть вызваны как из методов классов, так и из основной функции.

Типичный пример — ввод значений из стандартного потока `cin` с защитой от непреднамеренных ошибок пользователя. Начнем с «наивной» реализации перегруженной операции `>>` для класса `Man`:

```
istream& operator >> ( istream& in, Man& obj ) {
// .....
    in >> obj.take_job_year;    // 1
    in >> obj.pay;
    return in;
}
```

Если при выполнении *оператора 1* пользователь введет вместо целого числа строку символов, программа «сломается». Вашему заказчику наверняка это не понравится. Такая же ситуация возможна и при вводе вещественного числа в следующем операторе. Для решения этих проблем в программе будут использованы функции `GetInt` и `GetDouble`, обеспечивающие надежный ввод целых и вещественных чисел соответственно. Так как эти функции универсальные и внеклассовые, их код целесообразно разместить в отдельном модуле.

Решение задачи, в котором реализованы рассмотренные концепции, представляет собой многофайловый проект, содержащий файлы `DBase.h`, `DBase.cpp`, `Man.h`, `Man.cpp`, `GetFunc.h`, `GetFunc.cpp` и `Main.cpp` (листинг 13.1).

Листинг 13.1. Многофайловый проект

```
////////////////////////////////////////// DBase.h
class DBase {
public:
    DBase(int);
    ~DBase();
```



```

        void InitInput();
        void Show();
        void SearchPayNotLess(double);
private:
    Man* pMan;
    int nRecords;
};
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// DBase.cpp
#include "Man.h"
#include "DBase.h"
DBase::DBase(int nRec) : nRecords(nRec),
pMan(new Man[nRec]) {}
DBase::~DBase() { if (pMan) delete [] pMan; }
void DBase::InitInput() {
    for ( int i = 0; i < nRecords; i++ ) cin >> *( pMan + i );           // 1
}
void DBase::Show() {
    cout << "=====" << endl;
    cout << "Содержимое базы данных:" << endl;
    for (int i = 0; i < nRecords; i++) cout << *( pMan + i );           // 2
}
void DBase::SearchPayNotLess(double anyPay) {
    bool not_found = true;
    for (int i = 0; i < nRecords; i++)
        if (( pMan + i )->GetPay() >= anyPay) {
            cout << *(pMan + i);
            not_found = false;
        }
    if (not_found) cout << "Таких сотрудников нет." << endl;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Man.h
#include <iostream>
#include <iomanip>
using namespace std;
const int l_name = 30;
class Man {
public:
    Man(int lName = 30);
    ~Man();
    double GetPay() const;
    friend istream& operator >>(istream&, Man&); // Операция извлечения (ввода)
    friend ostream& operator <<(ostream&, Man&); // Операция вставки (вывода)
private:
    char* pName;
    int take_job_year;
    double pay;
};
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Man.cpp
#include "windows.h"           // для работы в среде Windows с кириллицей
#include "Man.h"
#include "GetFunc.h"
Man::Man(int lName) { pName = new char[lName + 1]; }

```

продолжение ➤

Листинг 13.1 (продолжение)

```

Man::~Man() { if (pName) delete [] pName; }
double Man::GetPay() const { return pay; }
istream& operator >> (istream& in, Man& ob) { // ---- Операция извлечения (ввода)
    cout << "\nВведите данные в формате" << endl;
    cout << "Фамилия И.О. <Enter> Год поступления <Enter>";
    cout << "Оклад <Enter>:" << endl;
    in.getline(ob.pName, 1_name);
    OemToChar(ob.pName, ob.pName);    // для работы в среде Windows с кириллицей
    ob.take_job_year = GetInt(in);      // 3
    ob.pay = GetDouble(in);            // 4
    return in;
}
ostream& operator << (ostream& out, Man& ob) { // ---- Операция вставки (вывода)
    out << setw( 30 ) << setiosflags( ios::left );
    out << ob.pName << " ";
    out << ob.take_job_year << " ";
    out << ob.pay << endl;
    return out;
}
////////////////////////////////////////////////////////////////// GetFunc.h
int GetInt(istream&);           // Ввод целого числа
double GetDouble(istream&);     // Ввод вещественного числа
////////////////////////////////////////////////////////////////// GetFunc.cpp
#include "Man.h"
#include "GetFunc.h"
int GetInt(istream& in) { // ----- ввод целого числа
    int value;
    while ( true ) {
        in >> value;                // 5
        if (in.peek() == '\n') {    // 6
            in.get();                // 7
            break; }
        else {
            cout << "Повторите ввод (ожидается целое число):" << endl; // 8
            in.clear();              // 9
            while ( in.get() != '\n' ) {} ; // 10
        }
    }
    return value;
}
}
double GetDouble(istream& in) { // ----- ввод вещественного числа
    double value;
    while ( true ) {
        in >> value;
        if (in.peek() == '\n') { in.get(); break; }
        else {
            cout << "Повторите ввод (ожидается вещественное число):" << endl;
            in.clear();
        }
    }
}

```

```

        while ( in.get() != '\n' ) {}
    }
}
return value;
}
//////////////////////////////////// Main.cpp
#include "Man.h"
#include "GetFunc.h"
int main() {
    const int nRecord = 10;           // количество записей в базе данных
    double any_pay;
    setlocale(LC_ALL, "Russian");     // только для работы в среде Windows
    DBase dBase( nRecord );
    dBase.InitInput();
    dBase.Show();
    cout << "Ввод данных завершен." << endl;
    cout << "=====" << endl;
    cout << "Поиск сотрудников, чей оклад не меньше заданной величины." << endl;
    cout << "Поиск завершается при вводе -1." << endl;
    while ( true ) {
        cout << "\nВведите величину оклада или -1: ";
        any_pay = GetDouble(cin);
        if (any_pay == -1) break;
        dBase.SearchPayNotLess(any_pay);
    }
}

```

Обратите внимание на следующие моменты.

В реализации метода `InitInput` перегруженная для класса `Man` операция извлечения применяется к объекту, задаваемому выражением `*(pMan + i)`, то есть к объекту, адрес которого есть `pMan + i` (*оператор 1*).

Аналогичная адресация объекта для операции вставки использована в методе `Show` (*оператор 2*).

В реализации перегруженной операции извлечения (файл `Man.cpp`) обратите внимание на *операторы 3 и 4*, в которых вызываются функции `GetInt` и `GetDouble`, предназначенные для ввода из стандартного потока целых и вещественных чисел соответственно.

Реализация функций `GetInt` и `GetDouble` находится в файле `GetFunc.cpp`. Рассмотрим подробно первую из них. Ввод целого числа организован внутри бесконечного цикла `while` следующим образом.

Информация читается из входного потока *оператором 5*. Если в буфере типа `streambuf`, связанном с потоком `in`, находится изображение целого числа, завершаемое символом перевода строки `'\n'`, то по завершении операции извлечения в буфере останется только символ `'\n'`. *Оператор 6* проверяет это условие, используя метод `peek`. Если проверка завершилась успешно, *оператор 7* очищает входной буфер от символа `'\n'`, после чего происходит выход из цикла `while` с последующим

возвратом из функции значения `value`. Если же введенная информация не является корректным изображением целого числа, то выполняются три действия:

- ☐ оператор 8 выводит сообщение об этом, предлагая повторить ввод;
- ☐ сбрасываются флаги ошибок для потока `in` (оператор 9);
- ☐ с помощью внутреннего цикла `while` из входного буфера извлекаются все символы, вплоть до символа `'\n'` (оператор 10). Внешний цикл `while` повторяется сначала.

Функция `GetDouble` работает аналогично.

Функция `main` (файл `Main.cpp`) особых пояснений не требует. Алгоритм прозрачный и воспринимается легко благодаря выразительным именам методов.

В заключение отметим, что рассмотренная задача очень похожа на задачу 10.1. Мы советуем читателю сравнить решение этих двух задач, чтобы увидеть преимущества технологии ООП. Особенно впечатляет сравнение кодов функции `main`: положите рядом тексты листингов 10.1 и 13.1, и вы будете поражены совершенством, лаконичностью и красотой второго решения по сравнению с первым.

Итоги

1. Ввод и вывод представляются в программе как поток байтов и обычно выполняются через буфер. Потоки поддерживаются в библиотеке C++ с помощью иерархии классов, которая реализует безопасный ввод-вывод как стандартных, так и пользовательских типов данных.
2. Ввод-вывод бывает форматированный и неформатированный. Для форматированного ввода-вывода используются перегруженные операции `<<` и `>>`, для неформатированного — методы стандартных классов.
3. Управление форматированием выполняется с помощью манипуляторов и методов стандартных классов.
4. Для вывода объектов пользовательских типов данных следует с помощью дружественных функций перегрузить операции чтения и извлечения.
5. Для обеспечения безопасного ввода необходимо диагностировать возможные ошибки, используя флаги состояния потока и (или) методы `peek`, `get` и `clear`.

Задания

Вариант 1

1. Определить класс с именем `STUDENT`, содержащий следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов).
Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа `STUDENT`.
2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из десяти объектов типа `STUDENT`; записи должны быть упорядочены по возрастанию номера группы;

- вывод на дисплей фамилий и номеров групп для всех студентов, включенных в массив, если средний балл студента больше 4.0;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 2

1. Определить класс с именем STUDENT, содержащий следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по возрастанию среднего балла;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки 4 и 5;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 3

1. Определить класс с именем STUDENT, содержащий следующие поля: фамилия и инициалы; номер группы; успеваемость (массив из пяти элементов).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа STUDENT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из десяти объектов типа STUDENT; записи должны быть упорядочены по алфавиту;
- вывод на дисплей фамилий и номеров групп для всех студентов, имеющих хотя бы одну оценку 2;
- если таких студентов нет, вывести соответствующее сообщение.

Вариант 4

1. Определить класс с именем AEROFLOT, содержащий следующие поля: название пункта назначения рейса; номер рейса; тип самолета.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа AEROFLOT.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из семи объектов типа AEROFLOT; записи должны быть упорядочены по возрастанию номера рейса;
- вывод на экран номеров рейсов и типов самолетов, вылетающих в пункт назначения, название которого совпало с названием, введенным с клавиатуры;
- если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 5

1. Определить класс с именем AEROFLOT, содержащий следующие поля: название пункта назначения рейса; номер рейса; тип самолета.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа AEROFLOT.

2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из семи объектов типа AEROFLOT; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
 - вывод на экран пунктов назначения и номеров рейсов, обслуживаемых самолетом, тип которого введен с клавиатуры;
 - если таких рейсов нет, выдать на дисплей соответствующее сообщение.

Вариант 6

1. Определить класс с именем WORKER, содержащий следующие поля:

- фамилия и инициалы работника;
- название занимаемой должности;
- год поступления на работу.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа WORKER.

2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из десяти объектов типа WORKER; записи должны быть размещены по алфавиту;
 - вывод на дисплей фамилий работников, чей стаж работы в организации превышает значение, введенное с клавиатуры;
 - если таких работников нет, вывести на дисплей соответствующее сообщение.

Вариант 7

1. Определить класс с именем TRAIN, содержащий следующие поля: название пункта назначения; номер поезда; время отправления.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа TRAIN.

2. Написать программу, выполняющую следующие действия:
 - ввод с клавиатуры данных в массив, состоящий из восьми объектов типа TRAIN; записи должны быть размещены в алфавитном порядке по названиям пунктов назначения;
 - вывод на экран информации о поездах, отправляющихся после введенного с клавиатуры времени;
 - если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 8

1. Определить класс с именем `TRAIN`, содержащий следующие поля: название пункта назначения; номер поезда; время отправления.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа `TRAIN`.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из шести объектов типа `TRAIN`; записи должны быть упорядочены по времени отправления поезда;
- вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры;
- если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 9

1. Определить класс с именем `TRAIN`, содержащий следующие поля: название пункта назначения; номер поезда; время отправления.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа `TRAIN`.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа `TRAIN`; записи должны быть упорядочены по номерам поездов;
- вывод на экран информации о поезде, номер которого введен с клавиатуры;
- если таких поездов нет, выдать на дисплей соответствующее сообщение.

Вариант 10

1. Определить класс с именем `MARSH`, содержащий следующие поля:

- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа `MARSH`.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа `MARSH`; записи должны быть упорядочены по номерам маршрутов;
- вывод на экран информации о маршруте, номер которого введен с клавиатуры;
- если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 11

1. Определить класс с именем `MARSH`, содержащий следующие поля:

- название начального пункта маршрута;
- название конечного пункта маршрута;
- номер маршрута.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа MARSH.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа MARSH; записи должны быть упорядочены по номерам маршрутов;
- вывод на экран информации о маршрутах, которые начинаются или кончаются в пункте, название которого введено с клавиатуры;
- если таких маршрутов нет, выдать на дисплей соответствующее сообщение.

Вариант 12

1. Определить класс с именем NOTE, содержащий следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о человеке, номер телефона которого введен с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 13

1. Определить класс с именем NOTE, содержащий следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть размещены по алфавиту;
- вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 14

1. Определить класс с именем NOTE, содержащий следующие поля: фамилия, имя; номер телефона; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа NOTE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа NOTE; записи должны быть упорядочены по трем первым цифрам номера телефона;

- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 15

1. Определить класс с именем ZNAK, содержащий следующие поля: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о человеке, чья фамилия введена с клавиатуры;
- если такого нет, выдать на дисплей соответствующее сообщение.

Вариант 16

1. Определить класс с именем ZNAK, содержащий следующие поля: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по датам дней рождения;
- вывод на экран информации о людях, родившихся под знаком, наименование которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 17

1. Определить класс с именем ZNAK, содержащий следующие поля: фамилия, имя; знак Зодиака; день рождения (массив из трех чисел).

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ZNAK.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ZNAK; записи должны быть упорядочены по знакам Зодиака;
- вывод на экран информации о людях, родившихся в месяц, значение которого введено с клавиатуры;
- если таких нет, выдать на дисплей соответствующее сообщение.

Вариант 18

1. Определить класс с именем PRICE, содержащий следующие поля:

- название товара;

- название магазина, в котором продается товар;
- стоимость товара в руб.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа PRICE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям товаров;
- вывод на экран информации о товаре, название которого введено с клавиатуры;
- если таких товаров нет, выдать на дисплей соответствующее сообщение.

Вариант 19

1. Определить класс с именем PRICE, содержащий следующие поля:

- название товара;
- название магазина, в котором продается товар;
- стоимость товара в руб.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа PRICE.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа PRICE; записи должны быть размещены в алфавитном порядке по названиям магазинов;
- вывод на экран информации о товарах, продающихся в магазине, название которого введено с клавиатуры;
- если такого магазина нет, выдать на дисплей соответствующее сообщение.

Вариант 20

1. Определить класс с именем ORDER, содержащий следующие поля: расчетный счет плательщика; расчетный счет получателя; перечисляемая сумма.

Определить методы доступа к этим полям и перегруженные операции извлечения и вставки для объектов типа ORDER.

2. Написать программу, выполняющую следующие действия:

- ввод с клавиатуры данных в массив, состоящий из восьми объектов типа ORDER; записи должны быть размещены в алфавитном порядке по расчетным счетам плательщиков;
- вывод на экран информации о сумме, снятой с расчетного счета плательщика, введенного с клавиатуры;
- если такого расчетного счета нет, выдать на дисплей соответствующее сообщение.

Семинар 14. Файловые и строковые потоки.

Строки класса string

Теоретический материал: с. 280–294.

Файловые потоки

Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы, указанные в табл. 14.1.

Таблица 14.1. Классы файловых потоков

Класс	Инстанцирован из шаблона	Базовый шаблонный класс	Назначение
ifstream	basic_ifstream	basic_istream	Входной файловый поток
ofstream	basic_ofstream	basic_ostream	Выходной файловый поток
fstream	basic_fstream	basic_iostream	Двунаправленный файловый поток

Так как классы файловых потоков являются производными от классов `istream`, `ostream` и `iostream` соответственно, то они наследуют все методы указанных классов, перегруженные операции вставки и извлечения, манипуляторы, состояние потоков и т.д. Как и стандартные потоки, файловые потоки обеспечивают гораздо более надежный ввод-вывод, чем старые функции библиотеки C. Для использования файловых потоков необходимо подключить заголовок `<fstream>`.

Работа с файлом обычно предполагает следующие операции:

- ☐ создание потока (потокowego объекта);
- ☐ открытие потока и связывание его с файлом;
- ☐ обмен с потоком (ввод-вывод);
- ☐ закрытие файла.

Классы файловых потоков содержат несколько конструкторов, позволяющих варьировать способы создания потоковых объектов. *Конструкторы с параметрами* создают объект соответствующего класса, открывают файл с указанным именем и связывают файл с объектом:

```
ifstream( const char* name, int mode = ios::in );
ofstream( const char* name, int mode = ios::out | ios::trunc );
fstream ( const char* name, int mode = ios::in | ios::out );
```

Второй параметр конструктора задает режим открытия файла. Если значение по умолчанию вас не устраивает, можно указать другое, выбрав одно или несколько значений (объединенных операцией |) из указанных в табл. 14.2.

Таблица 14.2. Значения аргумента mode

Флаг	Назначение
ios::in	Открыть файл для ввода
ios::out	Открыть файл для вывода
ios::ate	Установить указатель на конец файла
ios::app	Открыть в режиме добавления в конец файла
ios::trunc	Если файл существует, обрезать его до нулевой длины
ios::binary	Открыть в двоичном режиме (по умолчанию используется текстовый)

Конструкторы без параметров создают объект соответствующего класса, не связывая его с файлом. В этом случае связь потока с конкретным файлом осуществляется позже — вызовом метода `open`, который имеет параметры, аналогичные параметрам рассмотренных выше конструкторов. Приведем примеры создания потоковых объектов и связывания их с конкретными файлами:

```
ofstream flog( "flog.txt" );           // <---- Файлы для вывода
ofstream fout1, fout2;                // <----
fout1.open( "test1", ios::app );      // <----
fout2.open( "test2", ios::binary );   // <----
ifstream finpl( "data.txt" );         //      Файл для ввода
fstream myfile;                       //      Файл для ввода и вывода
myfile.open( "mf.dat" );
```

Если в качестве параметра `name` задано *краткое имя файла* (без указания полного пути), подразумевается, что файл открывается в текущем каталоге, в противном случае требуется задавать *полное имя файла*, например:

```
ifstream finpl( "D:\\Vcwork\\Task1\\data.txt" );
```

ВНИМАНИЕ

Если файл не удастся открыть (например, входной файл в указанном каталоге не найден или нет свободного места на диске для выходного файла), то независимо от способа его открытия — конструктором или методом `open` — потоковый объект принимает значение, равное нулю. Поэтому рекомендуется всегда проверять, чем завершилась попытка открытия файла, например:

```
ifstream finpl( "data.txt" );
if ( !finpl ) { cerr << "Файл data.txt не найден." << endl;
                throw "Ошибка открытия файла "; }
```

После того как файловый поток открыт, работа с ним чрезвычайно проста: с входным потоком можно обращаться так же, как со стандартным объектом `cin`, а с выходным так же, как со стандартным объектом `cout`.

