

Т. А. Павловская
Ю. А. Щупак

ПИТЕР®

C/C++

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ
Структурное программирование

ПРАКТИКУМ

- для студентов и преподавателей высших учебных заведений
- сочетание теоретических сведений и практических занятий
- примеры программ на C++ в соответствии со стандартом ISO/IEC 14882



**Т. А. Павловская
Ю. А. Щупак**

C/C++

**ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ
Структурное программирование**

Допущено Министерством образования Российской Федерации
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению «Информатика и вычислительная техника»



Москва · Санкт-Петербург · Нижний Новгород · Воронеж

Ростов-на-Дону · Екатеринбург · Самара

Киев · Харьков · Минск

2003

Краткое содержание

Рецензенты:

Фомичев В. С., профессор кафедры вычислительной техники Санкт-Петербургского электротехнического университета, доктор технических наук
Соколов Р. В., доцент кафедры МО ЭВМ Санкт-Петербургского электротехнического университета, кандидат технических наук

П12 С/C++. Структурное программирование: Практикум / Т. А. Павловская, Ю. А. Щупак. — СПб.: Питер, 2003. — 240 с.: ил.

ISBN 5-94723-447-5

Практикум предназначен для изучения языка С++ на семинарах и для его самостоятельного освоения. Он является дополнением к учебнику Т. А. Павловской «С/C++. Программирование на языке высокого уровня», выпущенному издательством «Питер» в 2001 году.

В практикуме на примерах рассматриваются средства С++, используемые в рамках структурной парадигмы: стандартные типы данных, основные конструкции, массивы, строки, структуры, функции, шаблоны, динамические структуры данных. Обсуждаются алгоритмы, приемы отладки, вопросы качества и стиля. По каждой теме приведено несколько комплектов из 20 вариантов заданий.

Допущено Министерством образования Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению «Информатика и вычислительная техника».

ББК 32.973-018.1я7
УДК 681.3.06(075)

ISBN 5-94723-447-5

© ЗАО Издательский дом «Питер», 2003

Предисловие	10
Семинар 1. Линейные программы	13
Семинар 2. Разветвляющиеся программы. Циклы	30
Семинар 3. Одномерные массивы и указатели	55
Семинар 4. Двумерные массивы	71
Семинар 5. Строки и файлы	89
Семинар 6. Структуры	110
Семинар 7. Функции	132
Семинар 8. Перегрузка и шаблоны функций	161
Семинар 9. Динамические структуры данных	169
Приложение 1. Интегрированная среда Visual C++ 6.0	212
Приложение 2. Интегрированная среда Borland C++ 3.1 ...	227
Алфавитный указатель	236

Содержание

Предисловие	10
Семинар 1. Линейные программы	13
Задача 1.1. Расчет по формуле	13
Особенности работы в интегрированной среде Visual C++ 6.0	17
Отладка программы	19
Описание переменных	22
Задача 1.2. Временной интервал	24
Задания	26
Семинар 2. Разветвляющиеся программы. Циклы	30
Разветвляющиеся программы	30
Задача 2.1. Вычисление значения функции, заданной графически	30
Задача 2.2. Выстрел по мишени	33
Задача 2.3. Клавиши курсора	36
Циклы	37
Задача 2.4. Таблица значений функции	38
Задача 2.5. Вычисление суммы ряда	40
Задания	44
Семинар 3. Одномерные массивы и указатели	55
Задача 3.1. Количество элементов между минимумом и максимумом	56
Динамические массивы	59
Задача 3.2. Сумма элементов правее последнего отрицательного	60
Задача 3.3. Быстрая сортировка массива	63
Задания	67

Содержание

Семинар 4. Двумерные массивы	71
Задача 4.1. Среднее арифметическое и количество положительных элементов	73
Динамические массивы	76
Задача 4.2. Номер столбца из положительных элементов	77
Задача 4.3. Упорядочивание строк матрицы	81
Задания	85
Семинар 5. Строки и файлы	89
Описание строк	89
Ввод-вывод строк	90
Операции со строками	93
Работа с символами	94
Задача 5.1. Поиск подстроки	95
Задача 5.2. Подсчет количества вхождений слова в текст	98
Задача 5.3. Вывод вопросительных предложений	102
Задания	107
Семинар 6. Структуры	110
Задача 6.1. Поиск в массиве структур	111
Задача 6.2. Сортировка массива структур	118
Задача 6.3. Структуры и бинарные файлы	120
Задача 6.4. Структуры в динамической памяти	122
Задания	125
Семинар 7. Функции	132
Задача 7.1. Передача в функцию параметров стандартных типов	134
Задача 7.1-а. Передача в функцию имени функции	140
Задача 7.2. Передача одномерных массивов в функцию	142
Задача 7.3. Передача строк в функцию	144
Задача 7.4. Передача двумерных массивов в функцию	145
Задача 7.5. Передача структур в функцию	146
Задача 7.6. Рекурсивные функции	149
Многофайловые проекты	150
Что и как следует размещать в заголовочном файле	151
Задача 7.7. Многофайловый проект — форматирование текста	154
Задания	160
Семинар 8. Перегрузка и шаблоны функций	161
Перегрузка функций	161
Задача 8.1. Перегрузка функций	163

Шаблоны функций	165
Задача 8.2. Шаблоны функций	166
Задания	168
Семинар 9. Динамические структуры данных	169
Задача 9.1. Стек	170
Задача 9.2. Линейный список	173
Задача 9.3. Очередь	182
Задача 9.4. Бинарное дерево	188
Задания	203

Приложение 1. Интегрированная среда Visual C++ 6.0 212

Запуск IDE. Типы приложений	212
Создание нового проекта	214
Добавление к проекту файлов с исходным кодом	216
Добавление существующего файла	216
Добавление нового файла	216
Многофайловые проекты	218
Компиляция, компоновка и выполнение проекта	218
Проблемы с вводом-выводом кириллицы	220
Конфигурация проекта	221
Как закончить работу над проектом	221
Как открыть проект, над которым вы работали ранее	221
Встроенная справочная система	222
Работа с отладчиком	222
Установка точки прерывания	223
Выполнение программы до точки прерывания	224
Пошаговое выполнение программы	225
Проверка значений переменных во время выполнения программы	225
Окна Auto и Watch1	226

Приложение 2. Интегрированная среда Borland C++ 3.1 ... 227

Запуск IDE.....	227
Работа с меню	228
Меню File	228
Меню Edit	230
Меню Run	230
Меню Compile	231
Меню Debug	231
Меню Project	232

Меню Options	233
Меню Window	233
Создание нового проекта	233
Модификация существующего проекта	234
Открытие проекта	234
Работа с проектом	234
Завершение работы с проектом	234
Работа с отладчиком	235
Алфавитный указатель	236

Предисловие

Эта книга предназначена для изучения языка C++ на практических занятиях и лабораторных работах по программированию для студентов, обучающихся по направлению «Информатика и вычислительная техника» и смежным направлениям, а также для любознательных школьников, гениальных дошкольников, упорных пенсионеров и вообще всех, кто решил самостоятельно освоить этот язык. Она является дополнением к учебнику Т. А. Павловской «С/C++. Программирование на языке высокого уровня», выпущенному издательством «ПИТЕР» в 2001 году. В дальнейшем, исключительно для краткости, а не из излишнего питета, мы будем называть ее просто «Учебник».

Практикум состоит из двух книг — «Структурное программирование» и «Объектно-ориентированное программирование». В первой книге, которую вы держите в руках, рассматриваются возможности C++, используемые в рамках структурной парадигмы написания программ. Это стандартные типы данных, основные конструкции языка, массивы, строки и структуры, функции, шаблоны функций и динамические структуры данных. По содержанию книга соответствует материалу первой части Учебника. Методы создания объектно-ориентированных программ, шаблонов классов, использование исключений, а также классов и алгоритмов стандартной библиотеки будут рассмотрены во второй книге, которая, мы надеемся, будет выпущена издательством «ПИТЕР» в обозримом будущем.

В этой книге на примерах рассматриваются различные алгоритмы, методы и приемы написания программ, структуры данных, типичные ошибки, которые совершают начинающие (и не только) программисты, обсуждаются вопросы качества и стиля. Большое внимание уделяется процессу отладки и тестирования. Весь материал разбит на отдельные семинары. Логика изложения материала в основном соответствует Учебнику. Программа обучения рассчитана на 1–3 семестра: в зависимости от объема курса и степени подготовленности студентов количество и содержание работ можно варьировать.

Практикум рассчитан на первоначальное освоение языка C++, но может быть использован и для совершенствования, при этом основное внимание нужно уделить более сложным примерам. Предполагается, что студенты имеют возможность пользоваться Учебником, поэтому теоретические сведения в полном объеме здесь не приводятся. В начале каждого семинара приведены ссылки на разделы Учебника,

содержащие соответствующий материал. Тем не менее при разборе текстов программ поясняются все использованные в них возможности языка, поэтому практикум можно рассматривать и как самостоятельное издание. Тексты заданий на лабораторные работы большей частью соответствуют Учебнику, по некоторым разделам добавлены дополнительные комплекты заданий в двадцати вариантах. Практикум предназначен для изучения языка C++, но может быть использован и для освоения С, поскольку по каждой теме, там, где это имеет смысл, приводятся два варианта написания программы — в стиле C++ и в стиле С.

В Учебнике синтаксис языка C++ изложен в соответствии со стандартом ISO/IEC 14882 (1998). В настоящее время существует великое множество компиляторов C++, которые поддерживают этот стандарт в разной степени. Как правило, чем позже выпущен компилятор, тем соответствие стандарту лучше.

Необходимо различать язык, компилятор и среду программирования. Чаще всего бывает так, что стандарт выпускается после того, как язык получил достаточно широкое распространение и появилось множество компиляторов, каждый из которых реализует различные версии и расширения языка, что приводит к путанице и проблемам с переносимостью. В стандарте после обсуждений экспертами суммируются те свойства, которые должны поддерживаться всеми компиляторами, однако на практике это не означает, что даже компиляторы, выпущенные после утверждения стандарта, полностью ему соответствуют. Это объясняется, как правило, сложностью реализации и человеческим фактором.

Среда программирования объединяет компилятор, отладчик, редактор текста программ и другие средства, облегчающие разработку программ. Среды создаются под конкретные платформы (например, Unix или Windows). Как правило, в стандартных библиотеках, кроме функций, соответствующих стандарту, содержатся и платформенно-зависимые функции, предназначенные для разработки приложений для этой платформы. Использование таких функций снижает переносимость программ, но совершенно оправдано в тех случаях, когда она не требуется. В этой книге функции, не входящие в стандарт, не рассматриваются.

Различные среды программирования имеют разные интерфейсы, часто достаточно сложные для освоения. Наш практикум не рассчитан на обучение работе в конкретной среде, для этого существуют специальные, порой гораздо более объемные, руководства. Тем не менее для удобства пользования практикумом в приложениях приведены основные понятия и приемы работы в двух достаточно распространенных оболочках — Microsoft Visual Studio 6.0 и Borland C++ 3.1.

Отчего мы остановили выбор именно на этих программных продуктах? Компилятор C++, входящий в пакет Microsoft Visual Studio 6.0 (в дальнейшем для краткости — VC), широко распространен и в основном удовлетворяет стандарту. Однако известно, что далеко не каждый вуз или частное лицо могут позволить себе установку этого продукта, который достаточно требователен к ресурсам компьютера, не говоря уже о лицензионной чистоте. Поэтому мы включили в практикум описание более старого и гораздо более скромного по запросам компилятора — Borland C++ 3.1 (в дальнейшем для краткости — BC). Как показал опрос, этот крепкий и живучий старик вовсю трудится на древней технике, которой все еще оснащены

многие вузы и школы. Естественно, что некоторые примеры в нем компилироваться не будут.

Для успешного изучения языка C++ по этой книге пользоваться упомянутыми выше средами совершенно не обязательно. Собственно говоря, не обязательно вообще работать в какой-либо среде. Любой компилятор можно вызвать из командной строки, а для программ, состоящих из нескольких исходных файлов, воспользоваться утилитой *make*. Можно работать с любыми компиляторами, рассчитанными на любые платформы (различные версии Windows, Linux, Macintosh, Solaris, FreeBSD и так далее), но при этом необходимо проверять по справочной системе, что изучаемые возможности языка поддерживаются данным конкретным компилятором. В задачах, рассмотренных в этой книге, применяются только средства, соответствующие стандарту.

Если у вас есть доступ в Интернет, попробуйте для разнообразия использовать его не для развлечения: зайдите на поисковый сервер (к примеру, на <http://www.google.com>), введите фразу «free C++ compiler list», и вы получите море информации по различным средам и компиляторам. Например, хорошо поддерживает стандарт компилятор *gcc*, используемый в различных оболочках. Это бесплатный программный продукт, его можно скачать по сети и установить на своем компьютере.

В практикуме, так же, как и в Учебнике, не рассматривается программирование под Windows, поэтому все примеры представляют собой так называемые «консольные приложения». Этот термин поясняется в Приложении 1. Все примеры программ протестированы с использованием компилятора Microsoft Visual C++ 6.0.

Книга поддержана проектом «Разработка концепции и научно-методического обеспечения регионального центра целевой подготовки разработчиков программного обеспечения и компьютерных технологий» программы Министерства образования Российской Федерации «Государственная поддержка региональной научно-технической политики высшей школы и развития ее научного потенциала» на 2002 год. В основу книги положены семинары, проводимые авторами в Санкт-Петербургском государственном институте точной механики и оптики (Техническом университете).

Авторы выражают благодарность участникам конференции FIDO SU.C_CPP за советы и предложения. Ваши замечания, пожелания и дополнения к практикуму и к Учебнику не ленитесь присылать по адресу mux@tp2055.spb.edu.

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:

$$C = \frac{5}{9}(F - 32),$$

где *C* – температура по Цельсию, а *F* – температура по Фаренгейту.

Перед написанием любой программы надо четко определить, что в нее требуется ввести и что мы должны получить в результате. В данном случае сомнений нет: в качестве исходных данных выступает одно вещественное число, представляющее собой температуру по Цельсию, в качестве результата – другое вещественное число. Алгоритмических сложностей решение этой задачи также не представляет, поэтому давайте попробуем написать эту программу «в лоб» и посмотрим, что получится. Сразу же оговоримся, что первый блин будет комом, но тем программирование и отличается в лучшую сторону от реальной жизни, что ошибки можно найти и исправить.

```
#include <iostream.h> // 1
int main(){
    float fahr, cels;
    cout << endl << " Введите температуру по Фаренгейту" << endl; // 2
    cin >> fahr;
    cels = 5 / 9 * (fahr - 32); // 3
    cout << endl << " Температура по Цельсию = " << cels; // 4
} // 5
```

```

cout << " По Фаренгейту: " << fahr << ", в градусах Цельсия: "
    << cels << endl;                                // 6
return 0;                                         // 7
}

```

Рассмотрим каждую строку программы отдельно. Не расстраивайтесь, если что-то пока останется непонятным — у нас впереди целая книга и много времени, и вы непременно рано или поздно освоите все, что хотели.

В начале программы записана директива препроцессора¹, по которой к исходному тексту программы подключается заголовочный файл `<iostream.h>`. Это текстовый файл, который содержит описание элементов стандартной библиотеки, необходимых для выполнения ввода-вывода. Если говорить более конкретно, то в этом файле описан набор классов для управления вводом/выводом, стандартные объекты-потоки `cin` для ввода с клавиатуры и `cout` для вывода на экран, а также операции помещения в поток `<<` (вывода на экран) и извлечения из потока `>>` (ввода с клавиатуры). Объекты мы будем рассматривать во второй части практикума, а пока давайте пользоваться стандартными объектами и операциями с ними как некими волшебными словами, не пытаясь полностью осознать их смысл, ведь и в реальной жизни большинством благ цивилизации мы пользуемся подобным же образом.

ВНИМАНИЕ

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

Программа на C++ состоит из функций. Функция — это именованная последовательность операторов. Функция состоит из заголовка и тела. Стока, помеченная комментарием 1, 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). По стандарту главная функция должна возвращать целочисленное значение. Забегая вперед, скажем, что этим в нашей программе занимается оператор 7. Впрочем, многие компиляторы реагируют спокойно и в случае, если функция ничего не возвращает.

После круглых скобок в фигурных скобках записывается тело функции, то есть те операторы, которые требуется выполнить. Для удобства восприятия принято располагать тело с отступом в 3–4 позиции от заголовка. Обратите внимание, что закрывающая скобка находится в той же колонке, что и первый символ заголовка функции. Это требование хорошего стиля, а не синтаксиса.

¹ Препроцессором называется предварительная фаза компиляции, то есть перевода программы с C++ на машинный язык.

Для хранения исходных данных и результатов надо выделить достаточно места в оперативной памяти. Для этого служит оператор 2. 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 должно начинаться не с цифры. Оттого, насколько удачно подобраны имена в программе, зависит ее читаемость — одна из важнейших характеристик 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 хранитсяmantissa 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 важным¹.

Поскольку температура может принимать не только целые значения, для переменных выбран вещественный тип `float`. Можно также выбрать тип `double`, позволяющий представлять вещественные числа большего диапазона значений и с большей точностью, но для данной задачи это нам кажется излишней роскошью.

ПРИМЕЧАНИЕ

В общем случае тип переменных выбирается исходя из возможного диапазона значений и требуемой точности представления данных. Например, нет необходимости заводить переменную вещественного типа для хранения величины, которая может принимать только целые значения — хотя бы потому, что 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 3). На экран выводится указанная в операторе строка символов, и курсор переводится на следующую строку. Стока 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 `<<` 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 называ-

¹ Впрочем, существуют языки, в которых отсутствует предварительное описание типа, при этом он определяется по контексту. Тем, кто испытывает непреодолимое отвращение к описаниям типа, можно порекомендовать вместо C++, например, язык Perl.

емый манипулятор; он управляет, то есть «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 4 выполняется ввод с клавиатуры одного числа в переменную `fahr`. Для этого используется стандартный объект `cin` и операция извлечения (чтения) `>>`. Как видите, семантика ввода очень проста и интуитивно понятна: значение со стандартного ввода передается в переменную, указанную справа. Если требуется ввести несколько величин, используется цепочка операций `>>`. В процессе ввода число преобразуется из последовательности символов, набранных на клавиатуре, во внутреннее представление вещественного числа и помещается в ячейку памяти, зарезервированную для переменной `fahr`.

В операторе 5 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++ горчительно много, поэтому в случае сомнений надо не лениться обращаться к справочной информации, а вот запоминать подобные вещи в деталях мы бы не рекомендовали – есть и более полезные применения головному мозгу.

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 6 применяется уже знакомый нам объект `cout`. Выводится цепочка, состоящая из четырех элементов. Это строка "По Фаренгейту:", значение переменной `fahr`, строка ", в градусах Цельсия:" и значение переменной `cels`. Обратите внимание, что при выводе строк все символы, находящиеся внутри кавычек, включая и пробелы, выводятся без изменений. При выводе значений переменных выполняется преобразование из внутреннего представления числа в строку символов, представляющую это число. Под значение отводится ровно столько позиций, сколько необходимо для вывода всех его значащих цифр. Это значит, что если вывести две переменные подряд, их значения «склеятся», например:

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

СОВЕТ

Рекомендуется всегда предварять выводимые значения текстовыми пояснениями.

В одиночные кавычки в языке C++ заключается отдельный символ. В данном примере это пробел. Наконец, последний оператор (оператор 7) этой программы предназначен для возврата из нее и передачи значения во внешнюю среду. В главной функции разрешается его опускать, но в этом случае компилятор может выдать *предупреждение* (warning) и сформирует возвращаемое значение по умолчанию, равное нулю. Предупреждение можно проигнорировать, но, на наш взгляд, приятнее видеть в результате компиляции сообщение «0 errors, 0 warnings» (0 ошибок, 0 предупреждений) или «Success» (успешное завершение) – при большом количестве запусков программы это экономит время, не отвлекая внимания.

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

ВНИМАНИЕ

При работе в интегрированной среде Borland C++ 3.1 (или любой другой оболочке, рассчитанной на работу в среде MS DOS) у вас не будет проблем с компиляцией и выполнением приведенных примеров. Напротив, работа в среде Microsoft Visual C++ 6.0 в режиме консольных приложений сопряжена с неудобствами, вызванными различными стандартами кодировки символов кириллицы в операционных системах MS DOS и Windows. Ниже мы рассмотрим этот вопрос подробнее.

Особенности работы в интегрированной среде Visual C++ 6.0

Мы предполагаем, что вы уже посмотрели в Приложении 1, как создаются приложения консольного типа, набрали текст приведенной выше программы, откомпилировали ее и запустили на выполнение. Вам подготовлен неприятный сюрприз: вместо приглашения «Введите температуру по Фаренгейту» вы увидите набор каких-то странных символов. Этому, конечно, должно быть какое-то объяснение. Начнем издалека. В старой добре операционной системе MS DOS для кодировки символов используется стандарт ASCII, являющийся международным только в первой половине кодов (от 0 до 127), вторая половина кодов (от 128 до 255) является национальной и различна для разных стран.

Например, в России для второй половины таблицы ASCII используется так называемая «альтернативная кодировка ГОСТа». В Windows же используется стандарт ANSI, в первой половине совпадающий с ASCII, а во второй половине отличающийся от его российского варианта. Разработчики интегрированной среды Visual C++ решили, что режим консольных приложений должен как бы имитировать работу в среде MS DOS, поэтому ввод-вывод выполняется в этом режиме в кодировке ASCII. В то же время в текстовом редакторе Visual C++, как и во всех Windows-

приложениях, используется кодировка ANSI. Теперь происхождение этих странных символов должно стать вам понятным.

Мы выяснили, «кто виноват» в рассматриваемой проблеме, осталось ответить на кардинальный вопрос — «что делать?». Выходов, как всегда, два. Первый — перейти полностью на английский язык (что может оказаться плюсом в вашем резюме при поступлении на работу на совместное предприятие). Второй — использовать функцию `CharToOem()` для преобразования символов из кодировки ANSI в кодировку ASCII и функцию `OemToChar()` для обратного преобразования. Очевидно, что первая из названных функций нужна для вывода русскоязычного текста на экран, а вторая — для ввода такого текста с клавиатуры, если в дальнейшем потребуется запись этого текста в документы (файлы) с кодировкой ANSI¹. Чтобы использовать обе эти функции, вы должны подключить к вашей программе заголовочный файл `<windows.h>`, который не предусмотрен стандартом языка C++.

Покажем, как будет выглядеть наша первая программа при ее реализации в среде Visual C++ 6.0 (приводится одно из возможных решений):

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

char* Rus(const char* text); // 0

int main(){
    float fahr, cels;
    cout << endl << Rus(" Введите температуру по Фаренгейту") << endl; // 3
    cin >> fahr; // 4
    cels = 5 / 9 * (fahr - 32); // 5
    cout << Rus(" По Фаренгейту: ") << fahr;
    cout << Rus(". в градусах Цельсия: ") << cels << endl; // 6
    return 0; // 7
}
/////////////////////////////////////////////////////////////////
char bufRus[256]; // 8
char* Rus(const char* text){ // 9
    CharToOem(text, bufRus); // 10
    return bufRus; // 11
}
///////////////////////////////////////////////////////////////// // 12
```

Непосредственное использование функции `CharToOem()` не всегда удобно. Дело в том, что эта функция возвращает значение типа `BOOL`², а результат преобразования записывает по адресу своего второго аргумента. Поэтому вызов этой функции нельзя подставить на место строковой константы ни при использовании объектов вывода `cout`, ни при использовании библиотечных функций ввода-вывода.

¹ Эта функция пригодится нам на семинаре 6.

² Это один из стандартных типов системы Windows, совпадающий по значению с типом `bool` языка C++.

Для решения проблемы мы написали свою небольшую функцию¹ `Rus()`, которая, обращаясь к функции `CharToOem()`, возвращает в качестве адреса преобразованной строки указатель на `char` (строки 9–12). В качестве временного хранилища преобразуемой строки функция использует глобальный² массив `bufRus` длиной 256 символов (предполагается, что в качестве аргумента будут подставляться строки, не превышающие эту длину), описанный в операторе 8. Использовать функцию очень просто: любая строковая_константа заменяется выражением `Rus(строковая_константа)`. Кроме того, мы добавили в нашу программу так называемый *prototip* функции `Rus` (оператор 0). Он требуется компилятору для проверки правильности ее использования.

ПРИМЕЧАНИЕ

Строго говоря, для использования такой функции в операциях вывода с объектом `cout` было бы правильнее потребовать, чтобы функция `Rus()` возвращала значение типа ссылки на класс `ostream`, но для этого нужно владеть основами работы с классами, поэтому к этому вопросу мы вернемся во второй части практикума. То, что функция `Rus()` возвращает значение типа `char*`, приводит к следующему ограничению: ее нельзя использовать более одного раза в цепочке операций `<<` для одного объекта `cout`.

Проблема кодировок рассматривается также на шестом семинаре, там же приведен еще один пример ее решения. Во всех остальных программах, приведенных в этой книге, мы делаем вид, что этой проблемы не существует, поскольку к изучению собственно языка она отношения не имеет. Вернемся к тестированию и отладке нашей программы.

Отладка программы

Запустите программу на выполнение несколько раз, задавая различные значения температуры. Не забудьте, что число, имеющее дробную часть, при вводе следует записывать с точкой, а не с 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 4. Константы 5 и 9 имеют целый тип, поэтому результат их деления также целочисленный. Округления при этом не происходит, дробная часть всегда отбрасывается. 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, например:

```
cels = 5. / 9 * (fahr - 32); // 5
```

Вещественная константа `<5.>` по умолчанию имеет тип `double`, и при выполнении деления происходит автоматическое преобразование к этому же типу другой константы, а затем и результата вычитания.

¹ Вопросам разработки функций посвящен семинар 7, а пока мы предлагаем вам просто принять наше решение на веру.

² О глобальных переменных мы расскажем чуть позже, в разделе «Описание переменных».

Можно записать это выражение и по-другому, просто изменив порядок действий:

```
cels = 5 * (fahr - 32) / 9; // 5
```

В этом случае будет выполнено преобразование констант к типу float, как к наиболее длинному из участвующих в выражении. .

Теперь программа работает верно и выдает в результате, например, следующее:

```
Введите температуру по Фаренгейту  
451  
По Фаренгейту: 451. в градусах Цельсия: 232.778
```

Обратите внимание, что на экран 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. Рекомендуется для контроля выводить исходные данные сразу же после ввода, чтобы исключить ошибки.

Попробуйте при вводе температуры задать нецифровые символы и проинтерпретировать полученный результат.

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, на бумаге или с помощью калькулятора, 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

Перед запуском программы необходимо иметь тестовые примеры, содержащие исходные данные и ожидаемые результаты. Количество этих 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 вторым способом, с использованием функций библиотеки C++, унаследованных из языка С. Этот способ также применяется достаточно часто, потому что в использовании этих функций есть свои преимущества. Когда вы их оцените, то сможете выбирать для каждой программы наиболее подходящий способ ввода-вывода.

```
#include <stdio.h>  
int main(){  
    float fahr, cels;  
    printf("\n Введите температуру по Фаренгейту\n"); // 1  
    // 2  
    // 3
```

```
scanf("%f", &fahr); // 4  
cels = 5 * (fahr - 32) / 9; // 5  
printf(" По Фаренгейту: %6.2f. в градусах Цельсия: %6.2f\n", fahr, // 6  
      cels); // 7  
return 0;  
}
```

Как видите, к программе подключается другой заголовочный файл — `<stdio.h>`. Он содержит описание функций, констант и других элементов, относящихся к вводу-выводу «в стиле С».

Рассмотрим отличия этой программы от предыдущей.

Функция `printf` в операторе 3 выполняет вывод переданного ей в качестве параметра 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. Эта последовательность задает переход на следующую строку.

Для ввода исходных данных в операторе 4 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 `%f` соответствует типу `float`¹. В качестве второго параметра функции передается адрес переменной, по которому будет помещено вводимое значение. Операция взятия адреса обозначается `&`.

Для вывода результата в операторе 6 применяется уже знакомая нам 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. Все символы литерала, кроме спецификаций формата `%f` и управляющей последовательности `\n`, выводятся на дисплей без изменений. При выводе форматные спецификации будут заменены конкретными значениями переменных `fahr` и `cels`.

Формат вывода чисел можно уточнить 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 дробную часть величины. Необходимо учитывать, что десятичная точка тоже занимает одну позицию. Если заданного количества позиций окажется недостаточно для размещения числа, компилятор простит нам этот промах и автоматически выделит поле достаточной длины.

На наш взгляд, в данном случае первый вариант программы (с использованием классов) более прост и нагляден, а также лучше защищен от ошибок кодирования.

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

С другой стороны, с помощью функций в стиле С легче управлять видом выводимой информации. Однако при записи форматов легко ошибиться, и, к сожалению, компилятор ничего об этом не сообщит; вы будете ломать голову в поисках алгоритмической ошибки, в то время как источником неприятностей будет функция `printf`.

Надо сказать, что при использовании классов также можно задавать любую форму представления информации, но мы пока не будем вдаваться в подобные детали. Мы не рекомендуем начинающим изучать на этом семинаре теоретический материал по классам ввода/вывода, поскольку для их полного понимания требуется освоить большой объем материала. Всему свое время!

Для каждой программы необходимо выбрать один наиболее удобный способ вывода (либо с помощью функций, либо с помощью классов), поскольку смешивать оба варианта не рекомендуется.

СОВЕТ

Используйте функции ввода/вывода в тех программах, 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*, размещается в сегменте данных¹ и изначально обнуляется, если вы не предусмотрели ее инициализацию каким-то другим значением. Время жизни глобальной переменной — 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 сегменте стека в момент выполнения оператора описания. Для локальных переменных автоматическое обнуление не производится. Областью действия локальной переменной 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.

¹ В IBM PC-совместимых компьютерах память представляется разделенной на сегменты. Исполняемая программа состоит из сегментов кода, в которых расположены машинные команды, сегментов данных и сегмента стека. Остальная доступная программе память называется динамической (а также хипом или кучей).

Есть один вид локальных переменных, которые, подобно глобальным, размещаются в сегменте данных, существуют на всем протяжении выполнения программы и инициализируются однократно. Это *статические* переменные. 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	а	б	с
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`). 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. Вариант с глобальными переменными выглядит так:

```
#include <iostream.h>
float fahr, cels; // глобальные переменные
int main(){
    cout << endl << "Введите температуру по Фаренгейту" << endl;
    cin >> fahr;
    cels = 5 * (fahr - 32) / 9;
    cout << "По Фаренгейту: " << fahr << ". в градусах Цельсия: " << cels << endl;
    return 0;
}
```

Для данной простой программы разницы в этих способах нет, но в общем случае 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 нам предстоит неоднократно в этом убедиться.

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 единицах¹.

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 иметь весьма большие значения, ведь в сутках 86400 секунд. В величинах типа `short` могут храниться значения, не превышающие 32767 для величин со знаком (`signed short`) и 65535 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.

Ниже приведен текст программы:

```
#include <iostream.h>
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;
    return 0;
}
```

¹ Задача позаимствована из книги: Юрик А. Г. Задачник по программированию. — СПб.: Питер, 2002.

Задача 1.2. Временной интервал

Для перевода результата из секунд обратно в часы и минуты используется уже знакомый нам эффект отбрасывания дробной части при делении целого числа на целое.

ВНИМАНИЕ

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

Протестируйте программу на различных наборах исходных данных.

Для обладателей старых компиляторов приведем текст программы с использованием функций ввода-вывода в стиле C:

```
#include <stdio.h>
int main(){
    short hour1, min1, sec1, hour2, min2, sec2, hour, min, sec;
    printf(" Введите время начала интервала (час мин сек)\n");
    scanf("%i%i%i", &hour1, &min1, &sec1);
    printf(" Введите время конца интервала (час мин сек)\n");
    scanf("%i%i%i", &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;

    printf(" Длительность промежутка от %i:%i:%i до %i:%i:%i\n", hour1,
        min1, sec1, hour2, min2, sec2);
    printf(" равна %i:%i:%i\n", hour, min, sec);

    return 0;
}
```

Давайте повторим наиболее важные моменты этого семинара.

1. Выбирайте тип переменных с учетом диапазона и требуемой точности представления данных.
2. Давайте переменным имена, отражающие их назначение.
3. Ввод с клавиатуры предваряйте приглашением. Для контроля сразу же после ввода выводите исходные данные на дисплей (по крайней мере, в процессе отладки).
4. До запуска программы подготовьте тестовые примеры, содержащие исходные данные и ожидаемые результаты. Отдельно проверьте реакцию программы на неверные исходные данные.
5. При записи выражений обращайте внимание на приоритет операций.
6. 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.

7. При использовании стандартных функций или классов требуется с помощью директивы `#include` подключить к программе соответствующие заголовочные файлы. Установить, какой именно файл необходим, можно с помощью справочной системы¹.
8. Не смешивайте в одной программе ввод/вывод с помощью классов (в стиле C++) и с помощью функций библиотеки (в стиле C).
9. Отдавайте предпочтение локальным переменным перед глобальными. Переменная должна иметь минимальную из возможных областей действия.
10. Данные при вводе разделяйте пробелами, символами перевода строки или табуляции.

Задания

Напишите программу для расчета по двум формулам. Предварительно подготовьте тестовые примеры по второй формуле с помощью калькулятора (результат вычисления по первой формуле должен совпадать со второй). Список математических функций библиотеки C++ приведен в Учебнике на с. 410. Для их использования необходимо подключить к программе заголовочный файл `<math.h>`.

Вариант 1

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

$$z_2 = \frac{1}{4} - \frac{1}{4}\sin\left(\frac{5}{2}\pi - 8\alpha\right)$$

Вариант 2

$$z_1 = \cos\alpha + \sin\alpha + \cos 3\alpha + \sin 3\alpha$$

$$z_2 = 2\sqrt{2}\cos\alpha \cdot \sin\left(\frac{\pi}{4} + 2\alpha\right)$$

Вариант 3

$$z_1 = \frac{\sin 2\alpha + \sin 5\alpha - \sin 3\alpha}{\cos\alpha + 1 - 2\sin^2 2\alpha}$$

$$z_2 = 2\sin\alpha$$

¹ Например, в VC или BC для этого требуется установить курсор на имя функции и нажать F1 или CTRL-F1 соответственно.

Вариант 4

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

$$z_2 = \operatorname{tg} 3\alpha$$

Вариант 5

$$z_1 = 1 - \frac{1}{4}\sin^2 2\alpha + \cos 2\alpha$$

$$z_2 = \cos^2\alpha + \cos^4\alpha$$

Вариант 6

$$z_1 = \cos\alpha + \cos 2\alpha + \cos 6\alpha + \cos 7\alpha$$

$$z_2 = 4\cos\frac{\alpha}{2} \cdot \cos\frac{5}{2}\alpha \cdot \cos 4\alpha$$

Вариант 7

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

$$z_2 = \frac{\sqrt{2}}{2}\sin\frac{\alpha}{2}$$

Вариант 8

$$z_1 = \cos^4 x + \sin^2 y + \frac{1}{4}\sin^2 2x - 1$$

$$z_2 = \sin(y+x) \cdot \sin(y-x)$$

Вариант 9

$$z_1 = (\cos\alpha - \cos\beta)^2 - (\sin\alpha - \sin\beta)^2$$

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

Вариант 10

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

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

Вариант 11

$$z_1 = \frac{1 - 2\sin^2 \alpha}{1 + \sin 2\alpha}$$

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

Вариант 12

$$z_1 = \frac{\sin 4\alpha}{1 + \cos 4\alpha} \cdot \frac{\cos 2\alpha}{1 + \cos 2\alpha}$$

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

Вариант 13

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

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

Вариант 14

$$z_1 = \frac{\cos \alpha + \sin \alpha}{\cos \alpha - \sin \alpha}$$

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

Вариант 15

$$z_1 = \frac{\sqrt{2b + 2\sqrt{b^2 - 4}}}{\sqrt{b^2 - 4 + b + 2}}$$

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

Задания**Вариант 16**

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

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

Вариант 17

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

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

Вариант 18

$$z_1 = \left(\frac{a+2}{\sqrt{2a}} - \frac{a}{\sqrt{2a}+2} + \frac{2}{a-\sqrt{2a}} \right) \cdot \frac{\sqrt{a}-\sqrt{2}}{a+2}$$

$$z_2 = \frac{1}{\sqrt{a}+\sqrt{2}}$$

Вариант 19

$$z_1 = \left(\frac{1+a+a^2}{2a+a^2} + 2 - \frac{1-a+a^2}{2a-a^2} \right)^{-1} (5-2a^2)$$

$$z_2 = \frac{4-a^2}{2}$$

Вариант 20

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

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

СЕМИНАР 2

Разветвляющиеся программы. Циклы

Разветвляющиеся программы

Теоретический материал: с. 40–44.

В линейной программе все операторы выполняются последовательно, один за другим. Для того чтобы в зависимости от 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 числа 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).

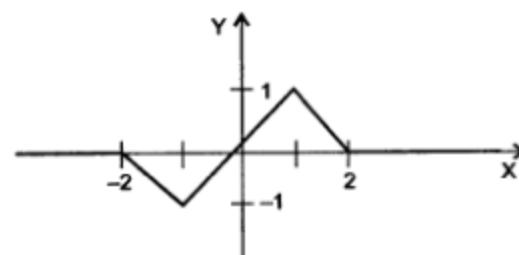


Рис. 2.1. Функция для задачи 2.1

Начинать решение даже простейшей задачи необходимо с четкого описания того, что является ее исходными данными и результатами. В данном случае это очевидно: исходными данными 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 «нисходящая разработка», позволяет создавать простые по структуре программы. По мере приобретения опыта вы убедитесь, насколько это важно. Как известно, алгоритм решения задачи можно описать в различном виде, например, в словесном или в виде блок-схемы. А нам удобнее будет для начала записать функцию в виде формул:

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

Ниже приведено описание алгоритма в неформальной словесной форме. Этот способ мы всячески рекомендуем, потому что только после того, как задача четко описана на естественном языке, ее можно успешно записать на языке программирования. Для такой простой задачи, как рассматриваемая, это, быть может, и не имеет смысла, но вы ведь не собираетесь всю жизнь писать только элементарные программы!

1. Ввести значение аргумента x .
2. Определить, какому интервалу из области определения функции оно принадлежит.
3. Вычислить значение функции y по соответствующей формуле.
4. Вывести значение y .

Детализировать этот алгоритм уже практически некуда, поэтому сразу же смело напишем первый вариант программы:

```
#include <iostream.h>
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;
    return 0;
}
```

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).

Как видно из программы, с помощью последовательности условных операторов практически «один в один» записана математическая форма описания функции, приведенная выше. Обратите внимание на запись условий, содержащих два сравнения.

Операции отношения ($<$, $>$, -- , \leq , \geq , $!=$) являются бинарными, то есть имеют два операнда, и формируют результат типа `bool`, равный `true` или `false`¹. 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, поэтому компилятор не выдает каких-либо сообщений. 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 соответствует ожиданиям.

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;
```

Проверка на принадлежность аргумента очередному интервалу выполняется только в том случае, если x не входит в предыдущий интервал. Программа получилась более компактной, более эффективной, но, возможно, менее наглядной. В отличие от предыдущей версии, порядок следования условных операторов имеет здесь важное значение. Рассмотрим еще один вариант:

```
y = 0;
if ( x > -2 ) y = -x - 2;
```

¹ В компиляторах, не поддерживающих тип `bool`, истинным считается любое значение, не равное нулю.

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

Запись фрагмента стала еще короче, но появились два недостатка: значение функции вычисляется многократно (от двух до пяти раз в зависимости от интервала, которому принадлежит x), значит, увеличилось время выполнения, а главное – программа потеряла универсальность, то есть таким способом можно вычислить не всякую функцию. Для того чтобы в этом убедиться, попробуйте заменить $y=x$ на, к примеру, фрагмент окружности $y=\sqrt{1-x^2}$ и ввести значение x , большее 1 – это приведет к ошибке в программе, связанной с вычислением квадратного корня из отрицательной величины.

Какой же вариант лучше? Для решения данной задачи разница между ними несущественна, но наша цель состоит в том, чтобы на простых примерах продемонстрировать общие принципы, следование которым позволит вам впоследствии создавать надежные и красивые программы.

СОВЕТ

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, наиболее наглядный вариант предпочтительнее.

Нам кажется, что наиболее наглядным является самый первый вариант программы, поскольку по нему проще проследить логику ее работы.

В заключение приведем вариант с использованием функций ввода-вывода в стиле C:

```
#include <stdio.h>
int main(){
    float x, y;
    printf(" Введите значение аргумента:\n");
    scanf("%f", &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;
    printf(" Для x = %5.2f значение функции y = %5.2f\n", x, y);
    return 0;
}
```

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.

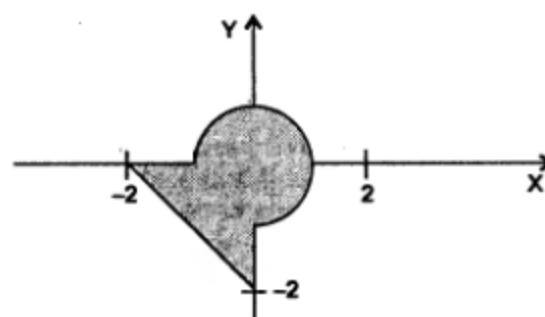


Рис. 2.2. Графически заданная область для задачи 2.2

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:

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

Первое условие задает попадание точки в круг, второе — в треугольник. Программа для решения задачи выглядит следующим образом:

```
#include <iostream.h>
int main()
{
    float 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;
    return 0;
}
```

Три условия из правых фигурных скобок должны выполняться одновременно, поэтому в программе они объединяются с помощью операции И. Ее приоритет выше, чем у ИЛИ (`||`), и ниже, чем у операций отношения, поэтому дополнительных скобок не требуется.

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";
```

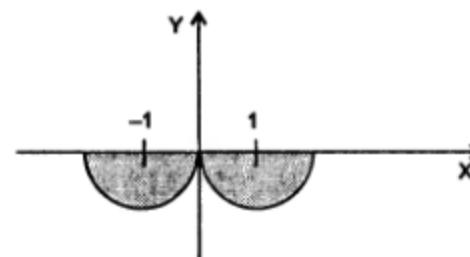


Рис. 2.3. Пример еще одной области

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.

ПРИМЕЧАНИЕ

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

Выполните программу несколько раз, задавая различные положения точки. Задайте заштрихованную область какого-либо другого вида и измените программу в соответствии с этой областью.

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 оператором¹.

ВНИМАНИЕ

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!
```

¹ Эта возможность в старых компиляторах не реализована.

Значение величины, с которой сравнивается модуль разности, следует выбирать в зависимости от решаемой задачи и точности участвующих в выражении переменных. Снизу эта величина ограничена определенными в заголовочном файле `<float.h>` константами `FLT_EPSILON = 1.192092896e-07F` и `DBL_EPSILON = 2.2204460492503131e-016`. (`FLT_EPSILON` – это минимально возможное значение переменной типа `float`, такое, что `1.0 + FLT_EPSILON != 1.0`, `DBL_EPSILON` – аналогичная константа для типа `double`).

Для присваивания какой-либо переменной в зависимости от выполнения условия двух различных значений 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 operand 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 operand 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`.

Давайте теперь рассмотрим простой пример применения оператора `switch`.

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. В случае нажатия функциональных или курсорных клавиш эта функция возвращает 0 либо 0xE0 (в зависимости от компилятора), а ее повторный вызов позволяет получить расширенный код клавиши.

```
#include <stdio.h>
#include <conio.h>
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");
    }
    return 0;
}
```

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, например:

```
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 автоматически передано на первый оператор следующей ветви, необходимо после каждой ветви использовать оператор `break`.

СОВЕТ

Хотя наличие слова `default` и 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 – участок программы, повторяемый многократно. В C++ три взаимозаменяемых оператора цикла – `while`, `do while` и `for`. При написании любого цикла надо



Рис. 2.4. Блок-схема цикла

иметь в виду, что в нем всегда явно или неявно присутствуют четыре элемента: начальные установки, тело цикла, модификация параметра цикла и проверка условия продолжения цикла (рис. 2.4). Начинающие чаще всего забывают про первое и/или третье.

Задача 2.4. Таблица значений функции

Написать программу печати таблицы значений функции

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

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

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 словесной форме алгоритм можно сформулировать так:

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

В каждый момент времени требуется хранить одно значение функции, поэтому для него достаточно завести одну переменную вещественного типа. Шаги 3–7 повторяются многократно, поэтому для их выполнения надо организовать цикл. В приведенном ниже варианте используется цикл `while`:

```
#include <stdio.h>
#include <math.h>
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");
    while (Xn <= Xk) {
        if (Xn < 0) y = t;
        if (Xn >= 0 && Xn < 10) y = t * Xn;
        if (Xn >= 10) y = 2 * t;
        if (t > 100) printf("%9.2f %9d \n", Xn, (int)y);
        else printf("%9.2f %9.2f \n", Xn, y);
        Xn += dX;
    }
    printf("----- \n");
    return 0;
}
```

Задача 2.4. Таблица значений функции

```
printf("----- \n");
double x = Xn; // Начальные установки
while (x <= Xk){
    if (x < 0) y = t;
    if (x >= 0 && x < 10) y = t * x;
    if (x >= 10) y = 2 * t;
    if (t > 100) printf("%9.2f %9d \n", x, (int)y);
    else printf("%9.2f %9.2f \n", x, y);
    x += dX; // Модификация параметра цикла
}
printf("----- \n");
return 0;
}
```

А вот та же программа с использованием оператора `for`:

```
#include <stdio.h>
#include <math.h>
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("%9.2f %9d \n", x, (int)y);
        else printf("%9.2f %9.2f \n", x, y);
    }
    printf("----- \n");
    return 0;
}
```

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 первом варианте программы область видимости x простирается от точки описания до конца программы, во втором областью ее видимости является только цикл¹, что предпочтительнее, поскольку переменная x вне цикла не требуется. Вообще

¹ В старых версиях компиляторов (и даже в VC!) переменная видна и после цикла.

говоря, в условии цикла `while` допускается описывать и инициализировать переменную таким же образом, как в операторе `if`, но при этом синтаксис не допускает ее сравнения с `Xk`. Другим преимуществом второго варианта программы является то, что все управление циклом `for` сосредоточено в его заголовке. Это делает программу более читаемой.

Для преобразования значения функции к целому в программе использовалась конструкция `(int)y`, унаследованная из языка С. Строго говоря, в данном случае лучше применить операцию преобразования типа `static_cast`, введенную в C++¹, но в старых компиляторах она может не поддерживаться:

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

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

Рассмотрим теперь пример цикла, количество итераций которого заранее подсчитать невозможно².

Задача 2.5. Вычисление суммы ряда

Написать программу вычисления значения функции $Ch x$ (гиперболический косинус) с помощью бесконечного ряда Тейлора с точностью ϵ по формуле:

$$y = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \dots + \frac{x^{2n}}{2n!} + \dots$$

Этот ряд сходится при $|x| < \infty$. Для достижения заданной точности требуется суммировать члены ряда, абсолютная величина которых больше ϵ . Для сходящегося ряда модуль члена ряда C_n при увеличении n стремится к нулю. При некотором n неравенство $|C_n| \geq \epsilon$ перестает выполняться, и вычисления прекращаются.

Общий алгоритм решения этой задачи очевиден: 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 зна-

¹ Об этой операции рассказывается в Учебнике на с. 237.

² Аналогичный пример приведен в Учебнике на с. 50.

менатель, 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, ограничено¹. Второй недостаток связан с эффективностью вычислений: как легко заметить, при вычислении очередного члена ряда нам 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 эту формулу C_n и C_{n+1} , получим выражение для вычисления T :

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

Ниже приведен текст программы с комментариями. Обратите внимание на заголовок цикла: в нем, кроме части начальных установок, задано условие выхода из цикла и модификация параметра цикла, то есть в данной программе все составные части цикла присутствуют в явном виде.