Если при чтении данных требуется контролировать, был ли достигнут *конец файла* после очередной операции ввода, используется метод eof, возвращающий нулевое значение, если конец файла еще не достигнут, и ненулевое значение — если уже достигнут. Учтите, что в C++ после чтения из файла *последнего элемента* условие конца файла *не возникает*! Оно возникает при следующем чтении, когда программа пытается считать данные за последним элементом в файле.

Если при выполнении операций ввода-вывода фиксируется некоторая ошибочная ситуация, потоковый объект также принимает значение, равное нулю. Рекомендуется особо следить за состоянием потокового объекта во время выполнения операций вывода, так как диски «не резиновые» и имеют тенденцию переполняться.

Когда программа покидает область видимости потокового объекта, он уничтожается. При этом перестает существовать связь между потоковым объектом и физическим файлом, а физический файл закрывается. Если алгоритм требует более раннего закрытия файла, можно воспользоваться методом close.

Для примера работы с файловыми потоками приведем программу копирования одного файла в другой (листинг 14.1) и программу вывода на экран содержимого текстового файла (листинг 14.2).

Листинг 14.1. Копирование файлов

```
#include <iostream>
#include <fstream>
using namespace std;
void error(const char* text1, const char* text2 = "") {
    cerr << text1 << ' ' << text2 << endl;
    cin.get(); exit(1);
}
int main(int argc, char* argv[]) {    // Имена файлов берутся из командной строки
    if (argc != 3) error("Неверное число аргументов");
    ifstream from(argv[1]);           // открываем входной файл
    if (!from) error("Входной файл не найден:", argv[1]);
    ofstream to(argv[2]);             // открываем выходной файл
    if (!to) error("Выходной файл не открыт:", argv[2]);
    char ch;
    while ( from.get(ch) ) {
        to.put(ch);
        if (!to) error("Ошибка записи (диск переполнен).");
    }
    cout << "Копирование из " << argv[1] << " в " << argv[2] << " завершено.\n";
    cin.get();
}
```

Листинг 14.2. Вывод на экран содержимого текстового файла

```
#include <iostream>
#include <fstream>
using namespace std;
// ... <----- Здесь определение функции error() – из листинга 14.1
int main(int argc, char* argv[]) { // имя файла задается в командной строке
    if (argc != 2) error("Неверное число аргументов");
```

продолжение ➤

Листинг 14.2 (продолжение)

```

    ifstream tfile(argv[1]);                // открываем входной файл
    if (!tfile) error("Входной файл не найден:", argv[1]);
    char buf[1024];
    while (!tfile.eof()) {
        tfile.getline(buf, sizeof(buf));
        cout << buf << endl;
    }
    cin.get();
}

```

В программе предполагается, что длина строки в файле не превышает 1024 символов. При необходимости можно увеличить размер буфера `buf` до требуемой величины.

Строковые потоки

Работу со строковыми потоками обеспечивают классы `istringstream`, `ostringstream` и `stringstream`, которые являются производными от классов `istream`, `ostream` и `iostream` соответственно. Для использования строковых потоков необходимо подключить к программе заголовочный файл `<sstream>`.

Применение строковых потоков аналогично применению файловых потоков, но информация потока физически размещается в оперативной памяти, а не на диске. Кроме того, классы строковых потоков содержат метод `str`, возвращающий копию строки типа `string` или присваивающий потоку значение такой строки:

```

string str() const;
void str( const string& s );

```

Строковые потоки являются некоторыми аналогами функций `sscanf` и `sprintf` библиотеки C, которые также работают со строками в памяти, имитируя консольный ввод-вывод. Например, с помощью `sprintf` можно сформировать в памяти некоторую символьную строку, которую затем отобразить на экране. Эта же проблема легко решается с помощью объекта типа `ostringstream`.

В качестве примера приведем модифицированную версию листинга 14.2, которая выводит содержимое текстового файла на экран, предваряя каждую строку текстовой меткой «Line N:», где N — номер строки (листинг 14.3).

Листинг 14.3. Вывод на экран пронумерованного содержимого текстового файла

```

#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
using namespace std;
// ... <----- Здесь определение функции error() — из листинга 14.1
int main(int argc, char* argv[]) {
    if (argc != 2) error("Неверное число аргументов.");
    ifstream tfile( argv[1] );
    if (!tfile) error("Входной файл не найден:", argv[1]);
}

```

```

int n = 0;
char buf[1024];
while ( !tfile.eof() ) {
    n++;
    tfile.getline(buf, sizeof(buf));
    ostringstream line;
    line << "Line " << setw(3) << n << ": " << buf << endl;
    cout << line.str();
}
cin.get();
}

```

Строки класса string

Мы уже пользовались объектами класса string, начиная со второго семинара, и успели оценить удобства, обеспечиваемые этим классом в сравнении с традиционными C-строками (то есть массивами символов типа char, завершаемыми нулевым байтом). Сейчас мы рассмотрим строки типа string более подробно.

Для использования строк типа string необходимо подключить к программе заголовочный файл <string>. Важнейшей особенностью класса string является управление памятью как при размещении строки, так и при ее модификациях, изменяющих длину строки. Поэтому вы можете «забыть» об операциях new и delete, неаккуратное обращение с которыми является источником труднодиагностируемых ошибок.

Кроме этого, строки типа string защищены от ошибочных обращений к памяти, связанных с выходом за их границы. Но за все надо платить: строки типа string значительно проигрывают C-строкам в эффективности. Поэтому, если от программы требуется максимальное быстродействие, иногда лучше воспользоваться C-строками. В большинстве же программ на C++ строки типа string обеспечивают необходимую скорость обработки, поэтому их применение предпочтительней.

Для понимания определений методов класса string необходимо знать назначение некоторых имен. Так, в пространстве std определен идентификатор size_type, являющийся синонимом типа unsigned int. В классе string определена константа npos, задающая максимально возможное число, которое в зависимости от контекста означает либо «все элементы» строки, либо отрицательный результат поиска. Так как максимально возможное число имеет вид 0xFFFF...FFFF, то в случае присваивания его переменной типа int получится значение -1.

В классе string имеется несколько конструкторов, вот самые простые:

```

string(); // создает пустой объект класса string
string( const char* ); // создает объект, инициализируя его значением C-строки

```

Класс содержит три операции присваивания:

```

string& operator=( const string& str ); // присвоить значение другой строки string
string& operator=( const char* s ); // присвоить значение C-строки
string& operator=( char c ); // присвоить значение символа

```

В табл. 14.3 приведены допустимые для объектов класса string операции.

Таблица 14.3. Операции класса `string`

Операция	Действие	Операция	Действие
=	Присваивание	>	Больше
+	Конкатенация	>=	Больше или равно
==	Равенство	[]	Индексация
!=	Неравенство	<<	Вывод
<	Меньше	>>	Ввод
<=	Меньше или равно	+=	Добавление

Использование операций очевидно. Размеры строк устанавливаются автоматически так, чтобы объект мог содержать присваиваемое ему значение. Нумерация элементов строки начинается с нуля. Кроме операции индексации, для доступа к элементу строки определен метод `at(size_type n)`, который можно использовать как для чтения, так и для записи n -го элемента строки:

```
cout << s.at(2);      // Будет выведен 2-й символ строки s
s.at(5) = 'W';        // 5-й символ заменяется символом W
```

Заметим, что в операции индексации не проверяется выход за диапазон строки. Метод `at`, напротив, такую проверку содержит, и, если индекс превышает длину строки, порождается исключение `out_of_range`.

В табл. 14.4 приведены некоторые употребительные методы класса `string`.

Таблица 14.4. Методы класса `string`

Метод	Назначение
<code>size_type size() const;</code>	Возвращает размер строки
<code>size_type length() const;</code>	То же, что и <code>size</code>
<code>insert(size_type pos1, const string& str);</code>	Вставляет строку <code>str</code> в вызывающую строку, начиная с позиции <code>pos1</code>
<code>replace(size_type pos1, size_type n1, const string& str);</code>	Заменяет <code>n1</code> элементов, начиная с позиции <code>pos1</code> вызывающей строки, элементами строки <code>str</code>
<code>string substr(size_type pos=0, size_type n=npos) const;</code>	Возвращает подстроку длиной <code>n</code> , начиная с позиции <code>pos</code>
<code>size_type find(const string& str, size_type pos=0) const;</code>	Ищет самое левое вхождение строки <code>str</code> в вызывающую строку, начиная с позиции <code>pos</code> . Возвращает позицию вхождения, или <code>npos</code> , если вхождение не найдено
<code>size_type find(char c, size_type pos=0) const;</code>	Ищет самое левое вхождение символа <code>c</code> , начиная с позиции <code>pos</code> . Возвращает позицию вхождения, или <code>npos</code> , если вхождение не найдено

Метод	Назначение
<code>size_type rfind(const string& str, size_type pos=0) const;</code>	Ищет самое правое вхождение строки <code>str</code> , начиная с позиции <code>pos</code> ¹
<code>size_type rfind(char c, size_type pos=0) const;</code>	Ищет самое правое вхождение символа <code>c</code> , начиная с позиции <code>pos</code>
<code>size_type find_first_of(const string& str, size_type pos=0) const;</code>	Ищет самое левое вхождение любого символа строки <code>str</code> , начиная с позиции <code>pos</code>
<code>size_type find_last_of(const string& str, size_type pos=0) const;</code>	Ищет самое правое вхождение любого символа строки <code>str</code> , начиная с позиции <code>pos</code>
<code>swap(string& str);</code>	Обменивает содержимого вызывающей строки и строки <code>str</code>
<code>erase(size_type pos=0, size_type n=npos);</code>	Удаляет <code>n</code> элементов, начиная с позиции <code>pos</code>
<code>clear();</code>	Очищает всю строку
<code>const char* c_str() const;</code>	Возвращает указатель на C-строку, содержащую копию вызывающей строки. Полученную C-строку нельзя пытаться изменить
<code>Size_type copy(char* s, size_type n, size_type pos=0) const;</code>	Копирует в массив <code>s</code> <code>n</code> элементов вызывающей строки, начиная с позиции <code>pos</code> . Нуль-символ в результирующий массив не заносится. Метод возвращает количество скопированных элементов

Поясним применение метода `find`. Допустим, что вы работаете над программой для игры в шахматы с компьютером, а в данный момент пишете функцию для ввода обозначения колонки шахматной доски. Эти колонки, как известно, обозначаются символами латинского алфавита от А до Н. Желательно, чтобы ваша функция не допускала ввод некорректных символов. Приведем одно из возможных решений:

```
char GetColumn() {
    string goodChar = "ABCDEFGH";
    char symb;
    cout << "Введите обозначение колонки: ";
    while (1) {
        cin >> symb;
        if ( -1 == goodChar.find( symb ) ) {
            cout << "Ошибка. Введите корректный символ:\n"; continue; }
        return symb;
    }
}
```

¹ Возвращаемые значения такие же, как и у метода `find`.

Метод `find` здесь используется для проверки, принадлежит ли введенный с клавиатуры символ множеству символов, заданному с помощью строки `goodChar`.

Задача 14.1. Подсчет количества вхождений слова в текст

Написать программу, которая определяет, сколько раз встретилось заданное слово в текстовом файле. Текст не содержит переносов слов. Максимальная длина строки в файле неизвестна¹.

Определим *слово* как последовательность алфавитно-цифровых символов, после которых следует либо знак пунктуации (., ,, ?, !), либо разделитель. В качестве разделителей могут выступать один или несколько пробелов, один или несколько символов табуляции `'\t'` и символ конца строки `'\n'`. Для хранения заданного слова (оно вводится с клавиатуры) определим переменную `word` типа `string`.

Поскольку максимальная длина строки в файле неизвестна, будем читать файл не построчно, а пословно, размещая очередное прочитанное слово в переменной `curword` типа `string`. Это можно реализовать с помощью операции `>>`, которая в случае операнда типа `string` игнорирует все разделители, предваряющие текущее слово, и считывает символы текущего слова в переменную `curword`, пока не встретится очередной разделитель.

Очевидно, что «опознание» текущего слова должно осуществляться с учетом возможного наличия после него одного из знаков пунктуации. Для решения этой задачи определим глобальную функцию

```
bool equal( const string& cw, const string& w );
```

которая возвращает значение `true`, если текущее слово `cw` совпадает с заданным словом `w` с точностью до знака пунктуации, или `false` — в противном случае.

Имея такую функцию, очень просто составить алгоритм основного цикла:

- ☐ прочитать очередное слово;
- ☐ если оно совпадает с заданным словом `w` (с точностью до знака пунктуации), то увеличить на единицу значение счетчика `count`.

Текст решения приведен в листинге 14.4.

Листинг 14.4. Подсчет количества вхождений слова в текст

```
#include <windows.h> // здесь прототип функции OemToChar
#include <iostream>
#include <fstream>
#include <string>
#include <locale> // здесь прототип функции setlocale
using namespace std;
bool equal( const string& cw, const string& w ) {
    char punct[] = { '.', ',', '?', '!' };
```

¹ Аналогичная задача решалась в первой части практикума (задача 5.2), но с упрощающим ограничением: длина строки в файле не более 80 символов.

```

    if ( cw == w ) return true;
    for ( int i = 0; i < sizeof( punct ); ++i )
        if ( cw == w + punct[i] ) return true;
    return false;
}
int main() {
    string word, curword;
    setlocale( LC_ALL, "Russian" );           // Для работы с русскими символами
    cout << " Введите слово для поиска: ";
    cin >> word;
    char pattern[30];
    OemToChar( word.c_str(), pattern );       // Для работы с русскими символами
    ifstream fin("infile.txt");
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; cin.get(); return 1; }
    int count = 0;
    while ( !fin.eof() ) {
        fin >> curword;
        if ( equal( curword, string( pattern ) ) ) count++;
    }
    cout << "Количество вхождений слова: " << count << endl;
    cin.get(); cin.get();
}

```

Обратите внимание на реализацию функции `equal` и, в частности, на использование операции сложения для добавления в конец строки `w` одного из знаков пунктуации.

Задача 14.2. Вывод вопросительных предложений

Написать программу, которая считывает текст из файла и выводит на экран только вопросительные предложения из этого текста¹.

Итак, имеется текстовый файл неизвестного размера, состоящий из неизвестного количества предложений. Предложение может занимать несколько строк, поэтому читать файл построчно неудобно. При решении аналогичной задачи в первой части практикума было принято решение выделить буфер, в который поместится *весь* файл. Такое решение тоже нельзя признать идеальным — ведь файл может иметь сколь угодно большие размеры, и тогда программа окажется неработоспособной.

Поищем более удачное решение, используя новые средства языка C++, с которыми мы познакомились на этом семинаре. Попробуем читать файл пословно, как и в предыдущей программе, в переменную `word` типа `string`, и отправлять каждое прочитанное слово в строковый поток `sentence` типа `ostream`, который, как вы уже догадались, будет хранилищем очередного предложения.

При таком подходе, однако, есть проблема, связанная с потерей разделителей при чтении файла операцией `fin >> word`. Чтобы ее решить, будем «заглядывать» в следующую позицию файлового потока `fin` с помощью метода `peek`. При обнаружении

¹ Аналогичная задача рассмотрена в первой части практикума (задача 5.3).

символа-разделителя его нужно отправить в поток `sentence` и переместиться на следующую позицию в потоке `fin`, используя метод `seekg`. Подробности обнаружения символа-разделителя инкапсулируем в глобальную функцию `isLimit`.

Осталось решить подзадачи:

- ☐ обнаружить конец предложения, то есть один из символов '.', '!', '?';
- ☐ если это вопросительное предложение, вывести его в поток `cout`, в противном случае очистить поток `sentence` для накопления следующего предложения.

Рассмотренный алгоритм реализуется в листинге 14.5.

Листинг 14.5. Вывод вопросительных предложений

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;
bool isLimit( char c ) {
    char lim[] = { ' ', '\t', '\n' };
    for ( int i = 0; i < sizeof( lim ); ++i ) if ( c == lim[i] ) return true;
    return false;
}
int main() {
    ifstream fin( "infile.txt" );
    if ( !fin ) { cout << "Ошибка открытия файла." << endl; cin.get(); return 1; }
    int count = 0;
    string word;
    ostringstream sentence;
    while( !fin.eof() ) {
        char symb;
        while( isLimit( symb = fin.peek() ) ) {
            sentence << symb;
            if ( symb == '\n' ) break;
            fin.seekg( 1, ios::cur );
        }
        fin >> word;
        sentence << word;
        char last = word[word.size() - 1];
        if ( ( last == '.' ) || ( last == '!' ) ) {
            sentence.str(""); // очистка потока
            continue;
        }
        if ( last == '?' ) { cout << sentence.str(); sentence.str( "" ); count++; }
    }
    if ( !count ) cout << "Вопросительных предложений нет." << endl;
    cin.get();
}
```

Протестируйте приведенные программы. Не забудьте поместить в один каталог с программой текстовый файл `infile.txt`. Если программа запускается из среды

Visual Studio, файл должен находиться в каталоге с исходными текстами проекта, если же из папки `Debug` (например, в режиме командной строки), то в этой папке.

Итоги

1. Для поддержки файлового ввода и вывода стандартная библиотека C++ содержит классы `ifstream`, `ofstream`, `fstream`.
2. Работа с файлом предполагает следующие операции: создание потока, открытие потока и связывание его с файлом, обмен с потоком (ввод-вывод), закрытие файла. Рекомендуется всегда проверять, чем завершилось открытие файла.
3. Если в процессе ввода-вывода фиксируется ошибочная ситуация, потоковый объект принимает значение, равное нулю.
4. Следите за состоянием выходного потока после каждой операции вывода, так как на диске может не оказаться свободного места.
5. Работу со строковыми потоками обеспечивают классы `istringstream`, `ostringstream`, `stringstream`. Использование строковых потоков аналогично применению файловых потоков. Различие в том, что физически информация потока размещается в оперативной памяти, а не в файле на диске.
6. Класс `string` стандартной библиотеки C++ предоставляет программисту очень удобные средства работы со строками. Класс берет на себя управление памятью, как при размещении строки, так и при всех ее модификациях.

Задания

Выполнить задания семинара 5, используя потоковые классы.

Семинар 15. Стандартная библиотека шаблонов

Теоретический материал: с. 295–368.

Основные концепции стандартной библиотеки

Стандартная библиотека шаблонов (Standard Template Library, STL) состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты. Контейнерные классы являются шаблонными, поэтому хранимые в них объекты могут быть как встроенных, так и пользовательских типов. Эти объекты должны допускать *копирование* и *присваивание*. Встроенные типы этим требованиям удовлетворяют; то же самое относится к классам, если конструктор копирования или операция присваивания не объявлены в них закрытыми или защищенными. В контейнерных классах реализованы такие типовые структуры данных, как стек, список, очередь и т. д.

Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам — например, поиск, сортировку, слияние и т. п., но они не являются методами контейнерных классов. Наоборот, алгоритмы представлены в STL в форме глобальных шаблонных функций. Благодаря этому достигается их универсальность: эти функции можно применять не только к объектам контейнерных классов, но и к массивам. Независимость от типов контейнеров достигается за счет косвенной связи функции с контейнером: в функцию передается не сам контейнер, а пара адресов, *first*, *last*, задающая диапазон обрабатываемых элементов.

Реализация указанного механизма взаимодействия базируется на использовании так называемых *итераторов*. *Итераторы* — это обобщение концепции указателей: они ссылаются на элементы контейнера. Их можно инкрементировать для последовательного продвижения по контейнеру, как обычные указатели, а также разыменовывать для получения или изменения значения элемента.

Контейнеры

Контейнеры STL можно разделить на два типа: последовательные и ассоциативные.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым последовательным контейнерам относятся *векторы* (vector), *списки* (list) и *двусторонние очереди* (deque). Есть еще специализированные контейнеры (или *адаптеры* контейнеров), реализованные на основе базовых, — *стеки* (stack), *очереди* (queue) и *очереди с приоритетами* (priority_queue).

Кстати, обычный встроенный массив C++ также может рассматриваться как последовательный контейнер. Проблема с массивами заключается в том, что их размеры нужно указывать в исходном коде, а это часто бывает неизвестно заранее. Если же выделять память для массива динамически (операцией new), алгоритм усложняется из-за необходимости отслеживать время жизни массива и вовремя освобождать память. Использование контейнера *вектор* вместо динамического массива упрощает жизнь программиста, в чем вы могли убедиться на семинаре 11.

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

```
vector<int> aVect;      // создать вектор aVect целых чисел (типа int)
list<Man> department;  // создать список department из элементов типа Man
```

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Они построены на основе сбалансированных деревьев. Есть пять типов ассоциативных контейнеров: словари (map), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

Итераторы

Чтобы понять, зачем нужны итераторы, давайте посмотрим, как можно реализовать шаблонную функцию для поиска значения value в обычном массиве, хранящем объекты типа T. Например, возможно следующее решение:

```
template <class T> T* Find( T* ar, int n, const T& value ) {
    for ( int i = 0; i < n; ++i ) if ( ar[i] == value ) return &ar[i];
    return 0;
}
```

Функция возвращает адрес найденного элемента или 0, если элемент с заданным значением не найден. Цикл for может быть записан и в несколько иной форме:

```
for ( int i = 0; i < n; ++i ) if ( *( ar + i ) == value ) return ar + i;
```

Работа функции при этом останется прежней. При продвижении по массиву адрес следующего элемента вычисляется и в первом, и во втором случаях с использованием арифметики указателей, то есть адреса соседних элементов различаются на количество байтов, требуемое для хранения одного элемента.

Попытаемся теперь расширить сферу применения нашей функции — хорошо бы, чтобы она решала задачу поиска заданного значения среди объектов, хранящихся в виде линейного списка! Однако, к сожалению, ничего не выйдет: адрес следующего элемента в списке нельзя вычислить, пользуясь арифметикой указателей.

Элементы списка могут размещаться в памяти самым причудливым образом, а информация об адресе следующего объекта хранится в одном из полей текущего объекта.

Авторы STL решили эту проблему, введя понятие *итератора* как более абстрактной сущности, чем указатель, но обладающей похожим поведением. Для всех контейнерных классов STL определен тип `iterator`, однако реализация его в разных классах разная. Например, в классе `vector`, где объекты размещаются один за другим, как в массиве, тип итератора определяется посредством `typedef T* iterator`. А вот в классе `list` тип итератора реализован как встроенный класс `iterator`, поддерживающий основные операции с итераторами.

К *основным операциям*, выполняемым с любыми итераторами, относятся:

- ❑ Разыменование итератора: если `p` — итератор, то `*p` — значение объекта, на который он ссылается.
- ❑ Присваивание одного итератора другому.
- ❑ Сравнение итераторов на равенство и неравенство (`==` и `!=`).
- ❑ Перемещение его по всем элементам контейнера с помощью префиксного (`++p`) или постфиксного инкремента (`p++`).

Так как реализация итератора специфична для каждого класса, при объявлении объектов типа «итератор» всегда указывается область видимости в форме `имя_шаблона::`, например:

```
vector<int>::iterator iter1;  
list<Man>::iterator iter2;
```

Организация циклов просмотра элементов контейнеров тоже имеет некоторую специфику. Так, если `i` — некоторый итератор, то вместо привычной формы

```
for ( i = 0; i < n; ++i )
```

используется следующая:

```
for ( i = first; i != last; ++i )
```

где `first` — значение итератора, указывающее на первый элемент в контейнере, а `last` — значение итератора, указывающее на воображаемый элемент, который следует за последним элементом контейнера. Операция `<` заменена на операцию `!=`, поскольку операции `<` и `>` для итераторов в общем случае не поддерживаются.

Для всех контейнерных классов определены унифицированные методы `begin` и `end`, возвращающие адреса `first` и `last` соответственно.

Вообще, все типы итераторов в STL принадлежат одной из пяти категорий: входные, выходные, прямые, двунаправленные итераторы и итераторы произвольного доступа.

Входные итераторы (*InputIterator*) используются алгоритмами STL для чтения значений из контейнера, аналогично тому, как вводятся данные из потока `cin`.

Выходные итераторы (*OutputIterator*) используются алгоритмами для записи значений в контейнер, аналогично тому, как выводятся данные в поток `cout`.

Прямые итераторы (*ForwardIterator*) используются алгоритмами для навигации по контейнеру только в прямом направлении, причем они позволяют и читать, и изменять данные в контейнере.

Двунаправленные итераторы (*BidirectionalIterator*) имеют все свойства прямых итераторов, но позволяют осуществлять навигацию по контейнеру и в прямом, и в обратном направлениях (для них дополнительно реализованы операции префиксного и постфиксного декремента).

Итераторы произвольного доступа (*RandomAccessIterator*) имеют все свойства двунаправленных итераторов плюс операции (наподобие сложения указателей) для доступа к произвольному элементу контейнера.

Значения прямых, двунаправленных и итераторов произвольного доступа могут быть сохранены, а вот значения входных и выходных итераторов сохраняться не могут (аналогично тому, как не может быть гарантирован ввод тех же самых значений при вторичном обращении к потоку `cin`). Как следствие, любые алгоритмы, основанные на входных или выходных итераторах, должны быть однопроходными.

Вернемся к функции `Find`, которую мы безуспешно пытались обобщить для любых типов контейнеров. В STL аналогичный алгоритм имеет следующий прототип:

```
template <class InputIterator, class T>
InputIterator find( InputIterator first, InputIterator last, const T& value );
```

Для двунаправленных итераторов и итераторов произвольного доступа определены разновидности, называемые *адаптерами итераторов*. Адаптер, просматривающий последовательность в обратном направлении, называется `reverse_iterator`. Другие специализированные итераторы-адаптеры мы рассмотрим ниже.

Общие свойства контейнеров

В табл. 15.1 приведены имена типов, определенные с помощью `typedef` в большинстве контейнерных классов, а в табл. 15.2 — некоторые общие для всех контейнеров операции.

Таблица 15.1. Унифицированные типы, определенные в STL

Поле	Пояснение
<code>value_type</code>	Тип элемента контейнера
<code>size_type</code> ¹	Тип индексов, счетчиков элементов и т. д.
<code>iterator</code>	Итератор
<code>const_iterator</code>	Константный итератор (значения элементов изменять запрещено)
<code>reference</code>	Ссылка на элемент

продолжение ➤

¹ Эквивалентен `unsigned int`.

Таблица 15.1 (продолжение)

Поле	Пояснение
<code>const_reference</code>	Константная ссылка на элемент (значение элемента изменять запрещено)
<code>key_type</code>	Тип ключа (для ассоциативных контейнеров)
<code>key_compare</code>	Тип критерия сравнения (для ассоциативных контейнеров)

Таблица 15.2. Операции и методы, общие для всех контейнеров

Операция или метод	Пояснение
Операции равенства (<code>==</code>) и неравенства (<code>!=</code>)	Возвращают значение <code>true</code> или <code>false</code>
Операция присваивания (<code>=</code>)	Копирует один контейнер в другой
<code>clear</code>	Удаляет все элементы
<code>insert</code>	Добавляет один элемент или диапазон элементов
<code>erase</code>	Удаляет один элемент или диапазон элементов
<code>size_type size() const</code>	Возвращает число элементов
<code>size_type max_size() const</code>	Возвращает максимально допустимый размер контейнера
<code>bool empty() const</code>	Возвращает <code>true</code> , если контейнер пуст
<code>iterator begin()</code>	Возвращают итератор на начало контейнера (итерации будут производиться в прямом направлении)
<code>iterator end()</code>	Возвращают итератор на конец контейнера (итерации в прямом направлении будут закончены)
<code>reverse_iterator begin()</code>	Возвращают реверсивный итератор на конец контейнера (итерации будут производиться в обратном направлении)
<code>reverse_iterator end()</code>	Возвращают реверсивный итератор на начало контейнера (итерации в обратном направлении будут закончены)

Алгоритмы

Алгоритм — это функция, которая производит некоторые действия над элементами контейнера (контейнеров). Чтобы использовать обобщенные алгоритмы, нужно подключить к программе заголовочный файл `<algorithm>`. В табл. 15.3 приведены наиболее популярные алгоритмы STL.

Таблица 15.3. Некоторые типичные алгоритмы STL

Алгоритм	Назначение
<code>accumulate</code>	Вычисление суммы элементов в заданном диапазоне
<code>copy</code>	Копирование последовательности, начиная с первого элемента

Алгоритм	Назначение
count	Подсчет количества вхождений значения в последовательность
count_if	Подсчет количества выполнений условия в последовательности
equal	Попарное равенство элементов двух последовательностей
fill	Замена всех элементов заданным значением
find	Нахождение первого вхождения значения в последовательность
find_first_of	Нахождение первого значения из одной последовательности в другой
find_if	Нахождение первого соответствия условию в последовательности
for_each	Вызов функции для каждого элемента последовательности
merge	Слияние отсортированных последовательностей
remove	Перемещение элементов с заданным значением
replace	Замена элементов с заданным значением
search	Нахождение первого вхождения в первую последовательность второй последовательности
sort	Сортировка
swap	Обмен местами двух элементов
transform	Выполнение заданной операции над каждым элементом последовательности

В списках параметров всех алгоритмов первые два параметра задают диапазон обрабатываемых элементов в виде полуинтервала $[first, last)$ ¹, где *first* — итератор, указывающий на начало диапазона, а *last* — итератор, указывающий на выход за границы диапазона. Пример сортировки массива с помощью алгоритма `sort` (имя массива и `arr` имеет тип указателя `int*` и используется как итератор):

```
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 };
sort( arr, arr + 7 );
```

Примеры использования некоторых алгоритмов будут даны ниже.

Использование последовательных контейнеров

К основным последовательным контейнерам относятся *вектор* (*vector*), *список* (*list*) и *двусторонняя очередь* (*deque*). Чтобы использовать последовательный

¹ Полуинтервал $[a, b)$ — это промежуток, включающий *a*, но не включающий *b*, — последний элемент полуинтервала предшествует элементу *b*.

контейнер, нужно включить в программу соответствующий заголовочный файл: `<vector>`, `<list>` или `<deque>`.

Контейнер *вектор* является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации `[]` или метода `at`¹. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Контейнер *список* организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки *n*-го элемента нужно последовательно выбрать предыдущие *n-1* элементов.

Контейнер *двусторонняя очередь (дек)* во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора дек эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

Существует пять способов определить объект для последовательного контейнера.

1. Создать пустой контейнер:

```
vector<int> vec1;  
list<string> list1;
```

2. Создать контейнер заданного размера и инициализировать его элементы значениями по умолчанию:

```
vector<string> vec1( 100 );  
list<double> list1( 20 );
```

3. Создать контейнер заданного размера и инициализировать его элементы указанным значением:

```
vector<string> vec1( 100, "Hello!" );  
deque<int> dec1( 300, -1 );
```

4. Создать контейнер и инициализировать его элементы значениями диапазона `[first, last)` элементов другого контейнера:

```
int arr[7] = { 15, 2, 19, -3, 28, 6, 8 };  
vector<int> v1( arr, arr + 7 );  
list<int> l1( v1.begin() + 2, v1.end() );
```

5. Создать контейнер и инициализировать его элементы значениями элементов другого *однотипного* контейнера:

```
vector<int> v1;  
// ... здесь добавление элементов в v1  
vector<int> v2( v1 );
```

¹ Метод `at` аналогичен операции индексации, но в отличие от нее проверяет выход за границу вектора и в случае ошибки генерирует исключение `out_of_range`.

Для вставки и удаления последнего элемента контейнера любого из трех рассматриваемых классов предназначены методы `push_back` и `pop_back`. Кроме того, список и очередь (но не вектор) поддерживают операции вставки и удаления первого элемента контейнера `push_front` и `pop_front`. Учтите, что методы `pop_back` и `pop_front` *не возвращают* удаленное значение. Чтобы считать первый элемент, используется метод `front`, а для считывания последнего элемента — метод `back`. Кроме этого, все типы контейнеров имеют более общие операции вставки и удаления, перечисленные в табл. 15.4.

Таблица 15.4. Методы вставки в контейнер и удаления из контейнера

Метод	Пояснение
<code>insert(iterator position, const T& value)</code>	Вставка элемента со значением <code>value</code> в позицию, заданную итератором <code>position</code>
<code>insert(iterator position, size_type n, const T& value)</code>	Вставка <code>n</code> элементов со значением <code>value</code> , начиная с позиции <code>position</code>
<code>template <class InputIter> void insert(iterator position, InputIter first, InputIter last)</code>	Вставка диапазона элементов, заданного итераторами <code>first</code> и <code>last</code> , начиная с позиции <code>position</code>
<code>erase(iterator position)</code>	Удаление элемента, на который указывает итератор <code>position</code>
<code>erase(iterator first, iterator last)</code>	Удаление диапазона элементов, заданного позициями <code>first</code> и <code>last</code>

Задача 15.1. Сортировка вектора

В файле находится произвольное количество целых чисел. Вывести их на экран в порядке возрастания.

Решение этой задачи приводится в листинге 15.1. Вместо вектора можно использовать любой последовательный контейнер, заменив слово `vector` на `deque` или `list`. При этом изменится внутреннее представление данных, а результат работы программы останется таким же.

Листинг 15.1. Сортировка вектора

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    ifstream in ( "inpnun.txt" );
    if ( !in ) { cerr << "File not found\n"; exit( 1 ); }
    vector<int> v;
    int x;
    while ( in >> x ) v.push_back( x );
```

продолжение ➤

Листинг 15.1 (продолжение)

```

    sort( v.begin(), v.end() );
    vector<int>::const_iterator i;
    for ( i = v.begin(); i != v.end(); ++i ) cout << *i << " ";
}

```

Приведем еще один пример работы с векторами, демонстрирующий использование методов `swap`, `empty`, `back`, `pop_back`:

```

#include <iostream>
#include <vector>
using namespace std;
int main() {
    double arr[] = { 1.1, 2.2, 3.3, 4.4 };
    int n = sizeof( arr )/sizeof( double );
    vector<double> v1( arr, arr + n );           // Инициализация вектора массивом
    vector<double> v2;                           // пустой вектор
    v1.swap( v2 );                               // обменять содержимое v1 и v2
    while ( !v2.empty() ) {
        cout << v2.back() << ' ';               // вывести последний элемент
        v2.pop_back();                           // и удалить его
    }
} // Результат выполнения программы: 4.4 3.3 2.2 1.1

```

Шаблонная функция `print` для вывода содержимого контейнера

В процессе работы над программами, использующими контейнеры, часто приходится выводить на экран их текущее содержимое. Приведем шаблон функции, решающей эту задачу для любого типа контейнера (листинг 15.2).

Листинг 15.2. Шаблонная функция `print` для вывода содержимого контейнера

```

template <class T> void print( T& cont ) {
    typename T::const_iterator p = cont.begin();
    if ( cont.empty() ) cout << "Container is empty.";
    for ( p; p != cont.end(); ++p ) cout << *p << ' ';
    cout << endl;
}

```

Теперь можно пользоваться функцией `print`, включая ее определение в исходный файл с программой, как, например, в следующем эксперименте с очередью:

```

#include <iostream>
#include <deque>
using namespace std;
/* ... определение функции print ... */
int main() {
    deque<int> dec;      print( dec ); // Пустой контейнер
    dec.push_back( 4 );  print( dec ); // 4
}

```

```

dec.push_front( 3 ); print( dec ); // 3 4
dec.push_back( 5 ); print( dec ); // 3 4 5
dec.push_front( 2 ); print( dec ); // 2 3 4 5
dec.push_back( 6 ); print( dec ); // 2 3 4 5 6
dec.push_front( 1 ); print( dec ); // 1 2 3 4 5 6
}

```

Адаптеры контейнеров

Специализированные последовательные контейнеры — *стек*, *очередь* и *очередь с приоритетами* — не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются *адаптерами контейнеров*.

Стек

Шаблонный класс `stack` (заголовочный файл `<stack>`) определен как

```

template <class T, class Container = deque<T> >
class stack { /* ... */ };

```

где параметр `Container` задает класс-прототип. По умолчанию для стека прототипом является класс `deque`. Смысл такой реализации заключается в том, что специализированный класс переопределяет интерфейс класса-прототипа, ограничивая его только методами, необходимыми новому классу. В табл. 15.5 показано, как сформирован интерфейс класса `stack` из методов класса-прототипа.

Таблица 15.5. Интерфейс класса `stack`

Методы класса <code>stack</code>	Методы класса-прототипа
<code>push</code>	<code>push_back</code>
<code>pop</code>	<code>pop_back</code>
<code>top</code>	<code>back</code>
<code>empty</code>	<code>empty</code>
<code>size</code>	<code>size</code>

В соответствии со своим назначением стек не только не позволяет выполнить произвольный доступ к своим элементам, но даже не дает возможности пошагового перемещения, в связи с чем концепция итераторов в стеке не поддерживается. Напоминаем, что метод `pop` *не возвращает* удаленное значение. Чтобы прочитать значение на вершине стека, используется метод `top`. Ниже приведен пример работы со стеком. Программа вводит из файла числа и выводит их в обратном порядке¹:

```

int main() {
    ifstream in ( "inpnun.txt" );
    stack<int> s;

```

продолжение ➤

¹ В дальнейших примерах мы будем опускать директивы `#include` и объявление `using namespace std` для экономии места, то есть стоимости книги.

```
int x;
while ( in >> x ) s.push( x );
while ( !s.empty() ) {
    cout << s.top() << ' ';
    s.pop();
}
}
```

Объявление `stack<int> s` создает стек на базе двусторонней очереди (по умолчанию). Если это нас по каким-то причинам не устраивает и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом:

```
stack<int, list<int> > s; // Не забывайте о пробеле между угловыми скобками > >
```

Очередь

Шаблонный класс `queue` (заголовочный файл `<queue>`) является адаптером, который может быть реализован на основе двусторонней очереди (по умолчанию) или списка. Класс `vector` в качестве прототипа не подходит, так как в нем нет выборки из начала контейнера. Очередь использует для проталкивания данных один конец, а для выталкивания — другой. В соответствии с этим ее интерфейс образуют методы из табл. 15.6.