```
#include <iostream.h>
#include <math.h>
int main(){
    const int MaxIter = 500; /* максимально допустимое количество итераций */
    double x, eps;
    cout << "\nВведите аргумент и точность: ";
    cin >> x >> eps;
    bool done = true; // признак достижения точности
    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)); // 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;
        cout << "вычислено после " << n << " итераций" << endl;
    }
    return 0;
}
```

¹ Кроме того, большие числа могут переполнить разрядную сетку.

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 как в случае аварийного, так и в случае нормального завершения цикла происходит в одну и ту же точку программы, вводится булева переменная *done*, которая предотвращает печать значения функции после выхода из цикла в случае, когда 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.

Можно реализовать ту же логику и без специальной булевой переменной, объединив проверку обоих вариантов выхода из цикла в его заголовке:

```
#include <iostream.h>
#include <math.h>
#include <float.h>
int main(){

    const int MaxIter = 500; /* максимально допустимое количество
                           итераций */
    double x, eps = DBL_EPSILON;
    cout << "\nВведите аргумент": cin >> x :

    double ch = 1, y = ch; // первый член ряда и нач. значение суммы
    for (int n = 0; fabs(ch) > eps && n < MaxIter; n++) {
        ch *= x * x / ((2 * n + 1)*(2 * n + 2)); // очередной член ряда
        y += ch;
    }
    if (n < MaxIter) {
        cout << "\nЗначение функции: " << y << " для x = " << x << endl;
        cout << "вычислено после " << n << " итераций" << endl;
    }
    else cout << "\nРяд расходится!":

    return 0;
}
```

В этом варианте программы сумма ряда для разнообразия вычисляется с максимально возможной точностью.

В библиотеке есть функция *cosh(x)*, вычисляющая гиперболический косинус. Ее прототип находится в файле *<math.h>*. Измените программу так, чтобы она рядом

со значением, вычисленным через разложение в ряд, печатала и значение, определенное с помощью стандартной функции. Сравните результаты вычислений.

Итак, мы рассмотрели две задачи с использованием циклов. Циклы встречаются в программах повсеместно, и строятся они по одним и тем же принципам. Чтобы избежать ошибок при программировании, рекомендуется:

- проверить, всем ли переменным, встречающимся в правой части операторов присваивания в теле цикла, присвоены до этого начальные значения, а также возможно ли выполнение других операторов;
- проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла;
- предусмотреть аварийный выход из цикла по достижении некоторого количества итераций;
- не забывать заключать в фигурные скобки тело цикла, если в нем требуется выполнить более одного оператора.

Операторы цикла в языке C++ взаимозаменяемы, но можно привести некоторые рекомендации по выбору наилучшего в каждом конкретном случае.

Оператор *do while* обычно используют, когда цикл требуется обязательно выполнить хотя бы один раз, — например, если в цикле производится ввод данных.

Оператором *while* удобнее пользоваться в тех случаях, когда либо число итераций заранее неизвестно, либо очевидных параметров цикла нет, либо модификацию параметров удобнее записывать не в конце 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).

Давайте повторим наиболее важные моменты этого семинара.

1. Выражение, стоящее в круглых скобках операторов *if*, *while* и *do while*, вычисляется в соответствии с приоритетами операций и преобразуется к типу *bool*.
2. Если в какой-либо ветви вычислений условного оператора или в цикле требуется выполнить более одного оператора, то они объединяются в блок.
3. Проверка вещественных величин на равенство опасна.
4. Чтобы получить максимальную читаемость и простоту структуры программы, надо правильно выбирать способ реализации ветвлений (с помощью *if*, *switch* или условной операции), а также наиболее подходящий оператор цикла.
5. Выражение, стоящее в скобках после ключевого слова *switch*, и константные выражения в *case* должны быть целочисленного типа.
6. Рекомендуется всегда описывать в операторе *switch* ветвь *default*.
7. После каждой ветви для передачи управления на конец оператора *switch* используется оператор *break*.

8. 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 проверка условия продолжения цикла.
9. Если количество повторений цикла заранее не известно, необходимо предусматривать аварийный выход из цикла по достижении некоторого достаточно большого количества итераций.

Задания

Вариант 1

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 + b & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0 \text{ и } b = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац ИЛИ Вц) И (Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c, операции И и ИЛИ – поразрядные. Значения a, b, c, Xнач., Xкон., dX ввести с клавиатуры.

Вариант 2

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} \frac{1}{ax} - b & \text{при } x + 5 < 0 \text{ и } c = 0 \\ \frac{x-a}{x} & \text{при } x + 5 > 0 \text{ и } c \neq 0 \\ \frac{10x}{c-4} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение (Ац И Вц) ИЛИ (Вц И Сц)

Задания

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c, операции И и ИЛИ – поразрядные. Значения a, b, c, Xнач., Xкон., dX ввести с клавиатуры.

Вариант 3

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 + bx + c & \text{при } a < 0 \text{ и } c \neq 0 \\ \frac{-a}{x-c} & \text{при } a > 0 \text{ и } c = 0 \\ a(x+c) & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение Ац И (Вц ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c, операции И и ИЛИ – поразрядные. Значения a, b, c, Xнач., Xкон., dX ввести с клавиатуры.

Вариант 4

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} -ax - c & \text{при } c < 0 \text{ и } x \neq 0 \\ \frac{x-a}{-c} & \text{при } c > 0 \text{ и } x = 0 \\ \frac{bx}{c-a} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение Ац ИЛИ Вц ИЛИ Сц

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c, операция ИЛИ – поразрядная. Значения a, b, c, Xнач., Xкон., dX ввести с клавиатуры.

Вариант 5

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} a - \frac{x}{10 + b} & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x - a}{x - c} & \text{при } x > 0 \text{ и } b = 0 \\ 3x + \frac{2}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение $(Aц \text{ ИЛИ } Bц) \text{ И } Cц$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и ИЛИ – поразрядные. Значения $a, b, c, Xнач., Xкон., dX$ ввести с клавиатуры.

Вариант 6

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 + b^2x & \text{при } c < 0 \text{ и } b \neq 0 \\ \frac{x + a}{x + c} & \text{при } c > 0 \text{ и } b = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение $(Aц \text{ И } Bц) \text{ ИЛИ } (Aц \text{ И } Cц)$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и ИЛИ – поразрядные. Значения $a, b, c, Xнач., Xкон., dX$ ввести с клавиатуры.

Вариант 7

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} -ax^2 - b & \text{при } x < 5 \text{ и } c \neq 0 \\ \frac{x - a}{x} & \text{при } x > 5 \text{ и } c = 0 \\ -\frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение

$(Aц \text{ ИЛИ } Bц) \text{ МОД2 } (Aц \text{ ИЛИ } Cц)$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И, ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, Xнач., Xкон., dX$ ввести с клавиатуры.

Вариант 8

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} -ax^2 & \text{при } c < 0 \text{ и } a \neq 0 \\ \frac{a - x}{cx} & \text{при } c > 0 \text{ и } a = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение

$(Aц \text{ МОД2 } Bц) \text{ И } \text{НЕ}(Aц \text{ ИЛИ } Cц)$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И, ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, Xнач., Xкон., dX$ ввести с клавиатуры.

Вариант 9

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 + b^2x & \text{при } a < 0 \text{ и } x \neq 0 \\ x - \frac{a}{x - c} & \text{при } a > 0 \text{ и } x = 0 \\ 1 + \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение

$\text{НЕ}(Aц \text{ ИЛИ } Bц) \text{ И } (Bц \text{ ИЛИ } Cц)$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ, И и ИЛИ – поразрядные. Значения $a, b, c, Xнач., Xкон., dX$ ввести с клавиатуры.

Вариант 10

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 - bx + c & \text{при } x < 3 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 3 \text{ и } b = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение

$\text{НЕ(Ац ИЛИ Вц)} \text{ И } (\text{Ац МОД2 Сц})$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ, И, ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, X\text{нач.}, X\text{кон.}, dX$ ввести с клавиатуры.

Вариант 11

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 + \frac{b}{c} & \text{при } x < 1 \text{ и } c \neq 0 \\ \frac{x-a}{(x-c)^2} & \text{при } x > 1.5 \text{ и } c = 0 \\ \frac{x^2}{c^2} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение

$(\text{Ац И Вц}) \text{ МОД2 Сц}$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, X\text{нач.}, X\text{кон.}, dX$ ввести с клавиатуры.

Вариант 12

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

Задания

$$F = \begin{cases} ax^3 + b^2 + c & \text{при } x < 0.6 \text{ и } b + c \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0.6 \text{ и } b + c = 0 \\ \frac{x}{c} + \frac{x}{a} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение $(\text{Ац ИЛИ Вц}) \text{ И Сц}$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И и ИЛИ – поразрядные. Значения $a, b, c, X\text{нач.}, X\text{кон.}, dX$ ввести с клавиатуры.

Вариант 13

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} ax^2 + b & \text{при } x - 1 < 0 \text{ и } b - x \neq 0 \\ \frac{x-a}{x} & \text{при } x - 1 > 0 \text{ и } b + x = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение $(\text{Ац ИЛИ Вц}) \text{ МОД2 (Вц И Сц)}$

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции И, ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, X\text{нач.}, X\text{кон.}, dX$ ввести с клавиатуры.

Вариант 14

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Xнач. до Xкон. с шагом dX.

$$F = \begin{cases} -ax^3 - b & \text{при } x + c < 0 \text{ и } a \neq 0 \\ \frac{x-a}{x-c} & \text{при } x + c > 0 \text{ и } a = 0 \\ \frac{x}{c} + \frac{c}{x} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение
(Ац МОД2 Вц) ИЛИ (Ац МОД2 Сц)
не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 15

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} -ax^2 + b & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x}{x - c} + 5.5 & \text{при } x > 0 \text{ и } b = 0 \\ \frac{x}{-c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение
НЕ(Ац ИЛИ Вц ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ и ИЛИ – поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 16

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} a(x + c)^2 - b & \text{при } x = 0 \text{ и } b \neq 0 \\ \frac{x - a}{-c} & \text{при } x = 0 \text{ и } b = 0 \\ a + \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение
(Ац МОД2 Вц) И НЕ(Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ, И, ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 17

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} ax^2 - cx + b & \text{при } x + 10 < 0 \text{ и } b \neq 0 \\ \frac{x - a}{x - c} & \text{при } x + 10 > 0 \text{ и } b = 0 \\ \frac{-x}{a - c} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение
(Ац ИЛИ Вц) И НЕ(Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ, И и ИЛИ – поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 18

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} ax^3 + bx^2 & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x - a}{x - c} & \text{при } x > 0 \text{ и } b = 0 \\ \frac{x + 5}{c(x - 10)} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение
НЕ(Ац И Вц И Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ и И – поразрядные. Значения $a, b, c, X_{\text{нач.}}, X_{\text{кон.}}, dX$ ввести с клавиатуры.

Вариант 19

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от $X_{\text{нач.}}$ до $X_{\text{кон.}}$ с шагом dX .

$$F = \begin{cases} a(x+7)^2 - b & \text{при } x < 5 \text{ и } b \neq 0 \\ \frac{x-cd}{ax} & \text{при } x > 5 \text{ и } b = 0 \\ \frac{x}{c} & \text{в остальных случаях} \end{cases}$$

где a, b, c, d – действительные числа.

Функция F должна принимать действительное значение, если выражение

(Ац МОД2 Вц) ИЛИ (Ац МОД2 Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции ИЛИ и МОД2 (сложение по модулю 2) – поразрядные. Значения a, b, c, d , Хнач., Хкон., дХ ввести с клавиатуры.

Вариант 20

Вычислить и вывести на экран в виде таблицы значения функции F на интервале от Хнач. до Хкон. с шагом дХ.

$$F = \begin{cases} \frac{-2x-c}{cx-a} & \text{при } x < 0 \text{ и } b \neq 0 \\ \frac{x-a}{x-c} & \text{при } x > 0 \text{ и } b = 0 \\ -\frac{x}{c} + \frac{-c}{2x} & \text{в остальных случаях} \end{cases}$$

где a, b, c – действительные числа.

Функция F должна принимать действительное значение, если выражение

НЕ(Ац ИЛИ Вц) И НЕ(Ац ИЛИ Сц)

не равно нулю, и целое значение в противном случае. Через Ац, Вц и Сц обозначены целые части значений a, b, c , операции НЕ, И и ИЛИ – поразрядные. Значения a, b, c , Хнач., Хкон., дХ ввести с клавиатуры.

Вычисление функции с помощью разложения в ряд

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

$$1. \ln \frac{x+1}{x-1} = 2 \sum_{n=0}^{\infty} \frac{1}{(2n+1)x^{2n+1}} = 2\left(\frac{1}{x} + \frac{1}{3x^3} + \frac{1}{5x^5} + \dots\right) |x| > 1 s$$

$$2. e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \dots |x| < \infty$$

$$3. e^{-x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^n}{n!} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \dots |x| < \infty$$

$$4. \ln(x+1) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{n+1}}{n+1} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} - \dots -1 < x \leq 1$$

$$5. \ln \frac{1+x}{1-x} = 2 \sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1} = 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \dots\right) |x| < 1$$

$$6. \ln(1-x) = -\sum_{n=1}^{\infty} \frac{x^n}{n} = -(x + \frac{x^2}{2} + \frac{x^4}{4} + \dots) -1 \leq x < 1$$

$$7. \operatorname{arcctg} x = \frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1} x^{2n+1}}{2n+1} = \frac{\pi}{2} - x + \frac{x^3}{3} - \frac{x^5}{5} - \dots |x| \leq 1$$

$$8. \operatorname{arctg} x = \frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{(2n+1)x^{2n+1}} = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} \dots x > 1$$

$$9. \operatorname{arctg} x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots |x| \leq 1$$

$$10. \operatorname{Arth} x = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1} = x + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + \dots |x| < 1$$

$$11. \operatorname{Arth} x = \sum_{n=0}^{\infty} \frac{1}{(2n+1)x^{2n+1}} = \frac{1}{x} + \frac{1}{3x^3} + \frac{1}{5x^5} + \dots |x| > 1$$

$$12. \operatorname{arctg} x = -\frac{\pi}{2} + \sum_{n=0}^{\infty} \frac{(-1)^{n+1}}{(2n+1)x^{2n+1}} = -\frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \dots x < -1$$

$$13. e^{-x^2} = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{n!} = 1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!} - \dots |x| < \infty$$

$$14. \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots |x| < \infty$$

$$15. \frac{\sin x}{x} = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} - \dots |x| < \infty$$

$$16. \ln x = 2 \sum_{n=0}^{\infty} \frac{(x-1)^{2n+1}}{(2n+1)(x+1)^{2n+1}} = 2\left(\frac{x-1}{x+1} + \frac{(x-1)^3}{3(x+1)^3} + \frac{(x-1)^5}{5(x+1)^5} + \dots\right) x > 0$$

$$17. \ln x = \sum_{n=0}^{\infty} \frac{(-1)^n (x-1)^{n+1}}{(n+1)} = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} + \dots \quad 0 < x \leq 2$$

$$18. \ln x = \sum_{n=0}^{\infty} \frac{(x-1)^{n+1}}{(n+1)(x+1)^{n+1}} = \frac{x-1}{x} + \frac{(x-1)^2}{2x^2} + \frac{(x-1)^3}{3x^3} + \dots \quad x > \frac{1}{2}$$

$$19. \arcsin x = x + \sum_{n=1}^{\infty} \frac{1 \cdot 3 \cdot \dots \cdot (2n-1) \cdot x^{2n+1}}{2 \cdot 4 \cdot \dots \cdot 2n \cdot (2n+1)} = x + \frac{x^3}{2 \cdot 3} + \frac{1 \cdot 3 \cdot x^5}{2 \cdot 4 \cdot 5} + \frac{1 \cdot 3 \cdot 5 \cdot x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} \dots |x| < 1$$

$$20. \arccos x = \frac{\pi}{2} - \left(x + \sum_{n=1}^{\infty} \frac{1 \cdot 3 \cdot \dots \cdot (2n-1) \cdot x^{2n+1}}{2 \cdot 4 \cdot \dots \cdot 2n \cdot (2n+1)} \right) = \\ = \frac{\pi}{2} - \left(x + \frac{x^3}{2 \cdot 3} + \frac{1 \cdot 3 \cdot x^5}{2 \cdot 4 \cdot 5} + \frac{1 \cdot 3 \cdot 5 \cdot x^7}{2 \cdot 4 \cdot 6 \cdot 8} + \frac{1 \cdot 3 \cdot 5 \cdot 7 \cdot x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} \dots \right) \quad |x| < 1$$

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. Чтобы лучше себе это представить, простые переменные можно уподобить гражданскому населению, а массивы – обитателям мест лишения свободы.

Массивы, как и любые другие объекты, можно размещать либо с помощью операторов описания в сегментах данных или стека, либо в динамической области памяти с помощью операций выделения памяти.

При описании массива после имени в квадратных скобках задается количество его элементов (размерность), например `int a[10]`. Массив располагается в зависимости от места его описания либо в сегменте данных, либо в сегменте стека, и все инструкции по выделению памяти формирует компилятор до выполнения программы. Вследствие этого размерность массива может быть задана только константой или константным выражением.

При описании массив можно инициализировать, то есть присвоить его элементам начальные значения, например:

```
int a[10] = {1, 1, 2, 2, 5, 100};
```

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

ВНИМАНИЕ

Элементы массивов нумеруются с нуля, поэтому максимальный номер элемента всегда на единицу меньше размерности. Автоматический контроль выхода индекса за границы массива не производится, поэтому программист должен следить за этим самостоятельно.

Для данного массива элементы имеют номера от 0 до 9. Номер элемента указывается после его имени в квадратных скобках, например, $a[0]$, $a[3]$.

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 минимальным элементами.

Запишем алгоритм в самом общем виде.

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

Перед написанием программы полезно составить тестовый пример, 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 и 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 максимальный и минимальный элементы:
 - 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 (т. е. 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 минимальным 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 экономии времени при отладке значения элементов массива задаются путем инициализации.