Таблица 15.6. Интерфейс класса `queue`

Методы класса <code>queue</code>	Методы класса-прототипа
<code>push</code>	<code>push_back</code>
<code>pop</code>	<code>pop_front</code>
<code>front</code>	<code>front</code>
<code>back</code>	<code>back</code>
<code>empty</code>	<code>empty</code>
<code>size</code>	<code>size</code>

Очередь с приоритетами

Шаблонный класс `priority_queue` (заголовочный файл `<queue>`) поддерживает такие же операции, что и класс `queue`, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. *Очередь с приоритетами* отличается от обычной тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера. Если очередь с приоритетами организуется для объектов класса, определенного программистом, в этом классе следует определить операцию `<`. Пример работы с очередью с приоритетами приведен в листинге 15.3.

Листинг 15.3. Пример работы с очередью с приоритетами

```
int main() {
    priority_queue <int> P;
```



```

P.push( 17 ); P.push( 5 ); P.push( 400 ); P.push( 2500 ); P.push( 1 );
while ( !P.empty() ) {
    cout << P.top() << ' ';
    P.pop();
}
// Результат выполнения программы: 2500 400 17 5 1

```

Использование алгоритмов

Вернемся к изучению алгоритмов. Не забывайте включать заголовочный файл `<algorithm>` и добавлять определение функции `print`, если она используется.

Алгоритмы `count` и `find`

Алгоритм `count` подсчитывает количество вхождений в контейнер (или его часть) значения, заданного его третьим аргументом. Алгоритм `find` выполняет поиск заданного значения и возвращает итератор на самое первое вхождение этого значения. Если значение не найдено, возвращается итератор, соответствующий возврату метода `end`. В листинге 15.4 показано использование этих алгоритмов.

Листинг 15.4. Пример использования алгоритмов `count` и `find`

```

int main() {
    int arr[] = { 1, 2, 3, 4, 5, 2, 6, 2, 7 };
    int n = sizeof( arr ) / sizeof( int );
    vector<int> v1( arr, arr + n );
    int value = 2; // искомая величина
    int how_much = count( v1.begin(), v1.end(), value ); // 1
    cout << how_much << endl; // вывод: 3
    list<int> loc_list; // список позиций искомой величины
    vector<int>::iterator location = v1.begin();
    while ( 1 ) {
        location = find( location, v1.end(), value ); // 2
        if ( location == v1.end() ) break;
        loc_list.push_back( location - v1.begin() );
        location++;
    }
    print( loc_list ); // функция из листинга 15.2 // вывод: 1 5 7
}

```

Вектор `v1` создается, инициализируясь значениями массива `arr`. Затем с помощью алгоритма `count` подсчитывается количество вхождений в вектор значения `value`, равного двум. В цикле `while` выясняется, в каких позициях вектора размещена эта величина. Первый аргумент алгоритма `find` (переменная `location`) первоначально, перед входом в цикл, принимает значение итератора, указывающего на нулевой элемент контейнера (элементы вектора нумеруются с нуля). Затем `location` получает значение итератора, указывающего на найденный элемент.

Если поиск завершился успешно, то, во-первых, вычисляется позиция найденного элемента как разность значений `location` и адреса нулевого элемента и полученное

значение заносится в список `loc_list`. Во-вторых, итератор `location` сдвигается операцией инкремента на следующую позицию в контейнере, чтобы обеспечить на следующей итерации цикла продолжение поиска в оставшейся части контейнера. Если поиск завершился неудачей, то `break` приведет к выходу из цикла.

Алгоритмы `count_if` и `find_if`

Алгоритмы `count_if` и `find_if` отличаются от алгоритмов `count` и `find` тем, что их третьим аргументом является некоторый *предикат*. *Предикат* — это функция или функциональный объект, возвращающие значение типа `bool`. Например, если в листинге 15.4 добавить определение глобальной функции

```
bool isMyValue( int x ) { return ( ( x > 2 ) && ( x < 5 ) ); }
```

и заменить инструкцию с вызовом `count` (оператор, помеченный цифрой 1) на:

```
int how_much = count_if( v1.begin(), v1.end(), isMyValue );
```

то программа определит, что контейнер содержит два числа, значение которых больше двух, но меньше пяти.

Аналогично, замена инструкции с вызовом `find` (*оператор 2*) на:

```
location = find_if( location, v1.end(), isMyValue );
```

приведет к наполнению списка `loc_list` значениями 2 и 3 (номера позиций вектора `v1`, в которых находятся числа, удовлетворяющие предикату `isMyValue`).

Алгоритм `for_each`

Этот алгоритм позволяет выполнить некоторое действие над каждым элементом диапазона `[first, last)`. Чтобы определить, какое именно действие должно быть выполнено, нужно написать соответствующую функцию с одним аргументом типа `T` (`T` — тип данных, содержащихся в контейнере). Функция не имеет права модифицировать данные в контейнере, но может их использовать в своей работе. Имя этой функции передается алгоритму третьим аргументом. Например, в следующей программе `for_each` используется для перевода всех значений массива из дюймов в сантиметры и вывода их на экран:

```
void InchToCm( double inch ) { cout << ( inch * 2.54 ) << ' '; }
int main() {
    double inches[] = {0.5, 1.0, 1.5, 2.0, 2.5};
    for_each( inches, inches + 5, InchToCm );
}
```

Алгоритм `search`

Некоторые алгоритмы оперируют одновременно двумя контейнерами. Таков и алгоритм `search`, находящий первое вхождение в первую последовательность `[first1, last1)` второй последовательности `[first2, last2)`. Пример приведен в листинге 15.5.

Листинг 15.5. Пример использования алгоритма `search`

```
int main() {
    int arr[] = { 11, 77, 33, 11, 22, 33, 11, 22, 55 };
    int pattern[] = { 11, 22, 33 };
    int* ptr = search( arr, arr + 9, pattern, pattern + 3 );
    if ( ptr == arr + 9 ) cout << "Pattern not found" << endl;
    else cout << "Found at position " << ( ptr - arr ) << endl;
    list<int> lst( arr, arr + 9 );
    list<int>::iterator ifound;
    ifound = search( lst.begin(), lst.end(), pattern, pattern + 3 );
    if ( ifound == lst.end() ) cout << "Pattern not found" << endl;
    else cout << "Found." << endl;
}
```

Результат выполнения программы:

```
Found at position 3
Found.
```

Отметим, что список не поддерживает произвольного доступа к своим элементам, а значит, не допускает операций "+" и "-" с итераторами. Поэтому мы можем только зафиксировать факт вхождения последовательности `pattern` в контейнер `lst`.

Алгоритм `sort`

Алгоритм сортировки можно применять только для тех контейнеров, которые обеспечивают произвольный доступ к элементам, — этому требованию удовлетворяют *массив*, *вектор* и *двусторонняя очередь*, но не удовлетворяет *список*. В связи с этим класс `list` содержит метод `sort`, решающий задачу сортировки.

Алгоритм `sort` имеет две сигнатуры:

```
template<class RandomAccessIt>
void sort( RandomAccessIt first, RandomAccessIt last );
template<class RandomAccessIt>
void sort( RandomAccessIt first, RandomAccessIt last, Compare comp );
```

Первая форма алгоритма обеспечивает сортировку элементов из диапазона `[first, last)`, причем для упорядочения по умолчанию используется операция `<`, которая должна быть определена для типа `T` (тип данных, содержащихся в контейнере). То есть сортировка по умолчанию выполняется по возрастанию значений, например:

```
int main() {
    double arr[6] = { 2.2, 0.0, 4.4, 1.1, 3.3, 1.1 };
    vector<double> v1( arr, arr + 6 );
    sort( v1.begin(), v1.end() );
    print( v1 );           // Функция из листинга 15.2
}                         // Результат выполнения программы:  0 1.1 1.1 2.2 3.3 4.4
```

Вторая форма алгоритма позволяет задать произвольный критерий упорядочения. Для этого нужно передать через третий аргумент соответствующий предикат,

то есть функцию или функциональный объект, возвращающие значение типа `bool`. Использование функции в качестве предиката было показано при описании алгоритмов `count_if` и `find_if`. Использованию функциональных объектов посвящен следующий раздел.

Функциональные объекты

На семинаре 12 было показано, как можно использовать функциональные объекты для настройки шаблонных классов, поэтому рекомендуем вам еще раз просмотреть этот материал. *Функциональным объектом* называется объект некоторого класса, для которого определена единственная операция вызова функции `operator()`.

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++. Они возвращают значение типа `bool`, то есть являются предикатами (табл. 15.7).

Таблица 15.7. Предикаты стандартной библиотеки

Операция	Эквивалентный предикат (функциональный объект)	Операция	Эквивалентный предикат (функциональный объект)
<code>==</code>	<code>equal_to</code>	<code><</code>	<code>less</code>
<code>!=</code>	<code>not_equal_to</code>	<code>>=</code>	<code>greater_equal</code>
<code>></code>	<code>greater</code>	<code><=</code>	<code>less_equal</code>

Очевидно, что при подстановке в качестве аргумента алгоритма требуется инстанцирование этих шаблонов, например: `equal_to<int>()`.

Вернемся к предыдущей программе сортировки вектора `v1`. Для упорядочивания по убыванию значений его элементов заменим вызов `sort` на следующий:

```
sort( v1.begin(), v1.end(), greater<double>() );
```

Несколько сложнее дело обстоит, когда сортировка выполняется для контейнера с объектами пользовательского класса. В этом случае программисту нужно самому позаботиться о наличии в классе предиката, задающего сортировку по умолчанию, а также (при необходимости) определить функциональные классы, объекты которых позволяют изменять настройку алгоритма `sort`.

В листинге 15.6 показаны варианты вызова алгоритма `sort` для вектора `men`, содержащего объекты класса `Man`. В классе `Man` определен предикат (операция `operator<()`), благодаря которому сортировка по умолчанию будет происходить по возрастанию значений поля `name`. Кроме этого, определен функциональный класс `LessAge`, использование которого позволяет выполнить сортировку по возрастанию значений поля `age`.

Листинг 15.6. Пример использования функциональных объектов

```
class Man {
public:
    Man ( string _name, int _age ) : name( _name ), age( _age ) {}
    // предикат, задающий сортировку по умолчанию:
    bool operator< ( const Man& m ) const { return name < m.name; }
```

```

    friend ostream& operator<< ( ostream&, const Man& );
    friend struct LessAge;
private:
    string name;
    int age;
};
ostream& operator<<( ostream& os, const Man& m ) {
    return os << endl << m.name << ",\t age: " << m.age;
}
struct LessAge {    // ----- Функциональный класс для сравнения по возрасту
    bool operator() ( const Man& a, const Man& b ) { return a.age < b.age; }
};
int main() {
    Man ar[] = {
        Man( "Mary Poppins", 36 ),    Man( "Count Basie", 70 ),
        Man( "Duke Ellington", 90 ), Man( "Joy Amore", 18 )
    };
    int size = sizeof( ar ) / sizeof( Man );
    vector<Man> men( ar, ar + size );
    sort( men.begin(), men.end() );           // Сортировка по имени (по умолчанию)
    print( men );                             // функция из листинга 15.2
    sort( men.begin(), men.end(), LessAge() ); // Сортировка по возрасту
    print( men );
}

```

Обратные итераторы

Эта разновидность итераторов (`reverse_iterator`) удобна для прохода по контейнеру от конца к началу. Например, для вывода содержимого контейнера `vector<double> v1` в обратном порядке можно написать

```

vector<double>::reverse_iterator ri;
ri = v1.rbegin();
while ( ri != v1.rend() ) cout << *ri++ << ' ';

```

Операция инкремента для такого итератора перемещает указатель на предыдущий элемент контейнера.

Итераторы вставки и алгоритм `copy`

Мы можем использовать алгоритм `copy` для копирования элементов одного контейнера в другой, причем источником может быть, например, вектор, а приемником — список, как показывает следующий фрагмент:

```

int main() {
    int a[4] = { 10, 20, 30, 40 };
    vector<int> v( a, a + 4 );
    list<int> L( 4 );           // список из 4 элементов
    copy( v.begin(), v.end(), L.begin() );
    print( L );                // функция из листинга 15.2
}

```

Алгоритм `copy` при таком использовании, как в этом примере, работает в *режиме замещения*. Это означает, что i -й элемент контейнера-источника замещает i -й элемент контейнера-приемника (это напоминает режим замены при вводе текста в текстовом редакторе). Этот же алгоритм может работать и в *режиме вставки*, если третьим аргументом использовать так называемый *итератор вставки*.

Итераторы вставки `front_inserter`, `back_inserter`, `inserter` предназначены для добавления новых элементов в начало, конец или произвольное место контейнера. Покажем использование этих итераторов на следующем примере.

```
int main() {
    int a[4] = { 40, 30, 20, 10 };      vector<int> va( a, a + 4 );
    int b[3] = { 80, 90, 100 };         vector<int> vb( b, b + 3 );
    int c[3] = { 50, 60, 70 };         vector<int> vc( c, c + 3 );
    list<int> L;                        // пустой список
    copy( va.begin(), va.end(), front_inserter(L) );           // 1
    print( L );                                                  // функция из листинга 15.2
    copy( vb.begin(), vb.end(), back_inserter(L) );            // 2
    print( L );
    list<int>::iterator from = L.begin();
    advance( from, 4 );
    copy( vc.begin(), vc.end(), inserter( L, from ) );          // 3
    print( L );
}
```

Результат выполнения программы:

```
10 20 30 40
10 20 30 40 80 90 100
10 20 30 40 50 60 70 80 90 100
```

В *операторе 1* выполняется копирование (вставка) вектора `va` в список `L`. Итератор вставки `front_inserter` обеспечивает вставку очередного элемента вектора `va` в начало списка, поэтому порядок элементов в списке изменяется на обратный.

В *операторе 2* элементы вектора `vb` пересылаются в конец списка `L` благодаря итератору вставки `back_inserter` (порядок копируемых элементов не меняется).

В *операторе 3* вектор `vc` копируется в заданное итератором `from` место списка `L`, а именно, после четвертого элемента списка. Чтобы определить нужное значение итератора `from`, мы предварительно устанавливаем его в начало списка, а затем обеспечиваем приращение на 4 вызовом функции `advance`.

Алгоритм merge

Алгоритм `merge` выполняет слияние отсортированных последовательностей для любого типа последовательного контейнера, более того — все три участника алгоритма могут представлять различные контейнерные типы. Например, вектор `a` и массив `b` могут быть слиты в список `c`:

```
int main() {
    int arr[5] = { 2, 3, 8, 20, 25 };
    vector<int> a( arr, arr + 5 );
```

```

int b[6] = { 7, 9, 23, 28, 30, 33 };
list<int> c;                                     // Список с начала пуст
merge( a.begin(), a.end(), b, b + 6, back_inserter(c) );
print( c );                                     // функция из листинга 15.2
}                                                // Результат выполнения программы: 2 3 7 8 9 20 23 25 28 30 33

```

Использование ассоциативных контейнеров

В ассоциативных контейнерах элементы не выстроены в линейную последовательность. Они организованы в более сложные структуры, что дает большой выигрыш в скорости поиска. Поиск выполняется с помощью *ключа*, обычно представляющего собой одно число или строку. Рассмотрим две основные категории ассоциативных контейнеров в STL: множества и словари (отображения).

В *множестве* (set) хранятся объекты, упорядоченные по некоторому ключу, являющемуся атрибутом самого объекта. Например, множество может хранить объекты класса Man, упорядоченные в алфавитном порядке по значению ключевого поля name. Если в множестве хранятся значения одного из встроенных типов, например int, то ключом является сам элемент.

Словарь (map) можно представить себе как своего рода таблицу из двух столбцов, в первом из которых хранятся объекты, содержащие ключи, а во втором — объекты, содержащие значения. Похожая организация данных рассматривалась нами в задаче 12.1 (шаблонный класс для разреженных массивов).

И в множествах, и в словарях все ключи являются уникальными (только одно значение соответствует ключу). *Мультимножества* (multiset) и *мультисловари* (multimap) аналогичны своим родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Ассоциативные контейнеры имеют много общих методов с последовательными контейнерами, но некоторые методы и алгоритмы характерны только для них.

Множества

Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение «меньше»:

```

template <class Key, class Compare = less<Key> >
class set{ /* ... */ };

```

Таким образом, если объявить некоторое множество set<int> s1 с опущенным вторым параметром шаблона, то по умолчанию для упорядочения членов множества будет использован предикат less<int>. Точно так же можно опустить второй параметр при объявлении множества set<MyClass> s2, если в классе MyClass определена операция operator<().

Для использования контейнеров типа set необходимо подключить заголовочный файл <set>. Имеется три способа определить объект типа set:

```

set<int> set1;                                     // создается пустое множество
int a[5] = { 1, 2, 3, 4, 5 };

```

продолжение ➤

```
set<int> set2( a, a + 5 );           // инициализация копированием массива
set<int> set3( set2 );               // инициализация другим множеством
```

Для вставки элементов в множество можно использовать метод `insert`, для удаления — метод `erase`. Также к множествам применимы общие для всех контейнеров методы, указанные в табл. 15.2.

Во всех ассоциативных контейнерах есть метод `count`, возвращающий количество объектов с заданным ключом. Так как и в множествах, и в словарях все ключи уникальны, то метод `count` возвращает либо 0, если элемент не обнаружен, либо 1.

Для множеств библиотека содержит некоторые специальные алгоритмы, в частности, реализующие традиционные теоретико-множественные операции. Эти алгоритмы перечислены ниже.

Алгоритм `includes` выполняет проверку включения одной последовательности в другую. Результат равен `true` в случае, когда каждый элемент последовательности `[first2, last2)` содержится в последовательности `[first1, last1)`.

Алгоритм `set_intersection` создает отсортированное *пересечение множеств*, то есть множество, содержащее только те элементы, которые одновременно входят и в первое, и во второе множество.

Алгоритм `set_union` создает отсортированное *объединение множеств*, то есть множество, содержащее элементы первого и второго множеств без повторяющихся элементов. Использование этих алгоритмов показано в листинге 15.7.

Листинг 15.7. Пример работы с множествами

```
int main() {
    const int N = 5;
    string s1[N] = { "Bill", "Jessica", "Ben", "Mary", "Monica" };
    string s2[N] = { "Sju", "Monica", "John", "Bill", "Sju" };
    typedef set<string> SetS;
    SetS A( s1, s1 + N );
    SetS B( s2, s2 + N );
    print( A ); print( B );
    SetS prod, sum;
    set_intersection( A.begin(), A.end(), B.begin(), B.end(),
                     inserter( prod, prod.begin() ) );
    print( prod );
    set_union( A.begin(), A.end(), B.begin(), B.end(),
              inserter( sum, sum.begin() ) );
    print( sum );
    if ( includes( A.begin(), A.end(), prod.begin(), prod.end() ) )
        cout << "Yes" << endl;
    else cout << "No" << endl;
}
```

Результат выполнения программы:

```
Ben Bill Jessica Mary Monica
Bill John Monica Sju
```


Bill Monica
 Ben Bill Jessica John Mary Monica Sju
 Yes

Словари

В определении класса `map` используется тип `pair`, который описан в заголовочном файле `<utility>` следующим образом:

```
template <class T1, class T2> struct pair{
    T1 first;
    T2 second;
    pair( const T1& x, const T2& y );
    . . .
};
```

Шаблон `pair` имеет два параметра, представляющих собой типы элементов пары. Первый элемент пары имеет имя `first`, второй — `second`. В этом же файле определены шаблонные операции `==`, `!=`, `<`, `>`, `<=`, `>=` для двух объектов типа `pair`.

Шаблон словаря имеет три параметра: тип ключа, тип элемента и тип функционального объекта, определяющего отношение «меньше»:

```
template <class Key, class T, class Compare = less<Key> >
class map {
public:
    typedef pair <const Key, T> value_type;
    explicit map ( const Compare& comp = Compare() );
    map( const value_type* first, const value_type* last,
        const Compare& comp = Compare() );
    map( const map <Key, T, Compare>& x );
    . . .
};
```

Тип элементов словаря `value_type` определяется как пара элементов типа `Key` и `T`. Первый конструктор класса `map` создает пустой словарь. Второй — создает словарь и записывает в него элементы, определяемые диапазоном `[first, last)`. Третий конструктор является конструктором копирования.

Для доступа к элементам по ключу определена операция `[]`:

```
T& operator[]( const Key & x );
```

С помощью нее можно не только получать значения элементов, но и добавлять в словарь новые. Для использования контейнеров типа `map` необходимо подключить заголовочный файл `<map>`.

Задача 15.2. Формирование частотного словаря

Написать программу формирования частотного словаря появления отдельных слов в некотором тексте. Исходный текст читается из файла `prose.txt`, результат (частотный словарь) записывается в файл `freq_map.txt`.

Текст программы приведен в листинге 15.8.

Листинг 15.8. Формирование частотного словаря

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <map>
#include <set>
#include <string>
using namespace std;
int main() {
    char punct[6] = { '.', ',', '?', '!', ':', ';' };
    set<char> punctuation( punct, punct + 6 );
    ifstream in( "prose.txt" );
    if ( !in ) { cerr << "File not found\n"; exit( 1 ); }
    map<string, int> wordCount;
    string s;
    while ( in >> s ) {
        int n = s.size();
        if ( punctuation.count( s[n - 1] ) ) s.erase( n - 1, n );
        ++wordCount[s];
    }
    ofstream out( "freq_map.txt" );
    map<string, int>::const_iterator it = wordCount.begin();
    for ( it; it != wordCount.end(); ++it )
        out << setw( 20 ) << left << it->first
            << setw( 4 ) << right << it->second << endl;
}
```

Определяя объект `wordCount` как словарь типа `map<string, int>`, мы показываем намерение связать каждое прочитанное слово с целочисленным счетчиком.

В цикле `while` разворачиваются следующие события.

- ☐ В строку `s` пословно считываются данные из входного файла.
- ☐ Определяется длина `n` строки `s`.
- ☐ С помощью метода `count` проверяется, принадлежит ли последний символ строки `s` множеству `punctuation`, содержащему знаки препинания, которыми может завершаться слово. Если да, то последний символ удаляется из строки (метод `erase`).
- ☐ Заслуживает особого внимания лаконичная инструкция `++wordCount[s]`. Здесь мы как бы «заглядываем» в объект `wordCount`, используя только что считанное слово в качестве ключа. Результат выражения `wordCount[s]` представляет собой некоторое целочисленное значение, обозначающее, сколько раз слово `s` уже встречалось ранее. Затем операция инкремента увеличивает это целое значение на единицу. А что будет, если мы встречаем некоторое слово в первый раз? Если в словаре нет элемента с таким ключом, он будет создан с инициализацией поля типа `int` значением по умолчанию, то есть нулем. Следовательно, после операции инкремента это значение будет равно единице.

Завершив считывание входных данных и формирование словаря `wordCount`, мы должны вывести в выходной файл `freq_map.txt` значения обнаруженных слов и соответствующих им счетчиков. Вывод результатов реализуется здесь практически так же, как и для последовательных контейнеров — с помощью соответствующего итератора. Однако есть тонкость, связанная с тем, что при разыменовании итератора `map`-объекта мы получаем значение, которое имеет тип `pair`, соответствующий данному `map`-объекту. Так как `pair` — это структура, доступ к ее полям через «указатель» `it` выполняется посредством выражений `it->first`, `it->second`.

Рассмотрим теперь более сложную задачу, чтобы продемонстрировать применение принципов ООП на практике и удобства, которые дает использование STL.

Задача 15.3. Морской бой

Написать программу, реализующую упрощенную версию игры «Морской бой» между двумя игроками: пользователь и компьютер. Упрощения: все корабли размещаются только вертикально; размещение кораблей — случайное у обоих игроков.

Тем, у кого было тяжелое детство, израненное компьютером, напомним правила:

- ❑ Имеется два игровых поля: «свое» и «противника», каждое 10×10 клеток. У каждого игрока по 10 кораблей: один четырехпалубный (состоящий из четырех клеток), два трехпалубных (из трех клеток), три двухпалубных (из двух), четыре однопалубных (из одной клетки). При расстановке кораблей они не должны касаться друг друга (находиться в соседних клетках).
- ❑ Каждый игрок видит размещение кораблей на своем игровом поле, но не имеет информации о размещении кораблей на поле противника.
- ❑ После расстановки кораблей игроки начинают «стрелять» друг в друга. Для этого стреляющий выбирает клетку на поле противника и объявляет ему ее координаты (A1, E5 и т. д.). Противник смотрит на своем поле, находится ли по указанным координатам его корабль, и сообщает результат выстрела:
 - *промах* — на данной клетке нет корабля противника;
 - *ранен (поврежден)* — на данной клетке есть корабль противника с хотя бы еще одной непораженной клеткой (палубой);
 - *убит* — на данной клетке есть корабль противника, и все его клетки (палубы) уже поражены.
- ❑ В случае попадания в корабль противника игроку дается право на внеочередной выстрел, в противном случае ход переходит к противнику.
- ❑ Стрельба ведется до тех пор, пока у одного из игроков не окажутся «убитыми» все корабли (он признается проигравшим, а его противник — победителем).

Так как мы пишем консольное приложение, доступные нам графические средства сильно ограничены — это текстовые символы и символы псевдографики. Примем решение, что после некоторого хода играющих картинка в консольном окне будет иметь примерно такой вид, какой показан на рис.15.1.

Изображенные на рисунке игровые поля «Мой флот» и «Флот неприятеля» отображают текущее состояние игры со стороны пользователя. Изначальное размещение

кораблей на поле пользователя — в клетках, помеченных символом «заштрихованный прямоугольник» (символ 176 в кодовой таблице ср866/MS DOS). Символом «.» (точка) обозначены свободные клетки, по которым еще ни разу не стреляли, символом «o» — промахи стреляющих, символом «X» — пораженные клетки (палубы) кораблей. Пробегами обозначены клетки, в которых по правилам уже не могут находиться корабли противника. Эти «мертвые зоны» выявляются после гибели очередного корабля.

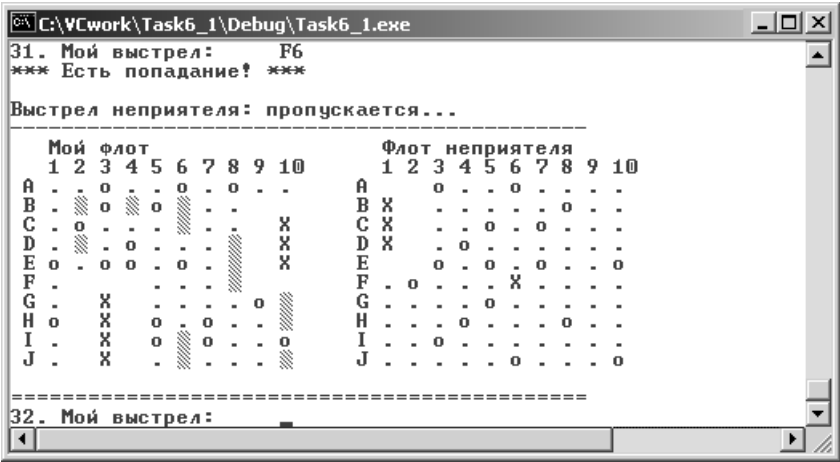


Рис. 15.1. Возможный вид консольного окна после i-го хода (i = 31)

После сделанных разъяснений мы можем приступить к решению задачи, и начнем, как всегда, с выявления понятий (классов) и их взаимосвязей.

Итак, имеется два игрока: пользователь (User) и компьютер, выступающий в роли робота (Robot). Каждый игрок «управляет» своим собственным флотом, поэтому логично создать два класса: UserNavy (флот пользователя) и RobotNavy (флот робота). Очевидно, что они обладают различным поведением — например, метод FireOff (выстрел по неприятелю) в первом классе должен пригласить пользователя ввести координаты выстрела, а во втором классе — автоматически сформировать координаты выстрела, сообразуясь с искусственным интеллектом робота. В то же время в этих классах есть и общие атрибуты, например, игровые поля (свое и неприятеля), корабли своего флота и т. д. Поэтому выделим все общие поля и методы в базовый класс Navy, который будут наследовать классы UserNavy и RobotNavy.

Каждый флот состоит из кораблей, отсюда вытекает потребность в классе Ship, объекты которого инкапсулируют такую информацию, как координаты размещения корабля, имя корабля, общее количество палуб, количество неповрежденных палуб.

Для описания размещения кораблей воспользуемся классом Rect, который позволяет задать любой прямоугольник в двумерном дискретном пространстве. Конечно, наши прямоугольники в формализме двумерного дискретного пространства

вырождаются в линию, но такое описание удобно для единообразного представления как вертикально, так и горизонтально размещенных кораблей¹.

Игровое поле (двумерное дискретное пространство) состоит из клеток (точек двумерного пространства), для представления которых используем класс `Cell`.

Наконец, игроки должны обмениваться информацией (координаты и результаты очередного выстрела). Для моделирования процесса обмена информацией создадим класс `Space`, поля которого будут использоваться как глобальные переменные, поэтому они должны быть статическими, а сам класс — базовым для класса `Navy`.

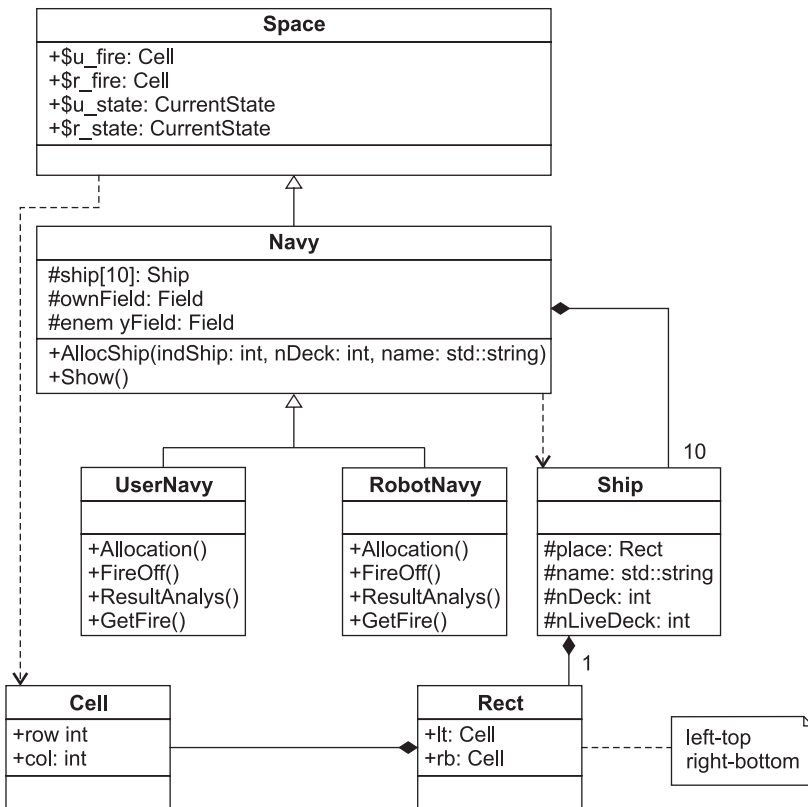


Рис. 15.2. Диаграмма классов для задачи 15.3

На рис. 15.2 показана диаграмма классов, обобщающая наши рассуждения по составу и взаимоотношениям классов для решения рассматриваемой задачи. Текст программы приводится в листинге 15.9.

¹ Хотя условие имеет упрощающее ограничение «все корабли размещаются вертикально», при проектировании структур данных рекомендуется учитывать возможность последующей модификации программы, снимающей это ограничение.

Листинг 15.9. «Морской бой»

```

//////////////////////////////////// Ship.h
#ifndef SHIP_H
#define SHIP_H
#include <set>
#include <map>
#include <string>
#define N 10 // размер поля для размещения флота (N * N клеток)
struct Cell;
typedef std::set<Cell> CellSet; // множество клеток
struct Cell { // ----- Клетка (ячейка) на игровом поле
    Cell( int _r = 0, int _c = 0 ) : row( _r ), col( _c ) {}
    bool InSet( const CellSet& ) const; // определяет
    // принадлежность клетки множеству типа CellSet
    bool operator<( const Cell& ) const;
    int row; // ряд
    int col; // колонка
};
struct Rect { // --- Прямоугольная область (размещение кораблей и их "оболочек")
    Rect() {}
    Rect( Cell _lt, Cell _rb ) : lt( _lt ), rb( _rb ) { FillCset(); }
    void FillCset(); // заполнить cset клетками данной области
    bool Intersect( const CellSet& cs ) const; // определить наличие
    // непустого пересечения прямоугольника с множеством cs
    Cell lt; // left-top клетка
    Cell rb; // right-bottom клетка
    CellSet cset; // множество клеток, принадлежащих прямоугольнику
};
class Ship { // ----- Класс Ship (для представления корабля)
    friend class UserNavy;
    friend class RobotNavy;
public:
    Ship() : nDeck( 0 ), nLiveDeck( 0 ) {}
    Ship( int, std::string, Rect );
protected:
    Rect place; // координаты размещения
    std::string name; // имя корабля
    int nDeck; // количество палуб
    int nLiveDeck; // количество неповрежденных палуб
};
#endif /* SHIP_H */
//////////////////////////////////// Ship.cpp
#include <string>
#include <algorithm>
#include "Ship.h"
using namespace std;
bool Cell::InSet( const CellSet& cs ) const { // ----- Класс Cell
    return ( cs.count( Cell( row, col ) ) > 0 );
}

```

```

bool Cell::operator<( const Cell& c ) const {
    return ( ( row < c.row ) || ( ( row == c.row ) && ( col < c.col ) ) );
}
void Rect::FillCset() {          // ----- Класс Rect
    for ( int i = lt.row; i <= rb.row; i++ )
        for ( int j = lt.col; j <= rb.col; j++ )
            cset.insert( Cell( i, j ) );
}
bool Rect::Intersect( const CellSet& cs ) const {
    CellSet common_cell;
    set_intersection( cset.begin(), cset.end(), cs.begin(), cs.end(),
        inserter( common_cell, common_cell.begin() ) );
    return ( common_cell.size() > 0 );
}
Ship::Ship( int _nDeck, string _name, Rect _place ) :    // ----- Класс Ship
    place( _place ), name( _name ), nDeck( _nDeck ), nLiveDeck( _nDeck ) {}
////////////////////////////////////////////////////////////////// Navy.h
#include "Ship.h"
#define DECK 176          // исправная клетка-палуба
#define DAMAGE 'X'        // разрушенная клетка-палуба
#define MISS 'o'          // пустая клетка, в к-рую упал снаряд
typedef unsigned char Field[N][N];    // игровое поле
typedef std::map<Cell, int> ShipMap;   // словарь
// ассоциаций "клетка - индекс корабля"
enum CurrentState { Miss, Damage, Kill }; // результат попадания в цель
struct Space { // ----- Класс Space - информационное пространство для обмена
public:
    static Cell u_fire;          // огонь от пользователя
    static Cell r_fire;          // огонь от робота (компьютера)
    static CurrentState u_state; // состояние пользователя
    static CurrentState r_state; // состояние робота
    static int step;
};
class Navy : public Space { //----- Базовый класс Navy
public:
    Navy();
    void AllocShip( int, int, std::string ); // разместить корабль
    void Show() const;                       // показать поля ownField и enemyField
    int GetInt();                             // ввод целого числа
    bool IsLive() { return ( nLiveShip > 0 ); } // мы еще живы?
    Rect Shell( Rect ) const; /* вернуть "оболочку" для заданного прямоугольника
(сам прямоугольник плюс пограничные клетки) */
    void AddToVetoSet( Rect );                // добавить клетки прямоугольника
                                           // в множество vetoSet.
protected:
    Ship ship[10];    // корабли флота
    Field ownField;    // мое игровое поле
    Field enemyField;  // игровое поле неприятеля
    ShipMap shipMap;   // словарь ассоциаций "клетка - индекс корабля"

```

продолжение ➤

Листинг 15.9 (продолжение)

```

    CellSet vetoSet; // множество "запрещенных" клеток
    CellSet crushSet; // множество "уничтоженных" клеток
    int nLiveShip;    // количество боеспособных кораблей
};
class UserNavy : public Navy { //----- Класс UserNavy
public:
    UserNavy() { Allocation(); }
    void Allocation();
    void FireOff();           // выстрел по неприятелю
    void ResultAnalys();      // анализ результатов выстрела
    void GetFire();           // "прием" огня противника
    void FillDeadZone( Rect r, Field& ); // заполнить
                                   // пробелами пограничные клетки для r
};
class RobotNavy : public Navy { // ----- Класс RobotNavy
public:
    RobotNavy();
    void Allocation();
    void FireOff();           // выстрел по неприятелю
    void ResultAnalys();      // анализ результатов выстрела
    void GetFire();           // "прием" огня противника
private:
    bool isCrushContinue; // предыдущий выстрел был успешным
    bool upEmpty;         // у поврежденного корабля противника
                           // нет "живых" клеток в верхнем направлении
};
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Navy.cpp
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <algorithm>
#include "Navy.h"
using namespace std;
Cell Space::u_fire;
Cell Space::r_fire;
CurrentState Space::u_state = Miss;
CurrentState Space::r_state = Miss;
int Space::step = 1;
// Функция gap(n) возвращает строку из n пробелов
string gap( int n ) { return string( n, ' ' ); }
Navy::Navy() : nLiveShip( 10 ) { // ----- Класс Navy
    for ( int i = 0; i < N; i++ ) // Заполняем игровые поля символом "точка"
        for ( int j = 0; j < N; j++ ) {
            ownField[i][j] = '.';
            enemyField[i][j] = '.';
        }
}
Rect Navy::Shell( Rect r ) const {
    Rect sh( r );

```