```
#include <iostream.h>
int main(){
    const int n = 10;
    int a[n] = {1, 3, -5, 1, -2, 1, -1, 3, 8, 4};
    int i, imax, imin, count;
    for (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 (count = 0, i = ibeg + 1; i < iend; i++)
        if (a[i] > 0) count++;
    cout << " Количество положительных: " << count << endl;
    return 0;
}
```

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 никогда не пренебрегать этим способом отладки и не жалеть времени, стараясь сделать эту печать хорошо читаемой, то есть содержащей необходимые пояснения и хорошо оформленной. Это экономит много времени при отладке программы.

Массив просматривается, начиная с элемента, следующего за максимальным (минимальным), до элемента, предшествующего минимальному (максимальному). Индексы границ просмотра хранятся в переменных ibeg и iend. В приведенной выше программе для определения их значений используется тернарная условная операция (см. с. 36). Можно поступить и по-другому: просматривать массив всегда от максимума к минимуму, а индекс при просмотре увеличивать или уменьшать в зависимости от их взаимного расположения.

В приведенной ниже программе направление просмотра, то есть приращение индекса, хранится в переменной d. Если массив просматривается «слева направо», она равна 1, иначе — -1. Обратите внимание и на изменившееся условие продолжения этого цикла.

```
#include <iostream.h>
int main(){
    const int n = 10;
    int a[n];
    int i, imax, imin, kol;
```

```

cout << " Введите " << n << " целых чисел: " << endl;
for (i = 0; i < n; i++) cin >> a[i];
for (i = 0; i < n; i++) cout << a[i] << " ";
for (i = imax = imin = 0; i < n; i++) {
    if (a[i] > a[imax]) imax = i;
    if (a[i] < a[imin]) imin = i;
}
int d = 0;
if (imax < imin) d = 1;
else if (imax > imin) d = -1;
for (kol = 0, i = imax + d; i != imin; i += d) if (a[i] > 0) kol++;
cout << "\n Количество положительных: " << kol << endl;
return 0;
}

```

Ввод массива в этом варианте программы осуществляется с клавиатуры. Напоминаем, что в этом случае желательно для проверки вывести введенные значения на печать.

Тестовых примеров для этой задачи должно быть по крайней мере три для случаев, когда:

- 1) a[imin] расположен левее a[imax];
- 2) a[imin] расположен правее a[imax];
- 3) a[imin] и a[imax] совпадают.

Последняя ситуация имеет место, когда в массиве все элементы имеют одно и то же значение. Кстати, во втором варианте программы третий случай корректно обрабатывается благодаря значению $d = 0$ для этой ситуации. Желательно также проверить, как работает программа, если a[imin] и a[imax] расположены рядом, а также в начале и в конце массива (границные случаи). Элементы массива нужно задавать как положительные, так и отрицательные.

Разница между приведенными способами решения несущественна, но первый вариант более «прозрачен», поэтому он, на наш взгляд, предпочтительнее.

Измените приведенную выше программу так, чтобы она вычисляла произведение отрицательных элементов, расположенных между минимальным и максимальным по модулю элементами массива.

Часто бывает, что точное количество элементов в исходном массиве не задано, но известно, что оно не может превышать некое конкретное значение. В этом случае память под массив выделяется «по максимуму», а затем заполняется только часть этой памяти. Память можно выделить либо с помощью оператора описания в стеке или сегменте данных, либо в динамической области. 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 мы приводим ниже небольшую, но полезную программу, в которой выполняется только считывание элементов массива с клавиатуры и их вывод на экран:

```
#include <iostream.h>
int main(){

```

```

const int n = 1000;
int a[n];
int i, kol_a;
cout << "Введите количество элементов: " << kol_a;
if (kol_a > n) {
    cout << " Превышение размеров массива " << endl; return 1;
}
for (i = 0; i < kol_a; i++) cin >> a[i];
for (i = 0; i < kol_a; i++) cout << a[i] << " ";
cout << endl;
return 0;
}

```

Несмотря на то, что значение константы 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.

Динамические массивы

Если до начала работы программы неизвестно, сколько в массиве элементов, в программе следует использовать динамические массивы. Память под них выделяется с помощью операции `new` или функции `malloc` в динамической области памяти во время выполнения программы. Адрес начала массива хранится в переменной, называемой указателем. Например:

```

int n = 10;
int *a = new int[n];
double *b = (double *)malloc(n * sizeof (double));

```

Во второй строке 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 третьей строке для выделения памяти под n элементов типа `double` используется функция `malloc`, унаследованная из библиотеки С. Этот способ устарел, поэтому мы им пользоваться не будем.

ВНИМАНИЕ

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

Обращение к элементу динамического массива осуществляется так же, как и к элементу обычного — например `a[3]`. Можно обратиться к элементу массива и другим способом — `*(a + 3)`. В этом случае мы явно задаем те же действия, что выполняются при обращении к элементу массива обычным образом. Рассмотрим их под-

робнее. 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 работы программы перестает быть нужным и мы собираемся впоследствии использовать эту память повторно, необходимо освободить ее с помощью операции delete[], например:

```
delete [] a;
```

Размерность массива при этом не указывается.

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 производится 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:

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

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.*

В этой задаче размерность массива задана переменной величиной. Предполагается, что она будет нам известна на этапе выполнения программы до того, как мы будем вводить сами элементы. В этом случае мы сможем выделить в динамической памяти непрерывный участок нужного размера, а потом заполнять его вводи-

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

мыми 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 достаточное количество памяти не удастся и придется воспользоваться так называемыми динамическими структурами данных, например списком. Мы рассмотрим эти структуры на девятом семинаре, а пока остановимся на первом предположении — что количество элементов массива вводится с клавиатуры до начала ввода самих элементов.

По аналогии с предыдущей задачей нам может прийти в голову такой алгоритм решения этой задачи: просматривая массив с начала до конца, найти номер последнего отрицательного элемента, а затем организовать цикл суммирования всех элементов, расположенных правее него. Вот как выглядит построенная по этому алгоритму программа (сразу же признаемся, что она далеко не так хороша, как может показаться с первого взгляда):

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

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 последовательно 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 массива, но мы не советуем так делать, потому что ввод данных, их вывод и анализ — разные по смыслу действия, и смешивание их в одном цикле не прибавит программе ясности. После отладки программы контрольный вывод (оператор 2) можно удалить или закомментировать. 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 самому интересному — к критическому анализу нашей первой попытки решения задачи. Для массивов, содержащих отрицательные элементы, эта программа работает верно, но при их отсутствии, как правило, завершается

аварийно. Это связано с тем, что если в массиве нет ни одного отрицательного элемента, переменной `ineg` значение не присваивается. Поэтому в операторе `for` (оператор 4) будет использовано значение `ineg`, ниспосланное Богом. Обычно всевышний таких ошибок не прощает, и программа незамедлительно «вылетает». Поэтому в программу необходимо внести проверку, есть ли в массиве хотя бы один отрицательный элемент. Для этого переменной `ineg` присваивается начальное значение, не входящее в множество допустимых индексов массива (например, `-1`). После цикла поиска номера отрицательного элемента выполняется проверка, сохранилось ли начальное значение `ineg` неизменным. Если это так, это означает, что условие `a[i] < 0` в операторе 3 не выполнилось ни разу, и отрицательных элементов в массиве нет:

```
#include <iostream.h>
int main(){
    int n;
    cout << " Введите количество элементов "; cin >> n;
    float sum, *a = new float [n]; // 1
    int i;
    cout << " Введите элементы массива: ";
    for (i = 0; i < n; i++) cin >> a[i];
    int ineg = -1;
    for (i = 0; i < n; i++) if (a[i] < 0) ineg = i; // 3
    if (ineg != -1){
        for (sum = 0, i = ineg + 1; i < n; i++) sum += a[i];
        cout << "\nСумма " << sum << endl;
    }
    else cout << "\nОтрицательных элементов нет" << endl;
    return 0;
}
```

Если не останавливаться на достигнутом и подумать¹, можно предложить и более рациональное решение этой задачи: просматривать массив в обратном порядке, суммируя его элементы, и завершить цикл, как только встретится отрицательный элемент:

```
#include <iostream.h>
int main(){
    int n;
    cout << " Введите количество элементов: "; cin >> n;
    float *a = new float [n];
    int i;
    cout << " Введите элементы массива: ";
    for (i = 0; i < n; i++) cin >> a[i];
    bool flag_neg = false;
    float sum = 0;
    for (i = n - 1; i >= 0; i--) {
        if (a[i] < 0) { flag_neg = true; break; }
        sum += a[i];
    }
}
```

¹ Думать иногда оказывается полезным.

```
if (flag_neg) cout << "\nСумма " << sum;
else cout << "\nОтрицательных элементов нет";
return 0;
}
```

В этой программе каждый элемент массива анализируется не более одного раза, а ненужные элементы не просматриваются вообще. Для больших массивов это играет роль, поэтому последний вариант программы предпочтительнее, хотя на вид он отличается от предыдущего незначительно¹.

Для исчерпывающего тестирования этой программы необходимо ввести по крайней мере три варианта исходных данных — когда массив содержит один, несколько или ни одного отрицательного элемента.

Измените приведенную выше программу так, чтобы она вычисляла сумму элементов, расположенных после самого левого отрицательного элемента.

Мы рассмотрели примеры задач поиска и вычисления 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, быстродействию, требуемому объему памяти, а также ограничениям, накладываемым на исходные данные. В Учебнике (см. 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 порядка N раз. Существуют алгоритмы и с лучшими характеристиками², самый известный из которых предложен Ч. Э. Р. Хоаром и называется «быстрая сортировка». Для него зависимость имеет вид:

$$t = a \cdot N \cdot \lg N + b \cdot N.$$

Давайте ради любопытства посмотрим, за счет чего достигается экономия.

Задача 3.3. Быстрая сортировка массива

Написать программу, которая упорядочивает вещественный массив методом быстрой сортировки.

Идея алгоритма состоит в следующем. Применим к массиву так называемую процедуру разделения относительно среднего элемента. Вообще-то, в качестве «среднего» можно выбрать любой элемент массива, но для наглядности мы будем выбирать действительно, по возможности, средний по своему номеру элемент.

Процедура разделения делит массив на две части. В левую помещаются элементы, меньшие, чем элемент, выбранный в качестве среднего, а в правой — большие. Это

¹ Впрочем, если в процессоре поддерживается опережающее считывание данных, этот вариант окажется медленнее.

² Кстати, наихудшим по характеристикам является любимый студентами метод пузырька. LMD.

достигается путем просмотра массива попеременно с обоих концов, при этом каждый элемент сравнивается с выбранным средним, и элементы, находящиеся в «не-подходящей» части, меняются местами. После завершения процедуры разделения средний элемент оказывается на своем окончательном месте, т. е. 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 автобус, в котором не работают все двери, кроме передней. Все сидячие места сломаны, а в проходе между ними помещается только по одному человеку в ряд. Человек, который имел несчастье первым войти в этот автобус, сможет покинуть его только самым последним. Вот это и есть стек — «первым пришел, последним ушел». В качестве упражнения придумайте более привлекательные примеры стеков.

ПРИМЕЧАНИЕ

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

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. Про данный способ реализации стека рассказывается в Учебнике на с. 126.

Ниже приведена программа, реализующая этот алгоритм.

```
#include <iostream.h>
#include <math.h>
int main(){
    const int n = 20;
    float arr[n], middle, temp;
    int *stackl = new int [n], *stackr = new int [n], sp = 0;
    int i, j, left, right;
    cout << "Введите элементы массива: ";
    for (i = 0; i < n; i++) cin >> arr[i];
    // Сортировка
    sp = 1; stackl[1] = 0; stackr[1] = n - 1; // 1
```

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

На каждом шаге сортируется один фрагмент массива. Левая граница фрагмента хранится в переменной `left`, правая — в переменной `right`. Сначала фрагмент устанавливается размером с массив целиком (строка 1). В операторе 8 выбирается «средний» элемент фрагмента.

Для продвижения по массиву слева направо в цикле 10 используется переменная `i`, справа налево — переменная `j` (в цикле 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 заносятся в стек границы правой части фрагмента. В операторе 13 устанавливаются новые границы левой части для сортировки на следующем шаге.

Если сортируемый 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.

Метод быстрой сортировки был предложен Ч. Хоаром. Впоследствии дотошный исследователь этого и других 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. В качестве дополнительного упражнения напишите улучшенную версию программы, в которой реализована эта идея и размер стека уменьшен до $\log_2 n$.

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.

Давайте повторим основные моменты этого семинара.

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.
5. Имя массива является указателем на его нулевой элемент.
6. Обнуления динамической памяти при ее выделении не происходит. Инициализировать динамический массив нельзя.
7. Освобождение памяти, выделенной посредством `new[]`, выполняется с помощью операции `delete[]`.
8. Перед выходом локального указателя из области его действия необходимо освобождать связанную с ним динамическую память.
9. 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.
10. Алгоритмы сортировки массивов различаются по быстродействию, занимаемой памяти и области применения.

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

Задания

Задания этого семинара соответствуют приведенным в Учебнике на с. 136. При написании программ можно использовать как динамические, так и нединамические массивы. Размерность последних задается именованной константой.

Вариант 1

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) сумму отрицательных элементов массива;
- 2) произведение элементов массива, расположенных между максимальным и минимальным элементами.

Упорядочить элементы массива по возрастанию.

Вариант 2

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) сумму положительных элементов массива;
- 2) произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами.

Упорядочить элементы массива по убыванию.

Вариант 3

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) произведение элементов массива с четными номерами;
- 2) сумму элементов массива, расположенных между первым и последним нулевыми элементами.

Преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом — все отрицательные (элементы, равные 0, считать положительными).

Вариант 4

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) сумму элементов массива с нечетными номерами;
- 2) сумму элементов массива, расположенных между первым и последним отрицательными элементами.

Сжать массив, удалив из него все элементы, модуль которых не превышает 1. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 5

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) максимальный элемент массива;

- 2) сумму элементов массива, расположенных до последнего положительного элемента.

Сжать массив, удалив из него все элементы, модуль которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 6

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) минимальный элемент массива;
- 2) сумму элементов массива, расположенных между первым и последним положительными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, равные нулю, а потом — все остальные.

Вариант 7

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) номер максимального элемента массива;
- 2) произведение элементов массива, расположенных между первым и вторым нулевыми элементами.

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

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) номер минимального элемента массива;
- 2) сумму элементов массива, расположенных между первым и вторым отрицательными элементами.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает 1, а потом — все остальные.

Вариант 9

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) максимальный по модулю элемент массива;
- 2) сумму элементов массива, расположенных между первым и вторым положительными элементами.

Преобразовать массив таким образом, чтобы элементы, равные нулю, располагались после всех остальных.

Вариант 10

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) минимальный по модулю элемент массива;

- 2) сумму модулей элементов массива, расположенных после первого элемента, равного нулю.

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

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) номер минимального по модулю элемента массива;
- 2) сумму модулей элементов массива, расположенных после первого отрицательного элемента.

Сжать массив, удалив из него все элементы, величина которых находится в интервале $[a, b]$. Освободившиеся в конце массива элементы заполнить нулями.

Вариант 12

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) номер максимального по модулю элемента массива;
- 2) сумму элементов массива, расположенных после первого положительного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых лежит в интервале $[a, b]$, а потом — все остальные.

Вариант 13

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

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

Вариант 14

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, равных 0;
 - 2) сумму элементов массива, расположенных после минимального элемента.
- Упорядочить элементы массива по возрастанию модулей элементов.

Вариант 15

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, больших C ;
- 2) произведение элементов массива, расположенных после максимального по модулю элемента.

Преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом — все положительные (элементы, равные 0, считать положительными).

Вариант 16

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество отрицательных элементов массива;
- 2) сумму модулей элементов массива, расположенных после минимального по модулю элемента.

Заменить все отрицательные элементы массива их квадратами и упорядочить элементы массива по возрастанию.

Вариант 17

В одномерном массиве, состоящем из n целых элементов, вычислить:

- 1) количество положительных элементов массива;
- 2) сумму элементов массива, расположенных после последнего элемента, равного нулю.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, целая часть которых не превышает 1, а потом — все остальные.

Вариант 18

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) количество элементов массива, меньших C ;
- 2) сумму целых частей элементов массива, расположенных после последнего отрицательного элемента.

Преобразовать массив таким образом, чтобы сначала располагались все элементы, отличающиеся от максимального не более чем на 20%, а потом — все остальные.

Вариант 19

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) произведение отрицательных элементов массива;
- 2) сумму положительных элементов массива, расположенных до максимального элемента.

Изменить порядок следования элементов в массиве на обратный.

Вариант 20

В одномерном массиве, состоящем из n вещественных элементов, вычислить:

- 1) произведение положительных элементов массива;
- 2) сумму элементов массива, расположенных до минимального элемента.

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. Для этого при описании в квадратных скобках указывается вторая размерность. Если массив определяется с помощью операторов описания, то обе его размерности должны быть константами или константными выражениями, поскольку инструкции по выделению памяти формируются компилятором до выполнения программы. Например:

`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
| --- 0-я строка -- | -- 1-я строка - | - 2-я строка - |`

Строки массива ничем не отделены одна от другой, то есть прямоугольной матрицей двумерный массив является только в нашем воображении. В памяти сначала располагается одномерный массив $a[0]$, представляющий собой нулевую строку массива a , затем — массив $a[1]$, представляющий собой первую строку 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 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` представляет собой константный указатель на начало массива. В данном случае 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)`¹. Напомним, что при сложении указателя с константой учитывается длина адресуемого элемента, поэтому на самом деле прибавляется число 2, умноженное 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 начальные значения его элементов. Их записывают в фигурных скобках. Элементы массива инициализируются в порядке их расположения в памяти. Например, оператор

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

определяет матрицу со следующими значениями элементов:

```
1 2 1 3 5  
2 3 4 5 1  
1 3 2 6 1
```

Если количество значений в фигурных скобках превышает количество элементов в массиве, при компиляции будет выдано сообщение об ошибке. 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:

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

Остальные элементы массива обнуляются.

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}};
```

¹ См. с. 59, где приведен аналогичный разбор для одномерных массивов.

Задача 4.1. Среднее арифметическое и количество положительных элементов

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

Алгоритм решения этой задачи очевиден. Для вычисления среднего арифметического элементов массива 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. Блок-схема этого алгоритма приведена на рис. 4.1.

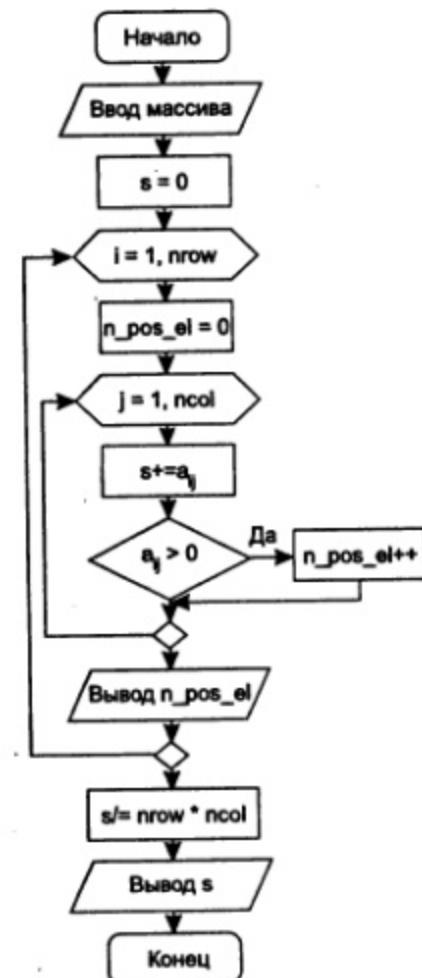


Рис. 4.1. Блок-схема алгоритма решения задачи 4.1

```
#include <iostream.h>
#include <iomanip.h>
void main(){
    const int nrow = 10, ncol = 20;
    int a[nrow][ncol];
    int i, j;
    cout << "Введите элементы массива:" << endl;
    for (i = 0; i < nrow; i++)
        for (j = 0; j < ncol; j++) cin >> a[i][j];

    for (i = 0; i < nrow; i++) {
        for (j = 0; j < ncol; j++) cout << setw(4) << a[i][j] << " ";
        cout << endl;
    }
    int n_pos_el;
    float s = 0;
    for (i = 0; i < nrow; i++) {
        n_pos_el = 0;
        for (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 4.2, с. 80, а также 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).

При вычислении количества положительных элементов для каждой строки выполняются однотипные действия: обнуление счетчика, просмотр каждого элемента строки и сравнение его с нулем, при необходимости увеличение счетчика на единицу, а после окончания вычислений — вывод результирующего значения счетчика. Впрочем, это лучше описывать на естественном языке, чем в программе.

Тем не менее, здесь следует обратить внимание на два момента. Во-первых, требуется еще до написания алгоритма решить, каким образом будут храниться резуль-

таты. Со средним арифметическим все ясно, для его хранения необходима одна простая переменная вещественного типа. А вот количество положительных элементов для каждой строки свое, и в результате мы должны получить столько значений, сколько строк в матрице. В данной задаче мы можем отвести для хранения этих значений одну-единственную переменную целого типа, поскольку они вычисляются последовательно, после чего выводятся на экран. Однако в других задачах эти значения могут впоследствии потребоваться одновременно. В этом случае для их хранения придется описать целочисленный массив с количеством элементов, равным количеству строк матрицы.

Второй важный момент — место инициализации суммы и количества. Сумма обнуляется перед циклом просмотра всей матрицы, а количество положительных элементов — перед циклом просмотра очередной строки, поскольку для каждой строки его вычисление начинается заново. Начинающие часто либо вообще забывают про инициализацию накапливаемых в цикле величин, либо 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 на экран. Для того чтобы элементы матрицы располагались один под другим, используется манипулятор `setw()`, устанавливающий для очередного выводимого значения ширину поля в четыре символа. Для использования манипулятора необходимо подключить к программе заголовочный файл `<iomanip.h>`. После каждой строки выводится символ перевода строки `endl`.

При написании вложенных циклов следите за отступами. 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 специалистов. В настоящее время наиболее распространенными являются два стиля: *стиль 1TBS* (One True Bracing Style), которого придерживались основоположники языка C — Б. Керниган (Brian Kernighan) и Д. Ритчи (Dennis Ritchie), и *стиль Алмана* (Eric Allman). Приведем отрывок кода, написанного в стиле 1TBS:

```
for (j = 0; j < MAX_LEN; j++) {
    foo();
}
```

Этот же отрывок в стиле Алмана выглядит так:

```
for (j = 0; j < MAX_LEN; j++)
{
    foo();
}
```

Наш выбор, естественно, остановился на *единственно верном стиле*.

Динамические массивы

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 сразу под весь массив количество строк (самую 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», нединамических массивов), например:

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

В этом фрагменте 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 `n` 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), но старые компиляторы могут ее не поддерживать:

```
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]`.

Для того чтобы понять, отчего динамические массивы описываются именно так, нужно разобраться в механизме индексации элемента массива, рассмотренном выше (см. с. 72). 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 под двумерный массив, когда обе его размерности задаются на этапе выполнения программы, приведен ниже:

```
int nrow, ncol;
cout << " Введите количество строк и столбцов : ";
cin >> nrow >> ncol;
int **a = new int *[nrow];
for(int i = 0; i < nrow; i++)
    a[i] = new int [ncol]; // 1
// 2
// 3
```

В операторе 1 объявляется переменная типа «указатель на указатель на `int`» и выделяется память под массив указателей на строки массива (количество строк — `nrow`). В операторе 2 организуется цикл для выделения памяти под каждую строку массива. В операторе 3 каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка состоит из `ncol` элементов типа `int` (рис. 4.2).

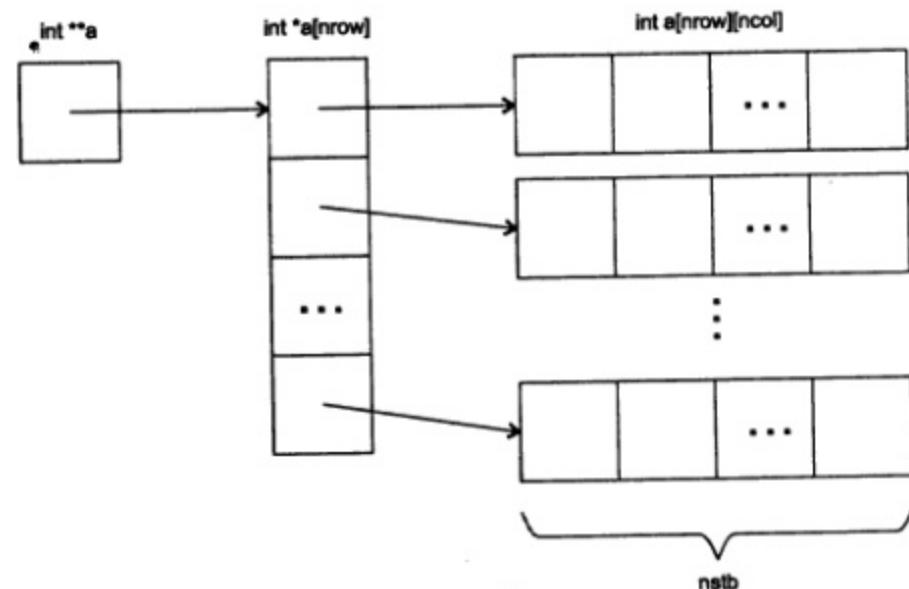


Рис. 4.2. Схема динамической области памяти, выделяемой под массивы

ВНИМАНИЕ

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

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.

Блок-схема алгоритма приведена на рис. 4.3. 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 перейти к следующему столбцу.

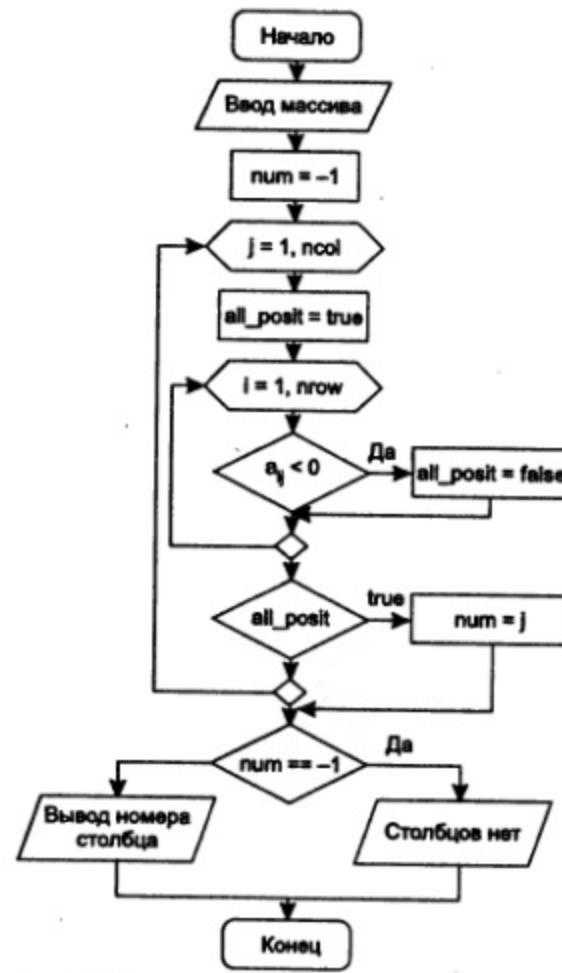


Рис. 4.3. Блок-схема алгоритма решения задачи 4.2

Эта логика реализуется с помощью переменной-флага `all_posit`, которая перед началом просмотра каждого столбца устанавливается в значение `true`, а при ма-
хождении отрицательного элемента «опрокидывается» в `false`. Если все элементы
столбца положительны, флаг не опрокинется и останется истинным, что будет яв-
ляться признаком присутствия в матрице искомого столбца.

Если столбец найден, просматривать матрицу дальше не имеет смысла, поэтому выполняется выход из цикла и вывод результата.

```

#include <iostream.h>
#include <iomanip.h>
int main()
{
    int nrow, ncol;
    cout << "Введите количество строк и столбцов: ";
    cin >> nrow >> ncol; // ввод размерности массива
    ...
}
    
```

```

int i, j;
int **a = new int *[nrow]; // выделение памяти под массив
for(i = 0; i < nrow; i++) a[i] = new int [ncol];
cout << "Введите элементы массива:" << endl;
for (i = 0; i < nrow; i++)
    for (j = 0; j < ncol; j++) cin >> a[i][j]; // ввод массива
for (i = 0; i < nrow; i++)
    for (j = 0; j < ncol; j++) cout << setw(4) << a[i][j] << " ";
cout << endl;

int num = -1;
bool all_posit;
for (j = 0; j < ncol; j++) { // просмотр по столбцам
    all_posit = true;
    for (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;
return 0;
}
    
```

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 оно после просмотра матрицы сохранилось неизменным, то есть осталось равным `-1`, то столбцов, удовлетворяющих заданному условию, в матрице нет.

Можно обойтись без анализа переменной `num`, да и вообще без этой переменной, если вывести номер столбца сразу после его определения, после чего завершить программу. Этот вариант приведен ниже. Заодно продемонстрированы простейшие способы работы с файлами.

СОВЕТ

Текстовые файлы очень удобно использовать для отладки программ, требующих ввода хотя бы нескольких величин, — ведь, как правило, программу не удается написать сразу без ошибок, а многократный ввод одних и тех же значений замедляет процесс отладки и может сильно испортить настроение. Кроме того, при подготовке данных в файле до выполнения программы можно спокойно продумать тестовые примеры для исчерпывающей проверки правильности программы.

¹ Распространенной ошибкой при проверке на равенство является использование вместо него операции присваивания (`=`). Будьте внимательны! Есть один прием, повышающий надежность кодирования таких выражений: на первом месте нужно записать константу, а после знака `<=>` поместить имя переменной; в этом случае потеря одного символа `<=>` обнаружит компилятор.

```
#include <fstream.h>
#include <iomanip.h>
int main(){
    ifstream fin("input.txt", ios::in | ios::nocreate);
    if (!fin) {
        cout << " Файл input.txt не найден." << endl; return 1;
    }
    ofstream fout("output.txt");
    if (!fout) {
        cout << " Невозможно открыть файл для записи." << endl; return 1;
    }

    int nrow, ncol;
    fin >> nrow >> ncol;
    int i, j;
    int **a = new int *[nrow];
    for(i = 0; i < nrow; i++) a[i] = new int [ncol];
    for (i = 0; i < nrow; i++)
        for (j = 0; j < ncol; j++) fin >> a[i][j];
    for (i = 0; i < nrow; i++) {
        for (j = 0; j < ncol; j++) fout << setw(4) << a[i][j] << " ";
        fout << endl;
    }

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

Ввод размерности массива и его элементов выполняется из файла `input.txt`, расположенного в том же каталоге, что и программа, а результаты выводятся в файл `output.txt`. В программе определены объект `fin` класса входных файловых потоков и объект `fout` класса выходных файловых потоков. Файловые потоки описаны в заголовочном файле `<fstream.h>`. Работа с этими объектами аналогична работе со стандартными объектами `cin` и `cout`, то есть можно пользоваться теми же операциями помещения в поток `<<` и извлечения из потока `>>`.

Предполагается, что файл с именем `input.txt` находится в том же каталоге, что и текст программы, иначе следует указать полный путь, дублируя символ об-

ратной косой черты, так как иначе он будет иметь специальное значение, например:

```
ifstream fin("c:\\prim\\cpp\\input.txt", ios::in | ios::nocreate);
```

После определения объектов проверяется успешность их создания. Это особенно важно делать для входных файлов, чтобы исключить вероятность ошибки в имени или местоположении файла. На следующем семинаре мы рассмотрим файловые потоки более подробно.

Если программа завершается успешно, то на экран выводится сообщение «Работа завершена». Благодаря этому пользователь вашей программы поймет, что она все же что-то сделала. Еще более гуманным было бы вывести дополнительное сообщение типа «Результаты смотрите в файле output.txt».

Входной файл `input.txt` можно создать в любом текстовом редакторе. Он, естественно, должен существовать до первого запуска программы. На расположение и формат исходных данных в файле никаких ограничений не накладывается.

Задача 4.3. Упорядочивание строк матрицы

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

Давайте на этом примере формализуем общий порядок создания структурной программы¹, которому мы ранее следовали интуитивно. Этого порядка полезно придерживаться при решении даже простейших задач. Более строго и детально он изложен в Учебнике на с. 109–113.

I. Исходные данные, результаты и промежуточные величины. Как уже неоднократно упоминалось, начинать решение задачи необходимо с четкого описания того, что является ее исходными данными и результатами и каким образом они будут представлены в программе.

Исходные данные. Поскольку размерность матрицы неизвестна, придется использовать динамический массив элементов целого типа. Ограничимся типом `int`, хотя для общности следовало бы воспользоваться максимально длинным целым типом. Но не будем параноиками.

Результаты. Результатом является та же матрица, но упорядоченная. Это значит, что нам не следует заводить для результата новую область памяти, а необходимо упорядочить матрицу *in situ*, то есть на том же месте. В данной задаче такое требование может показаться излишним, но в общем случае, когда программист работает в команде и должен передавать результаты коллеге, это важно. Представьте себе ситуацию, когда коллега думает, что получил от вас упорядоченную матрицу, а на самом деле вы сформировали ее в совершенно другой области памяти.

¹ Создание объектно-ориентированной программы требует несколько иных действий и будет рассмотрено во второй части практикума.

Промежуточные величины. Кроме конечных результатов, в любой программе есть промежуточные, а также служебные 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 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 выбора¹. Он состоит в том, что из массива выбирается наименьший элемент и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом 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). Пренебрегать словесным описанием не следует, потому что процесс формулирования на естественном языке полезен для более четкого понимания задачи. При этом надо стремиться разбить алгоритм на простую последовательность шагов. Например, любой алгоритм можно первоначально разбить на этапы ввода исходных данных, вычислений и вывода результата.

Вычисление в данном случае состоит из двух шагов: формирование сумм элементов каждой строки и упорядочивание матрицы. Упорядочивание состоит в выборе наименьшего элемента и обмене с первым из рассматриваемых. Разветвленные алгоритмы и алгоритмы с циклами полезно представить в виде обобщенной блок-схемы.

III. Когда алгоритм полностью прояснился, можно переходить к написанию *программы*. Одновременно с этим продумываются и подготавливаются тестовые примеры. Не ленитесь придумать переменным понятные имена и сразу же при написании аккуратно форматировать текст программы, чтобы по положению оператора было видно, на каком уровне вложенности он находится. Функционально завершенные части алгоритма отделяются пустой строкой, комментарием или хотя бы комментарием вида

// -----

СОВЕТ

Мы рекомендуем при написании программы всегда 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. Отладка программы с использованием средств оболочек описана в приложениях 1 и 2.

Ниже приведен текст программы сортировки.

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

int main(){
    ifstream fin("input.txt", ios::in | ios::nocreate);
    if (!fin) {
        cout << "Файл input.txt не найден" << endl; return 1; }

    int nrow, ncol;
    fin >> nrow >> ncol; // ввод размерности массива
    int i, j;
    int **a = new int *[nrow]; // выделение памяти под массив
    for(i = 0; i < nrow; i++) a[i] = new int [ncol];
    for (i = 0; i < nrow; i++) // ввод массива
        for (j = 0; j < ncol; j++) fin >> a[i][j];

    long *sum = new long [nrow]; // массив сумм элементов строк
    for (i = 0; i < nrow; i++) {
        sum[i] = 0;
        for (j = 0; j < ncol; j++) sum[i] += a[i][j];
    }

    for (i = 0; i < nrow; i++) { // контрольный вывод
        for (j = 0; j < ncol; j++) cout << setw(4) << a[i][j] << " ";
        cout << "| " << sum[i] << endl;
    }
    cout << endl;

    long buf_sum;
    int nmin, buf_a;
    for (i = 0; i < nrow - 1; i++) { // упорядочивание
        nmin = i;
        for (j = i + 1; j < nrow; j++)
            if (sum[j] < sum[nmin]) nmin = j;
        if (nmin != i) {
            buf_a = a[i][0];
            for (j = 0; j < ncol; j++)
                a[i][j] = a[nmin][j];
            a[nmin][0] = buf_a;
            buf_sum = sum[i];
            sum[i] = sum[nmin];
            sum[nmin] = buf_sum;
        }
    }
}
```

¹ Метод выбора рассмотрен в Учебнике на с. 59.

```

if (sum[j] < sum[nmin]) nmin = j;
buf_sum = sum[i]; sum[i] = sum[nmin]; sum[nmin] = buf_sum;
for (j = 0; j < ncol; j++) {
    buf_a = a[i][j]; a[i][j] = a[nmin][j]; a[nmin][j] = buf_a;
}
}

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

```

В программе используются две буферные переменные: `buf_sum`, через которую осуществляется обмен двух значений сумм, имеет такой же тип, что и сумма, а для обмена значений элементов массива определена переменная `buf_a` того же типа, что и элементы массива.

Как и в предыдущем примере, данные читаются из файла. Мы рекомендуем пользоваться именно этим способом, а не стандартным вводом, поскольку при формировании файла легче продумать, какие значения лучше взять для исчерпывающего тестирования программы. В данном случае для первого теста следует подготовить массив не менее чем из четырех строк с небольшими значениями элементов для того, чтобы можно было в уме проверить, правильно ли вычисляются суммы.

Обратите внимание, что для контроля вместе с исходным массивом рядом с каждой строкой выводится сумма ее элементов, отделенная вертикальной чертой.

В качестве второго тестового примера рекомендуется ввести значения элементов массива, 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.

Еще один пример работы с двумерными массивами вы можете найти в задаче 7.4.

Давайте повторим основные моменты этого семинара.

1. В массивах, определенных с помощью операторов описания, обе размерности должны быть константами или константными выражениями.
2. Массив хранится по строкам в непрерывной области памяти.
3. Первый индекс всегда представляет собой номер строки, второй — номер столбца. Каждый индекс может изменяться от 0 до значения соответствующей размерности, уменьшенной на единицу.
4. При описании массива можно в фигурных скобках задать начальные значения его элементов.
5. При выделении динамической памяти сразу под весь массив самую левую размерность можно задавать с помощью переменной, все остальные размерности должны быть константами. Для двумерного массива это означает, что первая

размерность может быть константой или переменной, а вторая — только константой.

6. Для выделения динамической памяти под массив, в котором все размерности переменные, используются циклы.

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

Ниже в сжатом виде приведены *рекомендации по порядку создания программы*.

1. Выбрать тип и способ хранения в программе исходных данных, результатов и промежуточных величин.
2. Записать алгоритм сначала в общем виде, стремясь разбить его на простую последовательность шагов, а затем детализировать каждый шаг.
3. Написать программу. При написании программы рекомендуется:

- давать переменным понятные имена;
- не пренебрегать содержательными комментариями;
- использовать промежуточную печать вычисляемых величин в удобном формате;
- при написании вложенных циклов следить за отступами;
- операторы инициализации накапливаемых в цикле величин задавать непосредственно перед циклом, в котором они вычисляются.

4. Параллельно с написанием программы задать тестовые примеры, которые проверяют все ветви алгоритма и возможные диапазоны значений исходных данных. Исходные данные удобнее формировать в файле (по крайней мере, при отладке), не забывая проверять в программе успешность его открытия.

Более подробно технология создания программы описана в Учебнике на с. 102–114.

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, расположить 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 на 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, расположить 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 на 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 на 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 p элементов вправо или вниз (в зависимости от введенного режима). p 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, расположить 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 матрицы.

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: С-строки и класс стандартной библиотеки C++ `string`. С-строка представляет собой массив символов, завершающийся символом с кодом 0¹. Класс `string` более безопасен в использовании, чем С-строки, но и более ресурсоемок. Для грамотного использования этого класса требуется знание объектно-ориентированного программирования, поэтому мы рассмотрим его во второй части практикума, а на этом семинаре ограничимся рассмотрением С-строк.

Описание строк

Память под строки, как и под другие массивы, может выделяться как компилятором, так и непосредственно в программе. Длина динамической строки может задаваться выражением, длина не-динамической строки должна быть только константным выражением. Чаще всего длина строки задается частным случаем константного выражения — константой. Удобно задавать длину с помощью именованной константы, поскольку такой вариант, во-первых, лучше читается, а во-вторых, при возможном изменении длины строки потребуется изменить программу только в одном месте:

```
const int len_str = 80;
char str[len_str];
```

При задании длины необходимо учитывать завершающий нуль-символ. Например, в строке, приведенной выше, можно хранить не 80 символов, а только 79. Строки можно при описании инициализировать строковыми константами, при этом нуль-символ в позиции, следующей за последним заданным символом, формируется автоматически:

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

¹ Этот вид строк, как вы догадались, пришел в C++ из языка С.

Если строка при определении 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 (первый способ предпочтительнее):

```
char *p = new char [m];
char *q = (char *)malloc( m * sizeof(char));
```

Естественно, что 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. Оператор

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

создает не строковую переменную, а указатель на строковую константу, изменить которую невозможно.

Ввод-вывод строк

Для ввода-вывода строк используются как уже известные нам объекты cin и cout, так и функции, унаследованные из библиотеки C. Рассмотрим сначала первый способ:

```
#include <iostream.h>
int main(){
    const int n = 80;
    char s[n];
    cin >> s; cout << s << endl;
    return 0;
}
```

Как видите, строка вводится точно так же, как и переменные известных нам типов. Запустите программу и введите строку, состоящую из одного слова. Запустите программу повторно и введите строку из нескольких слов. Во втором случае выводится только первое слово. Это связано с тем, что ввод выполняется до первого пробельного символа (то есть пробела, знака табуляции или символа перевода строки '\n')¹.

Можно ввести слова входной строки в отдельные строковые переменные:

```
#include <iostream.h>
int main(){
    const int n = 80;
    char s[n], t[n], r[n];
    cin >> s >> t >> r; cout << s << endl << t << endl << r << endl;
    return 0;
}
```

¹ Если во вводимой строке больше символов, чем может вместить выделенная для ее хранения область, поведение программы не определено. Скорее всего, она завершится аварийно.

Если требуется ввести строку, состоящую из нескольких слов, в одну строковую переменную, используются методы getline или get класса istream, объектом которого является cin. Во второй части практикума мы изучим, что такое методы класса¹, а пока можно пользоваться ими как волшебным заклинанием, не вдумываясь в смысл. Единственное, что нам пока нужно знать, это синтаксис вызова метода — после имени объекта ставится точка, а затем пишется имя метода:

```
#include <iostream.h>
int main(){
    const int n = 80;
    char s[n];
    cin.getline(s, n); cout << s << endl;
    cin.get(s, n); cout << s << endl;
    return 0;
}
```

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 строки² также считывается (удаляется) из входного потока, но не записывается в строковую переменную, вместо него размещается завершающий 0. Если в строке 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. Например:

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 в данном случае «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 (строки, содержащие 0 символов). 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 после операторов 1 и 3 нужно вставить вызов cin.get().

¹ Синонимом термина «метод» является «функция-член класса».

² Символ перевода строки '\n' появляется во входном потоке, когда вы нажимаете клавишу Enter.

Однако есть и более простое решение – использовать в таких случаях метод `getline`, который после прочтения строки не оставляет во входном потоке символ `\n`.

Если в программе требуется ввести несколько строк, метод `getline` удобно использовать в заголовке цикла, например:

```
#include <iostream.h>
int main(){
    const int n = 80;
    char s[n];
    while (cin.getline(s, n)) {
        cout << s << endl;
        // обработка строки
    }
    return 0;
}
```

Рассмотрим теперь способы ввода-вывода строк, перекочевавшие 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 <stdio.h>
int main(){
    const int n = 10;
    char s[n];
    scanf("%s", s); printf("%s", s);
    return 0;
}
```

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 `%c` (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 и далее.

Библиотека содержит также функции, специально предназначенные для ввода-вывода строк: `gets` и `puts`. Предыдущий пример с использованием этих функций выглядит так:

```
#include <stdio.h>
int main(){
```

```
const int n = 10;
char s[n];
gets(s); puts(s);
return 0;
}
```

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 строка является не основным типом данных, а массивом. Присваивание выполняется с помощью функций стандартной библиотеки или посимвольно «вручную» (что менее предпочтительно, так как чревато ошибками). Например, чтобы присвоить строке `r` строку `a`, можно воспользоваться функциями `strcpy` или `strncpy`:

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

Для использования этих функций к программе следует подключить заголовочный файл `<string.h>`.

Функция `strcpy(p, a)` копирует все символы строки, указанной вторым параметром (`a`), включая завершающий 0, в строку, указанную первым параметром (`p`).

Функция `strncpy(p, a, n)` выполняет то же самое, но не более `n` символов, то есть числа символов, указанного третьим параметром. 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` меньше или равно длине строки `a`) завершающий нуль-символ в `p` не добавляется.

Обе эти функции возвращают указатель на результирующую строку. Если области памяти, занимаемые строкой-назначением и строкой-источником, перекрываются, поведение программы не определено.

Функция `strlen(a)` возвращает фактическую длину строки `a`, не включая нуль-символ.

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.

Для преобразования строки в целое число используется функция `atoi(str)`. Функция 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 вещественное число с двойной точностью (`double`) называются `atol` и `atof` соответственно.

Пример применения функций преобразования:

```
char a[] = "10 Рост - 162 см. вес - 59.5 кг";
int num;
long height;
double weight;
num = atoi(a);
height = atol(&a[11]);
weight = atof(&a[25]);
cout << num << ' ' << height << ' ' << weight;
```

Библиотека предоставляет также различные 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 осуществляется как с помощью операций помещения в поток `<<` и извлечения из потока `>>`, так и методов `get()` и `get(char)`. Ниже приведен пример применения операций:

```
#include <iostream.h>
int main(){
    char c, d, e;
    cin >> c;
    cin >> d >> e;
```

Задача 5.1. Поиск подстроки

```
cout << c << d << e << endl;
return 0;
}
```

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

```
#include <iostream.h>
int main(){
    char c, d, e;
    c = cin.get(); cin.get(d); cin.get(e);
    cout << c << d << e << endl;
    return 0;
}
```

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 `<stdio.h>` определена функция `getchar()` для ввода символа со стандартного ввода, а также `putchar()` для вывода:

```
#include <stdio.h>
int main(){
    char c, d;
    c = getchar(); putchar(c);
    d = getchar(); putchar(d);
    return 0;
}
```

В библиотеке также 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.

Перейдем теперь к рассмотрению задач.

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 и результаты

Исходные данные:

1. Текстовый файл неизвестного размера, состоящий из строк длиной не более 80 символов. Поскольку по условию переносы отсутствуют, можно огра-

ничиться 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 об 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

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

III. Программа и тестовые примеры

```
#include <fstream.h>
#include <string.h>
int main(){
    const int len = 81;                                // 1
    char word[len], line[len];                         // 2
    cout << "Введите слово для поиска: "; cin >> word;

    ifstream fin("text.txt", ios::in | ios::nocreate); // 3
    if (!fin) { cout << "Ошибка открытия файла." << endl;
        return 1; }                                     // 4

    while (fin.getline(line, len)) {                    // 5
        if (strstr(line, word)) {                      // 6
            cout << "Присутствует!" << endl; return 0; }

    }
    cout << "Отсутствует!" << endl;                  // 7
    return 0;
}
```

Рассмотрим помеченные операторы. В операторе 1 описывается константа, определяющая длину строки файла и длину последовательности. В операторе 2 опи-

сывается переменная line для размещения очередной строки файла и переменная word для размещения 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, например:

```
ifstream fin("c:\\prim\\cpp\\text.txt", ios::in | ios::nocreate); // 3
```

В операторе 4 проверяется успешность создания объекта fin. Файлы, открываемые для чтения, проверять нужно обязательно! В операторе 5 организуется цикл чтения из файла в переменную line. Метод getline, описанный выше, при достижении конца файла вернет значение, завершающее цикл.

Для анализа строки в операторе 6 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. Если вторым параметром передается указатель на строку нулевой длины, функция возвращает указатель на начало строки line.

В качестве тестового примера приготовьте текстовый файл, состоящий из нескольких строк¹. 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.

Даже такую простую программу мы рекомендуем вводить и отлаживать по шагам. Это умение пригодится вам в дальнейшем. Предлагаемая последовательность отладки:

- Ввести «скелет» программы (директивы #include, функцию main(), операторы 1–4). Добавить контрольный вывод введенного слова. Запустив программу, проверить ввод слова и успешность открытия файла. Выполнить программу, задав имя несуществующего файла, для проверки вывода сообщения об ошибке. Удалить контрольный вывод слова.
- Проверить цикл чтения из файла: добавить оператор 5 с его завершающей фигурной скобкой, внутри цикла поставить контрольный вывод прочитанной строки:

```
cout << line << endl;
```

Удалить контрольный вывод строки.
- Дополнить программу операторами проверки и вывода сообщений. Для полной проверки программы следует выполнить ее для нескольких последователь-

¹ Файл можно создать в любом текстовом редакторе, в том числе и в той оболочке, в которой вы работаете. Для правильного отображения русских букв при выводе на консоль вид кодировки должен быть ASCII.

ностей. Длина одной из них должна составлять максимально допустимую — 80 символов.

СОВЕТ

При вводе текста программы не ленитесь сразу же форматировать его и снабжать комментариями.

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 программа не сильно отличается от предыдущей: вместо факта наличия искомой последовательности в файле требуется подсчитывать 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 в данной задаче нам 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, можно ограничиться поиском слова в каждой строке отдельно. Для ее хранения выделим строку длиной 81 символ.
2. Слово для поиска, вводимое с клавиатуры. Для его хранения также выделим строку длиной 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 потребуется 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, является ли оно отдельным словом, а не частью другого¹. Например, мы задали слово «кот». Эта последовательность символов содержится, например, в словах «котенок», «трикотаж», «трекотня» и «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 на последовательность шагов.

Шаг 1. Ввести «скелет» программы (директивы #include, функцию main(), описание переменных, открытие файла). Добавить контрольный вывод введенного слова. Запустив программу, проверить ввод слова и успешность открытия файла. Для проверки вывода сообщения об ошибке следует выполнить программу еще раз, задав имя несуществующего файла.

```
#include <fstream.h>
int main(){
    const int len = 81;
    char word[len], line[len];
    cout << "Введите слово для поиска: "; cin >> word;
    ifstream fin("text.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }
    return 0;
}
```

Шаг 2. Добавить в программу цикл чтения из файла, внутри цикла поставить контрольный вывод считанной строки (добавляемые операторы помечены признаком комментария):

```
#include <fstream.h>
int main(){
    const int len = 81;
    char word[len], line[len];
    cout << "Введите слово для поиска: "; cin >> word;
    ifstream fin("text.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }
```

¹ Кроме этого, слово может быть написано в разных регистрах, но мы для простоты будем искать точное совпадение.

```

while (fin.getline(line, len)) {
    cout << line << endl;
}
return 0;
}

```

Шаг 3. Добавить в программу цикл поиска последовательности символов, составляющих слово, с контрольным выводом:

```

#include <fstream.h>
#include <string.h>
int main(){
    const int len = 81;
    char word[len], line[len];
    cout << "Введите слово для поиска: "; cin >> word;
    int l_word = strlen(word);
    ifstream fin("text.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }

    int count = 0;
    while (fin.getline(line, len)) {
        char *p = line;
        while( p = strstr(p, word)) {
            cout << " совпадение: " << p << endl;
            p += l_word; count++;
        }
    }
    cout << count << endl;
    return 0;
}

```

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:

```

#include <fstream.h>
#include <string.h>
#include <ctype.h>
int main(){
    const int len = 81;
    char word[len], line[len];

```

```

cout << " Введите слово для поиска: "; cin >> word;
int l_word = strlen(word);

ifstream fin("text.txt", ios::in | ios::nocreate);
if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }

int count = 0;
while (fin.getline(line, len)) {
    char *p = line;
    while( p = strstr(p, word)) {
        char *c = p;
        p += l_word;
        // слово не в начале строки?
        if (c != line)
            // символ перед словом не разделитель?
            if ( !ispunct(*(c - 1)) && !isspace(*(c - 1))) continue;
        // символ после слова разделитель?
        if (ispunct(*p) || isspace(*p) || (*p == '\0')) count++;
    }
}
cout << "Количество вхождений слова: " << count << endl;
return 0;
}

```

Здесь вводится служебная переменная `c` для хранения адреса начала вхождения подстроки. Символы, ограничивающие слово, проверяются с помощью функций `ispunct` и `isspace`, прототипы которых хранятся в заголовочном файле `<ctype.h>`. 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. Для тестирования программы следует выполнить ее по крайней мере два раза: введя с клавиатуры слово, содержащееся в файле, и слово, которого в нем нет.

Давайте теперь рассмотрим другой вариант решения этой задачи. В библиотеке есть функция `strtok`, которая разбивает переданную ей строку на лексемы в соответствии с заданным набором разделителей. Если мы воспользуемся этой функцией, нам не придется «вручную» выделять и проверять начало и конец слова,

потребуется лишь сравнить с искомым словом слово, выделенное с помощью `strtok`. Правда, список разделителей придется задать вручную:

```
#include <fstream.h>
#include <string.h>
int main(){
    const int len = 81;
    char word[len], line[len];
    char delims[] = "...!/?\"(*)\":";
    cout << "Введите слово для поиска: "; cin >> word;

    ifstream fin("text.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }

    char *token;
    int count = 0;
    while (fin.getline(line, len)) {
        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;
    return 0;
}
```

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`.

Как видите, программа стала короче и яснее. На этом примере можно видеть, что средства, предоставляемые языком, влияют на алгоритм решения задачи, и поэтому перед тем, как продумывать алгоритм, необходимо эти средства изучить. Представьте, во что бы вылилась программа 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 дополнительного пространства под них выделять не требуется.

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.
2. Определить его длину в байтах.
3. Выделить в динамической памяти буфер соответствующего размера.
4. Считать файл с диска в буфер.
5. 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. В случае обнаружения вопросительного знака предложение выводится на экран, после чего также устанавливается новое значение позиции начала предложения.

III. Программа и тестовые примеры

Ниже приводится текст программы. 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.

```
#include <fstream.h>
#include <stdio.h>
int main(){
    ifstream fin("text.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла." << endl; return 1; }

    fin.seekg(0, ios::end); // 1
    long len = fin.tellg(); // 2
    char *buf = new char [len + 1]; // 3
    fin.seekg(0, ios::beg); // 4
    fin.read(buf, len); // 5
    buf[len] = '\0'; // 6
    long n = 0, i = 0, j = 0; // 7
    while(buf[i]) { // 8
        if( buf[i] == '?' ) {
```

```

for ( j = n; j <= i; j++) cout << buf[j];
n = i + 1;
}
if ( buf[i] == '.' || buf[i] == '!' ) n = j + 1;
i++;
}
fin.close(); // 9
cout << endl;
return 0;
}

```

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 перемещаем 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, ios::beg)` перемещает текущую позицию чтения из файла на `offset` байтов относительно `ios::beg`. Параметр `ios::beg` может принимать одно из трех значений:

`ios::beg` — от начала файла;
`ios::cur` — от текущей позиции;
`ios::end` — от конца файла.

`beg`, `cur` и `end` являются константами, определенными 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, переменная `j` — для вывода предложения.

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` до текущей, после чего в переменную `n` заносится позиция начала нового предложения.

Оператор 9 (закрытие потока) в данном случае не является обязательным, так как явный вызов `close()` необходим только тогда, 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];

```

Если требуется закрыть поток раньше окончания действия его области видимости, используется метод `close()`:

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

```

while((buf[1] = fin.get()) != EOF) {
    ...
    i++;
}

```

Надо учитывать, что посимвольное чтение из файла гораздо менее эффективно. В заключение приведем вариант решения этой же задачи с использованием вместо потоковых классов библиотечных функций, унаследованных из языка С.

```

#include <stdio.h>
int main(){
    FILE *fin; // 1
    fin = fopen("text.txt", "r"); // 2
    if (!fin) { puts("Ошибка открытия файла"); return 1; }

    fseek(fin, 0, SEEK_END); // 3
    long len = ftell(fin); // 4
    char *buf = new char [len + 1];

    const int l_block = 1024; // 5
    int num_block = len / l_block; // 6
    fseek(fin, 0, SEEK_SET); // 7
    fread(buf, l_block, num_block + 1, fin); // 8
    buf[len] = '\0';

    long n = 0, i = 0, j = 0;
    while(buf[i]) {
        if( buf[i] == '?' ) {
            for ( j = n; j <= i; j++) putchar(buf[j]);
            n = i + 1;
        }
        if ( buf[i] == '.' || buf[i] == '!' ) n = i + 1;
        i++;
    }
}

```

```

fclose(fin);
printf("\n");
return 0;
}

```

В операторе 1 определяется указатель на описанную в заголовочном файле <stdio.h> структуру FILE. Указатель именно такого типа формирует функция открытия файла fopen. Ее вторым параметром задается режим открытия файла. В данном случае файл открывается для чтения (r).

Файл можно открыть в двоичном (b) или текстовом (t) режиме. Эти символы записывают во втором параметре, например, "rb" или "rt". Двоичный режим означает, что символы перевода строки и возврата каретки (0x13 и 0x10) обрабатываются точно так же, как и остальные. В текстовом режиме эти символы преобразуются в одиночный символ перевода строки. По умолчанию файлы открываются в текстовом режиме.

Для позиционирования указателя текущей позиции используется функция fseek с параметрами, аналогичными соответствующему методу потока (операторы 3 и 7). Константы, задающие точку отсчета смещения, описаны в заголовочном файле <stdio.h> и имеют имена:

SEEK_SET — от начала файла;
SEEK_CUR — от текущей позиции;
SEEK_END — от конца файла.

Чтение из файла выполняется функцией fread(buf, size, num, file) блоками по size байт. Требуется также задать количество блоков num. В программе размер блока задан в переменной l_block равным 1024, поскольку размер кластера кратен степени двойки. В общем случае чем более длинными блоками мы читаем информацию, тем быстрее будет выполнен ввод. Для того чтобы обеспечить чтение всего файла, к количеству блоков добавляется 1 для округления после деления.

Вывод на экран выполняется посимвольно с помощью функции putchar.

Если требуется с помощью функций библиотеки вывести результаты выполнения программы не на экран, а в файл, в программе следует описать указатель на структуру FILE, с помощью функции fopen открыть файл для записи (второй параметр функции — w), а затем использовать этот указатель в соответствующих функциях вывода, например:

```

FILE *fout;
fout = fopen("textout.txt", "w");
if (!fout) { puts("Ошибка открытия файла вывода"); return 1; }

putc(buf[j], fout); // или fputc(buf[j], fout);

```

После окончания вывода файл закрывается с помощью функции fclose:

```
fclose(fout);
```

Функции вывода в файл описаны в Учебнике на с. 90 и 411.

Смешивать в одной программе ввод-вывод с помощью потоковых классов и с помощью функций библиотеки не рекомендуется.

В целом программа, написанная с использованием функций библиотеки, может получиться более быстродействующей, но менее безопасной, поскольку программист должен сам заботиться о большем количестве деталей.

Давайте повторим основные моменты этого семинара.

1. Длина динамической строки может быть переменной. Динамические строки нельзя инициализировать при создании.
2. Длина нединамической строки должна быть константным выражением.
3. При задании длины строки необходимо учитывать завершающий нуль-символ.
4. Присваивание строк выполняется с помощью функций библиотеки.
5. Для консольного ввода-вывода строк используются либо объекты cin и cout, либо функции библиотеки gets, scanf и puts, printf.
6. Ввод-вывод из файла может выполняться с помощью либо объектов классов ifstream и ofstream, либо функций библиотеки fgets, fscanf и fputs, fprintf.
7. 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 библиотеки gets и scanf.
8. Смешивать в одной программе ввод-вывод с помощью потоковых классов и с помощью функций библиотеки не рекомендуется.
9. Посимвольное чтение из файла неэффективно.
10. Разбивайте написание программы на последовательность шагов.
11. Выход за границы строки и отсутствие нуль-символа являются распространенными причинами ошибок в программах.
12. Средства, предоставляемые языком, влияют на алгоритм решения задачи, и поэтому перед тем, как продумывать алгоритм, необходимо эти средства изучить.
13. Программа, написанная с использованием функций, а не классов ввода-вывода, может получиться более быстродействующей, но менее безопасной.
14. Недостатком C-строк по сравнению с классом string является отсутствие проверки выхода строки за пределы отведенной ей памяти.

Задания¹

Вариант 1

Написать программу, которая считывает из текстового файла три предложения и выводит их в обратном порядке.

Вариант 2

Написать программу, которая считывает текст из файла и выводит на экран только предложения, содержащие введенное с клавиатуры слово.

¹ Задания на строки, приведенные в Учебнике на с. 159, рассчитаны на использование функций позиционирования курсора на экране и управления цветом символа и фона. Эти функции не поддерживаются стандартом, но входят в состав большинства старых оболочек, например Borland C++ 3.1 или Microsoft Quick C.

Вариант 3

Написать программу, которая считывает текст из файла и выводит на экран только строки, содержащие двузначные числа.

Вариант 4

Написать программу, которая считывает английский текст из файла и выводит на экран слова, начинающиеся с гласных букв.

Вариант 5

Написать программу, которая считывает текст из файла и выводит его на экран, меняя местами каждые два соседних слова.

Вариант 6

Написать программу, которая считывает текст из файла и выводит на экран только предложения, не содержащие запятых.

Вариант 7

Написать программу, которая считывает текст из файла и определяет, сколько в нем слов, состоящих из не более чем четырех букв.

Вариант 8

Написать программу, которая считывает текст из файла и выводит на экран только цитаты, то есть предложения, заключенные в кавычки.

Вариант 9

Написать программу, которая считывает текст из файла и выводит на экран только предложения, состоящие из заданного количества слов.

Вариант 10

Написать программу, которая считывает английский текст из файла и выводит на экран слова текста, начинающиеся и оканчивающиеся на гласные буквы.

Вариант 11

Написать программу, которая считывает текст из файла и выводит на экран только строки, не содержащие двузначных чисел.

Вариант 12

Написать программу, которая считывает текст из файла и выводит на экран только предложения, начинающиеся с тире, перед которым могут находиться только пробельные символы.

Вариант 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.

Вариант 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.

Вариант 15

Написать программу, которая считывает текст из файла, находит самое длинное слово и определяет, сколько раз оно встретилось в тексте.

Вариант 16

Написать программу, которая считывает текст из файла и выводит на экран сначала вопросительные, а затем восклицательные предложения.

Вариант 17

Написать программу, которая считывает текст из файла и выводит его на экран, после каждого предложения добавляя, сколько раз встретилось в нем введенное с клавиатуры слово.

Вариант 18

Написать программу, которая считывает текст из файла и выводит на экран все его предложения в обратном порядке.

Вариант 19

Написать программу, которая считывает текст из файла и выводит на экран сначала предложения, начинающиеся с однобуквенных слов, а затем все остальные.

Вариант 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 просто для логического объединения связанных между собой данных. В структуру, в противоположность массиву, 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 вся эта информация относится к одному событию, логично дать ему имя, чтобы впоследствии можно было к нему обращаться. Для этого описывается новый тип данных (обратите внимание на то, что после описания стоит точка с запятой):

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

Имя этого типа данных — `Event`. Можно описать переменные этого типа точно так же, как переменные встроенных типов, например:

```
Event e1, e2[10]; // структура и массив 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;
    char theme[100], name[100];
    int num;
} 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:

Задача 6.1. Поиск в массиве структур

```
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), например:

```
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 одного типа можно присваивать друг другу:

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

Но присваивание — это и все, что можно делать со структурами целиком. Другие операции, например сравнение на равенство или вывод, не определены. Впрочем, пользователь может задать их самостоятельно, поскольку структура является видом класса, а в классах можно определять собственные операции. Мы рассмотрим эту тему во второй части практикума.

Ввод/вывод структур, как и массивов, выполняется поэлементно. Вот, например, как выглядит ввод и вывод описанной выше структуры `e1` с использованием классов ввода-вывода (`<iostream.h>`):

```
cin >> e1.hour >> e1.min;
cin.getline(e1.theme, 100);
cout << e1.hour << ' ' << e1.min << ' ' << e1.theme << endl;
```

А вот вариант для любителей ввода-вывода в стиле С (подключается заголовочный файл `<stdio.h>`):

```
scanf("%d%d", &e1.hour, &e1.min); gets(e1.theme);
printf("%d %d %s", e1.hour, e1.min, e1.theme);
```

Структуры (но, конечно, не динамические) можно инициализировать перечислением значений их элементов:

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

Теперь, когда мы познакомились с основными сведениями о структурах, перейдем к решению задач.

Задача 6.1. Поиск в массиве 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, результаты и промежуточные величины

Исходные данные. База сотрудников находится в текстовом файле. Прежде всего надо решить, хранить ли в оперативной памяти одновременно всю информацию из файла или можно обойтись буфером на одну строку. Если бы сведения о сотруднике запрашивались однократно, можно было бы остановиться на втором варианте, но поскольку поиск по базе будет выполняться более одного раза, всю информацию желательно хранить в оперативной памяти, поскольку многократное чтение из файла крайне нерационально.

Максимальное 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.

В программу по условию требуется также вводить фамилии искомых сотрудников. Для хранения фамилии опишем строку символов той же длины, что и в базе.

Результаты. В результате работы программы требуется вывести на экран требуемые элементы исходного массива. Поскольку эти результаты представляют собой выборку из исходных данных, дополнительная память для них не отводится. Кроме того, необходимо подсчитать средний оклад для найденных сотрудников. Для этого необходима переменная вещественного типа.

Промежуточные величины. Для поиска среднего оклада необходимо подсчитать количество сотрудников, для которых выводились сведения. Заведем для этого переменную целого типа. Для описания формата входного файла будем использовать именованные константы.

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.

Необходимо решить, каким образом будет производиться выход из цикла вывода сведений о сотрудниках. Для простоты условимся, что для выхода из цикла вместо фамилии следует ввести слово end.

III. Программа и тестовые примеры

```
#include <fstream.h>
#include <string.h>
```

```
#include <stdlib.h>
// #include <windows.h>
int main(){
    const int l_name = 30, l_year = 5, l_pay = 10;
    l_buf = l_name + l_year + l_pay;
    struct Man {
        int birth_year;
        char name[l_name + 1];
        float pay;
    };
    const int l_dbase = 100;
    Man dbase[l_dbase];
    char buf [l_buf + 1];
    char name[l_name + 1];
    ifstream fin("dbase.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Ошибка открытия файла "; return 1; }

    int i = 0;
    while (fin.getline(buf, l_buf)) {
        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;
    float mean_pay = 0;

    while (true) {
        cout << " Введите фамилию или слово end: ";
        cin >> name;
        // OemToChar(name, name);
        if (strcmp(name, "end") == 0 )break;
        bool not_found = true;
        for (i = 0; i < n_record; i++) {
            if (strstr(dbase[i].name, name))
                if (