```

    sh.lt.row = ( --sh.lt.row < 0 ) ? 0 : sh.lt.row;
    sh.lt.col = ( --sh.lt.col < 0 ) ? 0 : sh.lt.col;
    sh.rb.row = ( ++sh.rb.row > ( N - 1 ) ) ? ( N - 1 ) : sh.rb.row;
    sh.rb.col = ( ++sh.rb.col > ( N - 1 ) ) ? ( N - 1 ) : sh.rb.col;
    return sh;
}

void Navy::AddToVetoSet( Rect r ) {
    for ( int i = r.lt.row; i <= r.rb.row; i++ )
        for ( int j = r.lt.col; j <= r.rb.col; j++ )
            vetoSet.insert( Cell( i, j ) );
}

void Navy::AllocShip( int indShip, int nDeck, string name ) {
    int i, j;
    Cell lt, rb;
    // Генерация случайно размещенной начальной клетки корабля с учетом
    // недопустимости "пересечения" нового корабля с множеством клеток vetoSet
    while( 1 ) {
        lt.row = rand( ) % ( N + 1 - nDeck );
        lt.col = rb.col = rand( ) % N;
        rb.row = lt.row + nDeck - 1;
        if ( !Rect( lt, rb ).Intersect( vetoSet ) ) break;
    }
    // Сохраняем данные о новом корабле
    ship[indShip] = Ship( nDeck, name, Rect( lt, rb ) );
    // Заносим новый корабль на игровое поле (символ DECK).
    // Добавляем соответствующие элементы в словарь ассоциаций
    for ( i = lt.row; i <= rb.row; i++ )
        for ( j = lt.col; j <= rb.col; j++ ) {
            ownField[i][j] = DECK;
            shipMap[Cell( i, j )] = indShip;
        }
    // Добавляем в множество vetoSet клетки нового корабля
    // вместе с пограничными клетками
    AddToVetoSet( Shell( Rect( lt, rb ) ) );
}

void Navy::Show() const {
    char rowName[10] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'};
    string colName( "1 2 3 4 5 6 7 8 9 10" );
    int i, j;
    cout << "-----\n";
    cout << gap( 3 ) << "Мой флот" << gap( 18 ) << "Флот неприятеля" << endl;
    cout << gap( 3 ) << colName << gap( 6 ) << colName << endl;
    for ( i = 0; i < N; i++ ) {
        string line = gap( 1 ) + rowName[i]; // Own
        for ( j = 0; j < N; j++ )
            line += gap( 1 ) + ( char )ownField[i][j];
        line += gap( 5 ) + rowName[i]; // Enemy
        for ( j = 0; j < N; j++ )
            line += gap( 1 ) + ( char )enemyField[i][j];
    }
}

```

продолжение ➤

Листинг 15.9 (продолжение)

```

        cout << line << endl;
    }
    cout << endl << "=====\\n";
    cout << step << ". " << "Мой выстрел: ";
    step++;
}

int Navy::GetInt() {
    int value;
    while ( true ) {
        cin >> value;
        if ( '\\n' == cin.peek() ) { cin.get(); break; }
        else {
            cout << "Повторите ввод колонки (ожидается целое число):" << endl;
            cin.clear();
            while ( cin.get() != '\\n' ) {};
        }
    }
    return value;
}

void UserNavy::Allocation() { // -----Класс UserNavy
    srand( ( unsigned )time( NULL ) );
    AllocShip(0, 4, "Авианосец 'Варяг'"); AllocShip(1, 3, "Линкор 'Муромец'");
    AllocShip(2, 3, "Линкор 'Никитич'"); AllocShip(3, 2, "Крейсер 'Чудный'");
    AllocShip(4, 2, "Крейсер 'Добрый'"); AllocShip(5, 2, "Крейсер 'Справедливый'");
    AllocShip(6, 1, "Миноносец 'Храбрый'"); AllocShip(7, 1, "Миноносец 'Ушлый'");
    AllocShip(8, 1, "Миноносец 'Проворный'"); AllocShip(9,1, "Миноносец 'Смелый'");
    vetoSet.clear();
}

void UserNavy::FillDeadZone( Rect r, Field& field ) {
    int i, j;
    Rect sh = Shell( r );
    for ( i = sh.lt.row, j = sh.lt.col; j <= sh.rb.col; j++ )
        if ( sh.lt.row < r.lt.row ) field[i][j] = ' ';
    for ( i = sh.rb.row, j = sh.lt.col; j <= sh.rb.col; j++ )
        if ( sh.rb.row > r.rb.row ) field[i][j] = ' ';
    for ( j = sh.lt.col, i = sh.lt.row; i <= sh.rb.row; i++ )
        if ( sh.lt.col < r.lt.col ) field[i][j] = ' ';
    for ( j = sh.rb.col, i = sh.lt.row; i <= sh.rb.row; i++ )
        if ( sh.rb.col > r.rb.col ) field[i][j] = ' ';
}

void UserNavy::FireOff() {
    string capital_letter = "ABCDEFGHJIJ";
    string small_letter = "abcdefghij";
    unsigned char rowName; // обозначение ряда (A, B, ... , J)
    int colName;          // обозначение колонки (1, 2, ..., 10)
    int row;              // индекс ряда (0, 1, ... , 9)
    int col;              // индекс колонки (0, 1, ... , 9)
    bool success = false;

```

```

while ( !success ) {
    cin >> rowName;
    row = capital_letter.find( rowName );
    if ( -1 == row ) row = small_letter.find( rowName );
    if ( -1 == row ) { cout << "Ошибка. Повторите ввод.\n"; continue; }
    colName = GetInt();
    col = colName - 1;
    if ( ( col < 0 ) || ( col > 9 ) ) {
        cout << "Ошибка. Повторите ввод.\n"; continue;
    }
    success = true;
}
u_fire = Cell( row, col );
}

void UserNavy::ResultAnalys() {
    // r_state - сообщение робота о результате выстрела
    // пользователя по клетке u_fire
    switch( r_state ) {
    case Miss:
        enemyField[u_fire.row][u_fire.col] = MISS; break;
    case Damage:
        enemyField[u_fire.row][u_fire.col] = DAMAGE;
        crushSet.insert( u_fire ); break;
    case Kill:
        enemyField[u_fire.row][u_fire.col] = DAMAGE;
        crushSet.insert( u_fire );
        Rect kill;
        kill.lt = *crushSet.begin();
        kill.rb = *( --crushSet.end() );
        FillDeadZone( kill, enemyField ); // Заполняем "обрамление" пробелами
        crushSet.clear();
    }
}

void UserNavy::GetFire() {
    // выстрел робота - по клетке r_fire
    string capital_letter = "ABCDEFGHJIJ";
    char rowName = capital_letter[r_fire.row];
    int colName = r_fire.col + 1;
    cout << "\nВыстрел неприятеля: " << rowName << colName << endl;
    if ( DECK == ownField[r_fire.row][r_fire.col] ) {
        cout << "*** Есть попадание! ***";
        ownField[r_fire.row][r_fire.col] = DAMAGE;
        u_state = Damage;
        int ind = shipMap[r_fire]; // индекс корабля, занимающего клетку r_fire
        ship[ind].nLiveDeck--;
        if ( !ship[ind].nLiveDeck ) {
            u_state = Kill;
            cout << gap( 6 ) << "О ужас! Погиб " << ship[ind].name << " !!!";
            nLiveShip--;
        }
    }
}

```

продолжение ➤

Листинг 15.9 (продолжение)

```

        Rect kill = ship[ind].place;
        FillDeadZone( kill, ownField ); }
    }
    else {
        u_state = Miss;
        cout << "*** Мимо! ***";
        ownField[r_fire.row][r_fire.col] = MISS;
    }
    cout << endl;
}
RobotNavy::RobotNavy() { // ----- Класс RobotNavy
    Allocation();
    isCrushContinue = false;
    upEmpty = false;
}
void RobotNavy::Allocation() {
    AllocShip(0, 4, "Авианосец 'Алькаида'"); AllocShip( 1, 3, "Линкор 'БенЛаден'");
    AllocShip(2, 3, "Линкор 'Хусейн'");      AllocShip(3, 2, "Крейсер 'Подлый'");
    AllocShip(4, 2, "Крейсер 'Коварный'"); AllocShip(5, 2, "Крейсер 'Злой'");
    AllocShip(6, 1, "Миноносец 'Гадкий'"); AllocShip(7, 1, "Миноносец 'Мерзкий'");
    AllocShip(8, 1, "Миноносец 'Пакостный'");AllocShip(9, 1, "Миноносец 'Душный'");
    vetoSet.clear();
}
void RobotNavy::FireOff() {
    Cell c, cUp;
    if ( !isCrushContinue ) {
        while( 1 ) { // случайный выбор координат выстрела
            c.row = rand() % N;
            c.col = rand() % N;
            if ( !c.InSet( vetoSet ) ) break;
        }
    }
    else {
        c = cUp = r_fire; // "пляшем" от предыдущего попадания
        cUp.row--;
        if ( ( !upEmpty ) && c.row && ( !cUp.InSet( vetoSet ) ) ) c.row--;
        else { c = *( --crushSet.end() ); c.row++; }
    }
    r_fire = c;
    vetoSet.insert( r_fire );
}
void RobotNavy::ResultAnalys() {
    // u_state - сообщение пользователя о результате
    // выстрела робота по клетке r_fire
    switch( u_state ) {
    case Miss:
        if ( isCrushContinue ) upEmpty = true; break;
    case Damage:

```

```

        isCrushContinue = true;
        crushSet.insert( r_fire );                break;
    case Kill:
        isCrushContinue = false;
        upEmpty = false;
        crushSet.insert( r_fire );
        Rect kill;
        kill.lt = *crushSet.begin();
        kill.rb = *( --crushSet.end() );
        AddToVetoSet( Shell( kill ) );
        crushSet.clear();
    }
}

void RobotNavy::GetFire() {
    // выстрел пользователя - по клетке u_fire
    if ( DECK == ownField[u_fire.row][u_fire.col] ) {
        cout << "*** Есть попадание! ***";
        r_state = Damage;
        int ind = shipMap[u_fire];    // индекс корабля, занимающего клетку u_fire
        ship[ind].nLiveDeck--;
        if ( !ship[ind].nLiveDeck ) {
            r_state = Kill;
            cout << gap( 6 ) << "Уничтожен " << ship[ind].name << " !!!";
            nLiveShip--; }
        }
    else { r_state = Miss; cout << "*** Мимо! ***"; }
    cout << endl;
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Main.cpp
#include <iostream>
#include "Navy.h"
using namespace std;
int main() {
    UserNavy userNavy;                                // Начальная позиция
    RobotNavy robotNavy;
    userNavy.Show();
    while ( userNavy.IsLive() && robotNavy.IsLive() ) {
        if ( Space::u_state != Miss ) {    // ----- Выстрел пользователя
            cout << "пропускается...: <Enter>" << endl;
            cin.get(); }
        else {
            userNavy.FireOff();
            robotNavy.GetFire();
            userNavy.ResultAnalys();
            if ( !robotNavy.IsLive() ) { userNavy.Show(); break; }
        }
        if ( Space::r_state != Miss )    // ----- Выстрел робота
            cout << "\nВыстрел неприятеля: пропускается..." << endl;
        else {

```

продолжение ➤

Листинг 15.9 (продолжение)

```

        robotNavy.FireOff();
        userNavy.GetFire();
        robotNavy.ResultAnalys();
    }
    userNavy.Show();
}
if ( userNavy.IsLive() ) cout << "\n:-))) Ура! Победа!!! :-)))" << endl;
else { cout << "\n:-((( Увы. Непрятель оказался сильнее." << endl;
      cout << ":-((( Но ничего, в следующий раз мы ему покажем!!!" << endl;
    }
    cin.get();
}

```

Обратите внимание на использование объектов контейнерных классов: `cset` типа `set<Cell>` в классе `Rect`; `vetoSet` и `crushSet` типа `set<Cell>` в классе `Navy`; `shipMap` типа `map<Cell, int>` в классе `Navy`.

В качестве упражнения рекомендуем вам доработать эту программу, сняв ограничение «только вертикальное расположение кораблей» (решение об ориентации размещаемого корабля принимать случайным образом).

Итоги

1. Стандартная библиотека шаблонов содержит общецелевые классы и функции, которые реализуют широко используемые алгоритмы и структуры данных.
2. STL построена на основе шаблонных классов, поэтому входящие в нее алгоритмы и структуры могут настраиваться на различные типы данных.
3. Использование STL позволяет значительно повысить надежность программ, их переносимость и универсальность, а также уменьшить сроки их разработки.
4. Везде, где это возможно, используйте классы и алгоритмы STL!

Задания**Вариант 1**

Написать программу для моделирования Т-образного сортировочного узла на железной дороге с использованием контейнерного класса `stack` из STL.

Программа должна разделять на два направления состав, состоящий из вагонов двух типов (на каждое направление формируется состав из вагонов одного типа). Предусмотреть возможность ввода исходных данных с клавиатуры и из файла.

Вариант 2

Написать программу, отыскивающую проход по лабиринту, с использованием контейнерного класса `stack` из STL.

Лабиринт представляется в виде матрицы, состоящей из квадратов. Каждый квадрат либо открыт, либо закрыт. Вход в закрытый квадрат запрещен. Если квадрат

открыт, то вход в него возможен со стороны, но не с угла. Программа находит проход через лабиринт, двигаясь от заданного входа. После отыскания прохода программа выводит найденный путь в виде координат квадратов.

Вариант 3

Написать программу, моделирующую управление каталогом в файловой системе. Для каждого файла в каталоге содержатся следующие сведения: имя файла, дата создания, количество обращений к файлу. Программа должна обеспечивать:

- ☐ начальное формирование каталога файлов;
- ☐ вывод каталога файлов;
- ☐ удаление файлов, дата создания которых раньше заданной;
- ☐ выборку файла с наибольшим количеством обращений.

Выбор моделируемой функции должен осуществляться с помощью меню. Для представления каталога использовать контейнерный класс `list` из STL.

Вариант 4

Написать программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- ☐ имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- ☐ имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- ☐ вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте.

Для представления необходимых списков использовать контейнерный класс `list`.

Вариант 5

Написать программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ добавление заявок в список;
- ☐ удаление заявок;
- ☐ вывод заявок по заданному номеру рейса и дате вылета;
- ☐ вывод всех заявок.

Для хранения данных использовать контейнерный класс `list`.

Вариант 6

Написать программу учета книг в библиотеке. Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ добавление данных о книгах, вновь поступающих в библиотеку;
- ☐ удаление данных о списываемых книгах;
- ☐ выдача сведений о всех книгах, упорядоченных по авторам;
- ☐ выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «фамилию и инициалы автора».

Вариант 7

Написать программу «Моя записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с помощью класса `map` или `multimap`.

Вариант 8

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ ввод заявки на обмен;
- ☐ поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- ☐ вывод всей картотеки.

Для хранения данных картотеки использовать контейнерный класс `list`.

Вариант 9

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ первоначальный ввод данных в систему (с клавиатуры или из файла);
- ☐ вывод сведений по всем поездам;
- ☐ вывод сведений по поезду с запрошенным номером;
- ☐ вывод сведений по поездам, следующим до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 10

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ загрузка «базы данных» словаря (из файла);
- ☐ выбор режима работы: англо-русский или русско-английский;
- ☐ вывод вариантов перевода заданного английского слова;
- ☐ вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `map`.

Вариант 11

Написать программу, реализующую игру «Крестики-нолики» между двумя игроками: пользователь и компьютер (робот). В программе использовать контейнеры STL.

Вариант 12

Написать программу, решающую игру-головоломку «Игра в 15». Начальное размещение номеров — случайное. Предусмотреть два режима демонстрации решения: непрерывный (с некоторой задержкой визуализации) и пошаговый (по нажатию любой клавиши). В программе использовать контейнерные классы STL.

Вариант 13

Составить программу формирования списка кандидатов, участвующих в выборах губернатора. Каждая заявка от кандидата содержит: фамилию и инициалы, дату рождения, место рождения, индекс популярности. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ добавление заявки в список кандидатов. Для ввода индекса популярности (значение указано в скобках) предусмотреть выбор с помощью подменю одного из следующих вариантов:
 - ☐ поддержан президентом (70);
 - ☐ поддержан оппозиционной партией (15);
 - ☐ оппозиционный кандидат, который снимет свою кандидатуру в пользу кандидата № 1 (10);
 - ☐ прочие (5).
- ☐ удаление заявки по заявлению кандидата;
- ☐ формирование и вывод списка для голосования.

Хранение данных организовать с применением контейнерного класса `priority_queue` из STL. Для надлежащего функционирования очереди с приоритетами побеспокоиться о надлежащем определении операции `<` (меньше) в классе, описывающем заявку кандидата. Формирование и вывод списка для голосования реализовать посредством выборки заявок из очереди.

Вариант 14

Составить программу моделирования работы автобусного парка. Сведения о каждом автобусе содержат: номер автобуса, фамилию и инициалы водителя, номер маршрута. Следует обеспечить выбор с помощью меню и выполнение функций:

- ☐ начальное формирование данных о всех автобусах в парке в виде списка (ввод с клавиатуры или из файла);
- ☐ имитация выезда автобуса из парка: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся в парке, и записывает эти данные в список автобусов, находящихся на маршруте;
- ☐ имитация въезда автобуса в парк: вводится номер автобуса; программа удаляет данные об этом автобусе из списка автобусов, находящихся на маршруте, и записывает эти данные в список автобусов, находящихся в парке;
- ☐ вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам автобусов;
- ☐ вывод сведений об автобусах, находящихся в парке, и об автобусах, находящихся на маршруте, упорядоченных по номерам маршрутов.

Хранение всех необходимых списков организовать с применением контейнерного класса `map`, в качестве ключа использовать «номер автобуса».

Вариант 15

Составить программу учета заявок на авиабилеты. Каждая заявка содержит: пункт назначения, номер рейса, фамилию и инициалы пассажира, желаемую дату вылета. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ добавление заявок в список;
- ☐ удаление заявок;
- ☐ вывод заявок по заданному номеру рейса и дате вылета;
- ☐ вывод всех заявок, упорядоченных по пунктам назначения;
- ☐ вывод всех заявок, упорядоченных по датам вылета.

Хранение данных организовать с применением контейнерного класса `multimap`, в качестве ключа использовать «пункт назначения».

Вариант 16

Написать программу учета книг в библиотеке. Сведения о книгах содержат: фамилию и инициалы автора, название, год издания, количество экземпляров данной книги в библиотеке. Следует обеспечить выбор с помощью меню и выполнение функций:

- ☐ добавление данных о книгах, вновь поступающих в библиотеку;
- ☐ удаление данных о списываемых книгах;
- ☐ выдача сведений о всех книгах, упорядоченных по авторам;
- ☐ выдача сведений о всех книгах, упорядоченных по годам издания.

Хранение данных организовать с применением контейнерного класса `vector`.

Вариант 17

Написать программу «Моя записная книжка». Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Хранение данных организовать с применением контейнерного класса `list`.

Вариант 18

Написать программу учета заявок на обмен квартир и поиска вариантов обмена. Каждая заявка содержит фамилию и инициалы заявителя, а также сведения о двух квартирах: требуемой (искомой) и имеющейся. Сведения о каждой квартире содержат: количество комнат, площадь, этаж, район. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ ввод заявки на обмен;
- ☐ поиск в картотеке подходящего варианта: при совпадении требований и предложений по количеству комнат и этажности и при различии по показателю «площадь» в пределах 10% выводится соответствующая карточка и удаляется из списка, в противном случае поступившая заявка включается в картотеку;
- ☐ вывод всей картотеки.

Хранение данных организовать с применением контейнерного класса `set`.

Вариант 19

Написать программу «Автоматизированная информационная система на железнодорожном вокзале». Информационная система содержит сведения об отправлении поездов дальнего следования. Для каждого поезда указывается: номер поезда, станция назначения, время отправления. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ первоначальный ввод данных в информационную систему (с клавиатуры или из файла);
- ☐ вывод сведений по всем поездам;
- ☐ вывод сведений по поезду с запрошенным номером;
- ☐ вывод сведений по поездам, следующим до запрошенной станции назначения.

Хранение данных организовать с применением контейнерного класса `set`.

Вариант 20

Написать программу «Англо-русский и русско-английский словарь». «База данных» словаря должна содержать синонимичные варианты перевода слов. Следует обеспечить выбор с помощью меню и выполнение следующих функций:

- ☐ загрузка «базы данных» словаря (из файла);
- ☐ выбор режима работы: англо-русский или русско-английский;
- ☐ вывод вариантов перевода заданного английского слова;
- ☐ вывод вариантов перевода заданного русского слова.

Базу данных словаря реализовать в виде двух контейнеров типа `set`.

Приложение. Основные приемы работы в Microsoft Visual C++ .NET 2005

Visual C++ входит в состав интегрированной среды Microsoft Visual Studio.NET 2005, позволяющей также работать с Visual C#, Visual Basic и Visual J#. Интегрированная среда разработки (*Integrated Development Environment, IDE*) — это программный продукт, объединяющий текстовый редактор, компилятор, отладчик и справочную систему. Среда позволяет создавать приложения различных типов, в частности, используемые в этой книге консольные приложения.

При запуске *консольного приложения* операционная система создает так называемое *консольное окно*, через которое идет весь ввод-вывод программы. Внешне это напоминает работу в операционной системе MS-DOS или других операционных системах в режиме командной строки. Этот тип приложений больше всего подходит для целей изучения языка C++, так как компилируемые программы не «покрываются» толстым слоем промежуточного Windows-кода.

Компания Microsoft бесплатно распространяет версии Express, содержащие все необходимое для изучения языка и стандартной библиотеки. В этом Приложении приводятся минимально необходимые сведения для начала работы, более подробную информацию можно извлечь из справочной системы.

Любая программа, создаваемая в среде, оформляется как отдельный *проект (project)* — набор взаимосвязанных исходных и, возможно, заголовочных файлов, компиляция и компоновка которых позволяет создать исполняемую программу. Проекты, которые разрабатываются согласованно, объединяют в *решение*, или *сборку (solution)*. Это удобно для коллективов разработчиков.

Запуск интегрированной среды. Создание пустого проекта

При *запуске* системы отображается стартовая страница (Start Page), содержащая несколько окон. Окно Recent Projects содержит ссылки на последние проекты,

с которыми выполнялась работа, и меню для открытия существующего проекта (Open...) или создания нового (Create...). Это меню дублирует пункты Open и New пункта File главного меню.

Для разработки консольной программы необходимо *создать проект консольного приложения*. Это можно сделать разными способами, мы покажем один из них.

Выберите в главном меню пункт File ► New ► Project... На экране появится окно создания нового проекта New Project, в левой части которого содержится дерево доступных типов проектов. Выберите в дереве папку Visual C++, в ней — папку Win32. Справа щелкните на значке Win32 Console Application. В поле Name введите имя проекта, а в поле Location выберите папку, в которой будет размещен проект. После нажатия кнопки OK появится окно Win32 Application Wizard. Щелкните слева на пункте Application Settings, в появившейся справа панели Application Settings установите флажок Empty project. После нажатия кнопки Finish будет создан новый *пустой проект*.

При таком способе создания проекта в его свойствах по умолчанию задан режим работы с набором символов Unicode. Это может быть причиной ошибок компиляции, если в коде используются библиотечные функции, не рассчитанные на работу с Unicode. Поэтому рекомендуется сразу изменить эту установку. Выберите пункт меню Project ► Имя_Проекта Properties. В появившемся окне Имя_Проекта Property Pages выберите слева папку Configuration Properties, а в ней опцию General. Справа для режима Character Set выберите значение Not Set.

Заготовка проекта готова, теперь надо добавить в него файлы для размещения текста программы.

Добавление файлов к проекту

Для *добавления нового файла* к проекту выберите пункт меню Project ► Add New Item (Ctrl+Shift+A). В открывшемся окне Add New Item слева выберите папку Code, а справа — тип создаваемого файла (для текста программы это C++ File). В поле Name задайте имя файла. По умолчанию в поле Location будет выбрана папка, в которой создан проект. После нажатия кнопки Add будет создана новая пустая вкладка с именем файла. В ней можно начинать писать программу.

Если файл с текстом программы был создан независимо от проекта, можно *присоединить его к проекту* с помощью команды Project ► Add Existing Item (Shift+Alt+A). На экране появится диалоговое окно Add Existing Item, в котором выбирается нужный файл.

В меню Project есть пункт Add Class, с помощью которого в проект *добавляются заготовки классов*, состоящие из исходного и заголовочного файлов. Система позволяет просмотреть иерархию созданных классов, а также глобальные переменные и функции, в окне Class View, которое можно открыть с помощью пункта меню View ► Class View (Ctrl+Shift+C).

Присоединенные файлы вставляются в папку Source Files или Header Files в дерево проекта в окне Solution Explorer, которое находится в левой или правой части экрана

в зависимости от выбранных режимов профиля. Если окна нет, открыть его можно с помощью команды View ► Solution Explorer (Ctrl+Alt+L).

Для *удаления файла из проекта* достаточно исключить его из дерева проекта в окне Solution Explorer с помощью правой кнопки мыши или клавиши Delete.

Многофайловые проекты

Никаких особых усилий при создании многофайловых проектов вам прилагать не придется: разработчики IDE уже обо всем позаботились. Надо просто несколько раз повторить процедуру создания (добавления) исходных файлов, описанную выше. В многофайловых проектах обычно присутствуют и заголовочные файлы, для которых выбирается тип файла C/C++ Header File.

ПРИМЕЧАНИЕ

Папки Source Files и Header Files, которые вы видите в окне Solution Explorer, на самом деле физически не существуют: все файлы помещаются в основную папку проекта, имя которой было задано при его создании. Такое упорядочение дерева списка файлов сделано для удобства.

Редактирование текста программы

При наборе и редактировании текста программы часто возникает необходимость выделения, копирования, вырезания, вставки, поиска и замены фрагментов текста. Для этого используются: меню Edit, кнопки панели инструментов, контекстное меню, вызываемое по правой кнопке мыши, и клавиатурные сочетания.

Выделение фрагмента выполняется разными способами. Самый простой — с помощью клавиши Shift и клавиш со стрелками. Если курсор находится в произвольной позиции строки, то удерживая нажатой клавишу Shift, можно с помощью клавиши → выделить любую подстроку. Если курсор находится в начале строки, удерживая нажатой клавишу Shift, можно с помощью клавиши ↓ выделить всю строку. Строку можно также выделить щелчком мыши чуть левее начала строки.

Копирование выделенного фрагмента в буфер выполняется с помощью клавиш Ctrl+C, *вставка* из буфера — Ctrl+V, *вырезание* фрагмента в буфер — Ctrl+X.

При отладке часто возникает необходимость *закомментировать* фрагменты текста. Для этого их выделяют, после чего используют кнопку на панели инструментов (Comment out the selected lines) или сочетание клавиш Ctrl+E,C. Рядом с этой кнопкой находится другая, выполняющая противоположное действие.

В левой полосе окна символически отображается структура программы (по аналогии с представлением папок в Проводнике Windows), с помощью которой удобно управлять видимостью отдельных элементов программы.

Текстовый редактор обладает впечатляющими возможностями автодополнения текста, который вы набираете, и контекстных подсказок. Никогда не набирайте вручную то, что можно получить с помощью автодополнения, это уменьшает количество ошибок.

Сохранение проекта и программы, завершение работы

Для сохранения проекта и текста программы выберите пункт меню File ► Save All (Ctrl+Shift+S). Для сохранения только программы выполните команду File ► Save (Ctrl+S). Можно сохранить текст программы под новым именем с помощью пункта меню File ► Save As. Чтобы закончить работу с текущей программой, надо закрыть проект, выбрав пункт меню File ► Close Solution. Для *завершения работы* с IDE выполните команду File ► Exit.

Продолжение работы над проектом

Существующий проект можно открыть, либо выбрав его из списка недавно использовавшихся (на стартовой странице), либо с помощью пункта меню File ► Recent Projects, либо с помощью пункта меню File ► Open ► Project/Solution (Ctrl+Shift+O). На экране появится окно Open Project. Система сохраняет файлы проектов с расширением `vsproj`, а файлы решений (solution) с расширением `sln`. Для открытия проекта можно выбрать либо файл проекта, либо файл решения. При нажатии кнопки Open система загружает проект, открывая все окна, которые были открыты на момент сохранения проекта.

Компиляция и компоновка программы

Для компиляции и компоновки проекта служит меню Build, которое появляется в главном меню после создания проекта. Для *компиляции отдельного файла* выберите пункт меню Build ► Compile (Ctrl+F7). Результат компиляции в виде списка синтаксических ошибок и предупреждений появляется в окне Output в нижней части экрана. Для каждой ошибки выводится ее тип (error или warning), имя файла и номер строки исходного текста.

Двойной щелчок по строке ошибки переводит указатель в окне редактора на ошибочную строку текста программы. Ошибки (error) не позволяют выполнять дальнейшую компоновку программы, предупреждения (warning) не препятствуют компоновке и выполнению, однако к ним надо относиться внимательно и всегда осознавать, почему выдается данное предупреждение.

Компиляция и компоновка проекта (создание исполняемого файла) выполняется с помощью пунктов Build Solution (F7) или Build *Имя_проекта*. По этим пунктам выполняется компиляция элементов, которые подверглись изменению со времени последней сборки. Безусловная компоновка проекта выполняется с помощью пунктов Rebuild. Пункты Clean предназначены для очистки сведений о компиляции и компоновке проекта, после чего меню Build работает как при первой компоновке. Сообщения об ошибках компоновки выводятся в окно Output. Если компоновка завершилась без ошибок, созданный исполняемый файл с расширением `.exe` может быть запущен на выполнение.

Настройки компиляции и компоновки. При выборе пункта Configuration Manager на экране появляется окно, в верхнем списке которого Active Solution Configuration задается конфигурация всей сборки и отдельного проекта: Debug — отладочная

конфигурация или Release — рабочая. Для учебных проектов установите режим Debug. Система в папке с проектом создает либо папку Debug, либо папку Release.

Установка значений *режимов трансляции и компоновки* по умолчанию выполняется с помощью пункта меню Project ► *Имя_Проекта* Properties. Окно *Имя_Проекта* Property Pages слева показывает дерево папок, а справа — набор режимов, которые относятся к соответствующей папке. В правой верхней части окна находится кнопка Configuration Manager, с помощью которой можно вызвать соответствующее окно и внести изменения в конфигурацию проекта.

Практически все пункты в свойствах проекта устанавливаются по умолчанию при создании и конфигурации проекта. Начинающего программиста интересует буквально пара пунктов в папке C/C++:

- ☐ General ► Additional Include Directories — позволяет указать дополнительный каталог (помимо стандартного), в котором препроцессор будет искать подключаемые файлы;
- ☐ Language ► Disable Language Extension — запретить языковые расширения; по умолчанию выставлен режим No (расширения разрешены); при установке режима Yes компилятор значительно строже следит за соответствием программы стандарту C++.

Настройка самой IDE выполняется с помощью пунктов меню Tools ► Options и Tools ► Customize.

Выполнение и отладка программы

Для прогона и отладки программы используется меню Debug. *Запуск и выполнение* программы без остановок под управлением интегрированной среды инициируется командой Debug ► Start (F5). Если в программе установлена контрольная точка (точка останова), выполнение будет прервано на ней.

Выполнить программу без остановок независимо от контрольных точек можно с помощью команды Debug ► Start Witout Debugging (Ctrl+F5). Этот режим полезно использовать *при запуске консольных приложений*, поскольку он не закрывает консольное окно после завершения выполнения программы.

Полное описание возможностей встроенного отладчика Visual C++ и приемов работы с ним может потребовать отдельной книги, настолько объемна эта тема. Поэтому мы дадим только начальные сведения о работе с отладчиком. Отладочные возможности системы Visual C++ представлены в меню Debug:

- ☐ Step Into (F11) — выполнить программу по шагам, заходя в функции;
- ☐ Step Over (F10) — выполнить программу по шагам, не заходя в функции;
- ☐ Step Out (Shift+F11) — выполнить до ближайшего выхода из функции;
- ☐ Stop Debugging (Shift+F5) — прекратить пошаговое выполнение программы;
- ☐ Continue (F5) — продолжить выполнение до ближайшей точки останова;
- ☐ Restart (Ctrl+Shift+F5) — начать выполнение сначала¹.

¹ Последние четыре пункта видны только в процессе отладки.

Отладчик позволяет установить в программе одну или несколько *контрольных точек*. Точка прерывания устанавливается в окне редактора на нужной строке программы. Это делается двумя способами:

- ☐ щелкните левой кнопкой мыши на широкой вертикальной полосе слева от нужной строки — напротив появится коричневый кружок;
- ☐ выберите пункт меню **Debug** ► **New Breakpoint** ► **Break at Function...** (Ctrl+B); откроется окно **New Breakpoint**, в котором в поле **Line** нужно набрать номер строки для прерывания, а затем нажать кнопку **OK**.

После задания точек останова можно выполнять программу по шагам, наблюдая интересные значения в окнах отладчика, которые открывают с помощью пункта **Debug** ► **Windows**. Например, значения локальных переменных можно наблюдать в окне **Autos**. Кроме того, для наблюдения за переменными и для вычисления выражений во время работы программы отладчик предоставляет окно **QuickWatch**, открываемое командой **Debug** ► **QuickWatch** (Ctrl+Alt+Q).

Удалить все контрольные точки позволяет пункт меню **Debug** ► **Delete All Breakpoints** (Ctrl+Shift+F9). Можно запретить остановки на точках прерывания, не удаляя самих точек, с помощью пункта меню **Debug** ► **Disable All Breakpoints**. Его название меняется на «противоположное» — **Enable All Breakpoints**, и смысл, соответственно, тоже — теперь эта команда будет включать отключенные контрольные точки.

Работа со справочной системой

Интегрированная среда Microsoft Visual Studio.NET 2005 использует в качестве справочной системы пакет MSDN — обычно он входит в состав поставки системы. Пакет устанавливается отдельно, но вписывается в интегрированную среду. При вызове изнутри интегрированной среды справка (на английском языке) выводится в отдельном окне (рис. П.1).

Справочная система имеет меню **Help** (Справка), наиболее важные пункты которого:

- ☐ **Contents** (Содержание) — Ctrl+Alt+F1;
- ☐ **Index** (Алфавитный перечень) — Ctrl+Alt+F2;
- ☐ **Search** (Поиск) — Ctrl+Alt+F3.

Для получения справки по C++ в окнах **Contents** (Содержание) и **Index** (Алфавитный перечень), размещаемых на одном и том же месте в левой части окна попеременно (в зависимости от выбранного пункта меню или кнопки на панели инструментов), необходимо выбрать фильтр, соответствующий этому языку, в списке **Filtered by:** (Отбор из:), находящемся в верхней части окна. Пример выдачи справочной информации по использованию справочной системы показан на рис. П.1 (для удобства восприятия приведена копия экрана на русском языке из следующей версии — Microsoft Visual Studio 2008). Для получения этой информации необходимо выбрать пункт меню **Help** ► **Contents** (Справка ► Содержание), а затем, раскрывая список тем (нажатием на «плюсы» в дереве тем), выбрать требуемую строку.

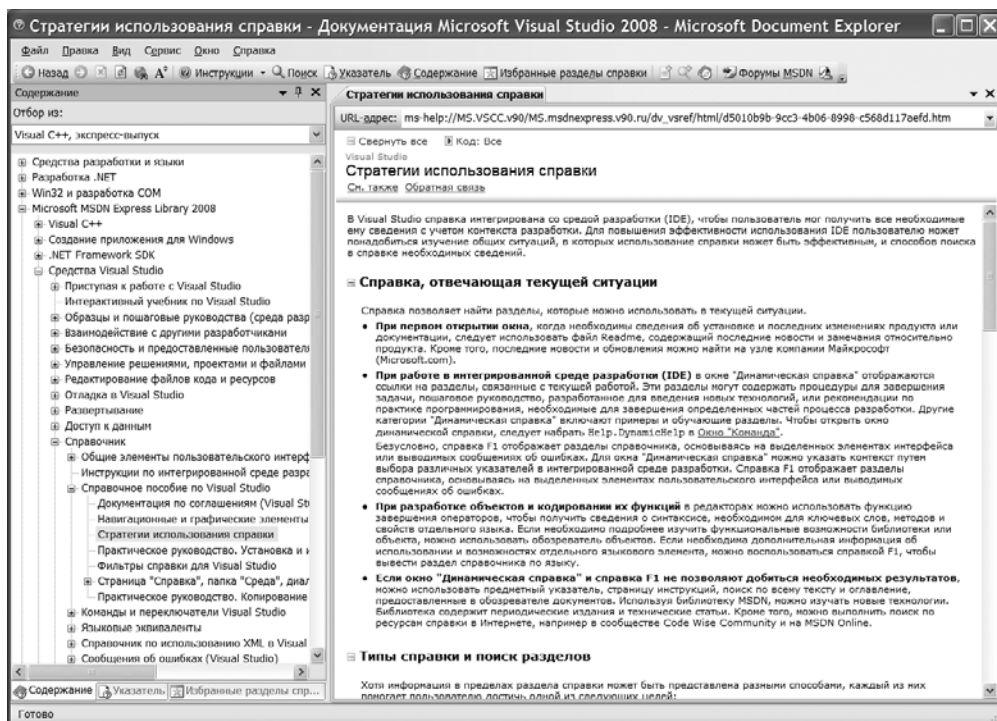


Рис. П. 1. Справка по использованию справки

Для получения контекстной справки по какому-либо элементу языка необходимо установить на него текстовый курсор в окне редактора кода и нажать клавишу F1. Для кнопок панелей инструментов IDE предусмотрена удобная контекстная помощь: если вы наведете курсор мыши на кнопку и задержитесь на секунду-другую, то всплывет подсказка с назначением кнопки.

Изучение информации, представленной в справочной системе, и стандарта языка обеспечат вам прочные и достоверные знания — в отличие от книг, в том числе и этой, не свободных от ошибок, опечаток, устаревшей информации и некомпетентности авторов или переводчиков.

Литература

1. *Александреску А.* Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования. — М.: Вильямс, 2008.
2. *Александреску А., Саттер Г.* Стандарты программирования на C++. — М.: Вильямс, 2008.
3. *Аммерааль Л.* STL для программистов на C++. — М.: ДМК, 1999 г.
4. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами на C++. — М.: Бином, 2001.
5. *Вандевурд Д., Джосаттис Н.* Шаблоны C++: справочник разработчика. — М.: Вильямс, 2003.
6. *Влссидес Дж.* Применение шаблонов проектирования. Дополнительные штрихи. — М.: Вильямс, 2003.
7. *Гамма Э., Хелм Р., Джонсон Р., Влссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2007.
8. *Дейтел П. Дж., Дейтел Х. М.* Как программировать на C++. Введение в объектно-ориентированное проектирование с использованием UML. — М.: Бином, 2009.
9. *Джосьютис Н.* C++. Стандартная библиотека. — СПб.: Питер, 2004.
10. *Кениг Э., Му Б.* Эффективное программирование на C++. Серия C++ In-Depth, т. 2. — М.: Вильямс, 2002.
11. *Коллинз У.Дж.* Структуры данных и стандартная библиотека шаблонов. — М.: Бином-Пресс, 2004.
12. *Коплиен Дж.* Программирование на C++. — СПб.: Питер, 2005.
13. *Лаптев В.* C++. Экспресс-курс. — СПб: БХВ-Петербург, 2004.
14. *Лаптев В. В.* C++. Объектно-ориентированное программирование: Учебное пособие. — СПб.: Питер, 2008.
15. *Лафоре Р.* Объектно-ориентированное программирование в C++. Классика Computer Science. 4-е изд. — СПб.: Питер, 2011.
16. *Липпман С. Б., Лажоие Ж., Му Б. Э.* Язык программирования C++. Вводный курс, 4-е изд.: Пер. с англ. — М.: Вильямс, 2007.
17. *Липпман С. Б.* Основы программирования на C++. Серия C++ In-Depth, т. 1: пер. с англ. — М.: Вильямс, 2002.

18. *Мейерс С.* Эффективное использование C++. 50 рекомендаций по улучшению наших программ и проектов / Пер. с англ — М.: ДМК Пресс, 2000.
19. *Мейерс С.* Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению наших программ и проектов / Пер. с англ — М.: ДМК Пресс, 2000.
20. *Мейерс С.* Эффективное использование STL. Библиотека программиста. — СПб.: Питер, 2002.
21. *Мейерс С.* Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ — М.: ДМК Пресс, 2006.
22. *Павловская Т. А.* C/C++ Программирование на языке высокого уровня. — СПб.: Питер, 2011.
23. *Павловская Т. А., Щупак Ю. А.* C/C++. Структурное программирование: Практикум. — СПб.: Питер, 2002 г.
24. *Павловская Т. А., Щупак Ю. А.* C++. Объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2004 г.
25. *Романов Е. Л.* Практикум по программированию на C++: Уч. пособие. — СПб.: БХВ-Петербург, 2004.
26. *Саттер Г.* Решение сложных задач на C++. Серия C++ In-Depth, т. 4: пер. с англ. — М.: Вильямс, 2002.
27. *Саттер Г.* Новые сложные задачи на C++: Пер. с англ. — М.: Вильямс, 2005.
28. *Страуструп Б.* Язык программирования C++, спец. изд. Пер. с англ. — М.: БИНОМ; СПб.: Невский Диалект, 2008 г.
29. *Сэджвик Р.* Фундаментальные алгоритмы на C++. Анализ / Структуры данных / Сортировка / Поиск. / пер. с англ. — Киев.: ДиаСофт, 2001.
30. *Сэджвик Р.* Фундаментальные алгоритмы на C++. Алгоритмы на графах: пер. с англ. — СПб.: ДиаСофтЮП, 2002.
31. *Уилсон М.* C++: практический подход к решению проблем программирования / Пер. с англ. — М.: КУДИЦ-ОБРАЗ, 2006.
32. *Уэллин С.* Как не надо программировать на C++. — СПб.: Питер, 2004.
33. *Черносвитов А.* Visual C++ 7: учебный курс. — СПб.: Питер, 2001.
34. *Шаллоуей А., Тротт Д.* Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию. — М.: Вильямс, 2002.
35. *Шилдт Г.* Искусство программирования на C++. — СПб.: BHV, 2005.
36. *Штерн В.* Основы C++. Методы программной инженерии. — М.: Лори, 2003.
37. *Эккель Б.* Философия C++. Введение в стандартный C++. 2-е изд. — СПб.: Питер, 2004.
38. *Эккель Б., Эллисон Б.* Философия C++. Практическое программирование. 2-е изд. — СПб.: Питер, 2004.
39. *Элджер Д.* C++: библиотека программиста. — СПб.: Питер, 2000.
40. Стандарт C++: International Standart ISO/IEC 14882:2003(E), Programming languages — C++.

Алфавитный указатель

Symbols

#endif, 115
#ifndef, 115
_getch, 87
&, операция, 20
&&, операция, 28
<<, операция, 15
>>, операция, 15
||, операция, 29
базовый класс, 194
виртуальные методы, 198
деструктор, 164
зависимость (использование), 202
интерфейс, 162
конструктор, 164
метод, 162
производный класс, 195
элемент класса, 162

A

abort, 249
algorithm, 298
app, ios, 284
ASCII, 18
ate, ios, 284
atof, 72, 111
atoi, 72, 111
atol, 72
at, string, 288

B

back_inserter, итератор вставки, 310
back, контейнеры, 301, 302

badbit, ios, 270
bad, ios, 270
begin, контейнеры, 296
BidirectionalIterator, 297
binary, ios, 284
bool, 28
break, 32, 36

C

case, 31
catch, 247
cctype, 77
cerr, 245
char, 68, 72
CharToOem, 75
cin, 15, 267
class, 236
clear, ios, 270
clear, контейнеры, 298
clocale, 18
close, 80
close, 285
cohesion, 161
conio.h, 87
const, 104, 124, 168
const_iterator, 297
const_reference, 298
copy, string, 289
copy, алгоритм, 309
cosh, 36
count_if, алгоритм, 306
count, алгоритм, 305

count, множество, 312
coupling, 161
cout, 15, 267
cstdio, 20
cstdlib, 92
cstring, 71
c_str, string, 289

D

DBL_EPSILON, 105
dec, манипулятор, 268
default, 32
delete, 41
deque, 299
double, 16
do while, 32
dynamic_cast, 231

E

empty, контейнеры, 298, 302
endl, 17
endl, 268
ends, 269
end, контейнеры, 296
eof, 285
eofbit, ios, 270
eof, ios, 270
equal_to, 308
erase, string, 289
erase, контейнеры, 298, 301
erase, множество, 312
exception, 247
extern, 113

F

failbit, ios, 270
fail, ios, 270
false, 28
feof, 87
fgets, 88
find_first_of, string, 289
find_if, алгоритм, 306
find_last_of, string, 289
find, string, 288
find, алгоритм, 305

fixed, 122, 268
FLT_EPSILON, 31
flush, 269
for, 32
for_each, алгоритм, 306
ForwardIterator, 297
fputc, 104
fread, 90
free, 41
front_inserter, итератор вставки, 310
front, контейнер queue, 304
front, контейнеры, 301
fscanf, 88
fseek, 91
fstream, 85, 283
fwrite, 90

G

gcount, 269
get, 69, 72, 80, 269
getch, 31
getchar, 73
getline, 69, 85
getline, 269
gets, 71
goodbit, ios, 270
good, ios, 270
greater, 308
greater_equal, 308

H

heap, 22
hex, манипулятор, 268

I

IDE, 332
if, 26
ifstream, 74, 283
includes, алгоритм, 312
in, ios, 284
inline, 165
InputIterator, 296
inserter, итератор вставки, 310
insert, string, 288
insert, контейнеры, 298, 301

insert, множество, 312
int, 15, 23
iomanip, 59, 268, 269
ios, 80, 122, 266
iostream, 15, 267
isalfa, 73
isdigit, 73
ispunct, 73, 77
isspace, 73, 77
istream, 69, 267
istreamstream, 286
iterator, 296, 297

К

key_compare, 298
key_type, 298

Л

length, string, 288
less, 308
less_equal, 308
list, 242, 295, 299
long, 22

М

main, 15, 99
malloc, 40, 56, 68
map, 238, 311, 313
math.h, 24
memcpy, 99
merge, алгоритм, 310
multimap, 311
multiset, 311

Н

new, 40, 45, 56, 68, 255
not_equal_to, 308
NULL, 75, 129

О

oct, манипулятор, 268
OemToChar, 75
ofstream, 80, 283
open, 284
ostream, 89, 267

ostreamstream, 286
out, ios, 284
out_of_range, string, 288
OutputIterator, 296

Р

pair, 313
peek, 269
pop_back, контейнеры, 301, 302
pop_front, контейнеры, 301
pop, контейнер queue, 304
pop, контейнер stack, 303
precision, 122
printf, 20, 70
priority_queue, 304
private, 162, 195
project, 332
protected, 162, 195
public, 162, 195
push_back, контейнеры, 301
push_front, контейнеры, 301
push, контейнер queue, 304
push, контейнер stack, 303
put, 269
putchar, 73
puts, 71

Q

qsort, 92
queue, 304

Р

rand, 49
RandomAccessIterator, 297
rdbuf, 269
rdstate, ios, 270
read, 80
read, 269
reference, 297
reinterpret_cast, 57, 93
replace, string, 288
return, 100
reverse_iterator, 297, 309
rfind, string, 289
RTTI, 231

S

scanf, 20, 70
scientific, ios, 268
search, алгоритм, 306
seekg, 80
set, 311
setf, 122
setfill, 268
set_intersection, алгоритм, 312
setlocale, 18
setprecision, 268
set_terminate, 250
set_unexpected, 251
set_union, алгоритм, 312
setw, 59, 122, 268
signed, 22
sin, 99
sizeof, 68
size, string, 288
size_type, 287, 297
sort, алгоритм, 307, 308
srand, 49
sstream, 286
stack, 303
state, ios, 270
static, 21, 113, 175
static_cast, 35
std, пространство имен, 15
STL, 294
strcpy, 71
streambuf, 266
string, 68, 208, 287
stringstream, 286
strlen, 72
strncpy, 71, 86
strstr, 75, 87
strtok, 77
struct, 83, 163
str, строковые потоки, 286
substr, string, 288
swap, string, 289
swap, контейнеры, 302
switch, 26

T

tellg, 80
template, 125, 235
terminate, 249
throw, 247, 251
tolower, 87
top, контейнер stack, 303
toupper, 87
true, 28
trunc, ios, 284
try, 247
typedef, 106
typename, 236

U

UML, 200
unexpected, 251
unsigned, 22
using, 213
utility, 313

V

value_type, 297
value_type, множество, 313
vector, 209, 295, 299
virtual, 198
void, 99
vptr, 198
vtbl, 198

W

while, 32
width, ios, 268
windows.h, 75
write, 269

X

x_precision, ios, 268

A

абстрактный класс, 199
агрегация, 176, 201
адаптер итератора, 297
адаптер контейнера, 303

алгоритмы, 298
аргументы функции, 100
ассоциативный контейнер, 295
ассоциация, 200
атрибут, 200

Б

бинарное дерево, 139
бинарный файл, 90, 111
блок, 30
буфер, 266
быстрая сортировка, 47, 128

В

включение, 176
время жизни, 21
встроенный метод, 165
вывод русских букв, 18
выражение, 17

Г

глобальная переменная, 21, 113

Д

двумерный массив, 55
декомпозиция, 160
дерево поиска, 139
деструктор
 виртуальный, 199
 наследование, 196
диаграмма видов деятельности, 221
диаграмма классов, 200
динамические структуры данных, 128
динамический массив, 40
директива препроцессора, 15

З

заглушка, 138
заголовок функции, 15, 99
заголовочный файл, 15, 113, 334

И

иерархии исключений, 250
иерархия, 195

извлечение из потока, 15
имя переменной, 16
индекс, 40, 55
инициализаторы конструктора, 170
инкапсуляция, 162
инстанцирование, 125, 236
интегрированная среда, 332
интерфейс, 113, 161
интерфейс функции, 99
исключения, 247
 в деструкторе, 254
 в конструкторе, 252
 неперехваченные, 249
итератор, 294, 295
 вставки, 310
 обратный, 309
 основные операции, 296

К

кириллица, 18, 75
класс, 162
классы исключений, 250
клиент, 161
ключ, 139, 311
кодировка, 18, 75
компиляция, 15
композиция, 202
консольное окно, 332
константный метод, 168
конструктор
 копирования, 171
 наследование, 196
 по умолчанию, 170
контейнер, 294
 ассоциативный, 311
 множество (set), 311
 общие типы, 297
 очередь (queue), 304
 очередь с приоритетами
 (priority_queue), 304
 последовательный, 300
 стек (stack), 303
контролируемый блок, 247
критерии качества программы, 28
кэш, 47

Л

линейная программа, 14
линейный список, 131
логическое И, 28
логическое ИЛИ, 29
локализация приложений, 18
локальная переменная, 21

М

манипулятор, 17
массив, 40
метод выбора, 62
многофайловый проект, 334
множество, 311
модификаторы формата, 20
модификация, 160
мультимножество (multiset), 311
мультисловарь, 311

Н

надежность, 19
наследование, 163, 194, 201
 неоднозначность, 196
неоднозначность перегрузки, 122
нуль-символ, 68

О

область действия, 21
обобщенное программирование, 235
обобщенный алгоритм, 294, 298
обработка исключений, 246
объект класса, 162
объекты-потoki, 15
объявление функции, 100
ООП, 160
операторы цикла, 32
операции отношения, 28
операция
 инкремента, 173
 присваивания, 17, 174
 выбора, 83
описание переменных, 21
отладка программы, 122
отладчик, 336
ошибки потоков, 269

П

параметры по умолчанию, 105
параметры шаблона, 238
паттерн, 203
перегрузка
 операций, 172
 функций, 121
передача параметров в функцию, 103
перехват исключения, 248
позднее связывание, 198
поле, 162
поле структуры, 83
полиморфизм, 163, 199
помещение в поток, 15
понижающее преобразование, 231
поточковый класс, 266
предикат, 306
предикаты стандартной библиотеки, 308
приоритет операций, 17
проект, 332
прототип функции, 99
прямой доступ, 90

Р

разадресация, 56
размерность массива, 40
разреженный массив, 241
раннее связывание, 198
раскрутка стека, 249
режим открытия файла, 284
рекурсивная функция, 111
русификация, 18

С

связанность, 161
сервер, 161
сигнатура, 165
символьный литерал, 16
словарь, 311
сложность ПО, 160
сопровождение продукта, 160
специализация шаблона, 238
спецификации исключений, 251
спецификация формата, 20, 70
список, 131

ссылка, 103
стандартная библиотека шаблонов, 294
статическая переменная, 21
стек, 48, 129
стражи включения, 115
строки, 68
строки класса string, 287
строковые потоки, 286
структура, 83
сцепление, 161

Т

таблица виртуальных методов, 198
табуляция, 43
тело функции, 16
тестовые примеры, 19
тип, 16
тип функции, 106

У

узел дерева, 139
указатель, 40
унифицированный язык моделирования
UML, 200
управляющая последовательность, 20
условная операция, 31
утечка памяти, 41

Ф

файловые потоки, 283
функциональный класс, 239
функциональный объект, 239, 308
функция, 15, 99
функция-операция, 172

Ц

цикл, 32

Ч

чисто виртуальный метод, 199
читабельность, 16

Ш

шаблон
класса, 235
функции, 125
шаблон проектирования, 203
DoubleSwitch, 215
Strategy, 204
Switch, 207

Татьяна Александровна Павловская, Юрий Абрамович Щупак

**С/С++. Структурное и объектно-ориентированное
программирование: Практикум**

Заведующий редакцией
Руководитель проекта
Ведущий редактор
Художественный редактор
Корректор
Верстка

*А. Кривцов
А. Юрченко
Ю. Сергиенко
Л. Адуевская
И. Тимофеева
Л. Родионова*

Подписано в печать 30.12.10. Формат 70х100/16. Усл. п. л. 28,38. Тираж 2000. Заказ
ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.

Отпечатано по технологии StP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., д. 15.



ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел./факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-79, 201-48-81
e-mail: gv@minsk.piter.com

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73. E-mail: fuganov@piter.com**

Издательский дом «Питер» приглашает к сотрудничеству авторов. Обращайтесь
по телефонам: **Санкт-Петербург – (812) 703-73-72, Москва – (495) 974-34-50**

Заказ книг для вузов и библиотек по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

ДАЛЬНИЙ ВОСТОК

Владивосток

«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»

тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»

тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

Архангельск, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfknight.ru

Воронеж, «Амитель», пл. Ленина, д. 4

тел.: (4732) 26-77-77
http://www.amital.ru

Калининград, «Вестер»,

сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»

Московское шоссе, д. 15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»

пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

СЕВЕРНЫЙ КАВКАЗ

Ессентуки, «Россы», ул. Октябрьская, 424

тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

СИБИРЬ

Иркутск, «ПродаЛитЪ»

тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»

тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир»

пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»

тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

ТАТАРСТАН

Казань, «Таис»,

сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancorp.ru

УРАЛ

Екатеринбург, ООО «Дом книги»

ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

Екатеринбург, ТЦ «Люмна»

ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

Челябинск, ООО «ИнтерСервис ЛТД»

ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!



*Зарегистрируйтесь на нашем сайте в качестве партнера по адресу **www.piter.com/ePartners***



Получите свой персональный уникальный номер партнера



*Выбирайте книги на сайте **www.piter.com**, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт **www.piter.com**)*

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка

<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

**Подробнее о Партнерской программе
ИД «Питер» читайте на сайте
WWW.PITER.COM**







КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM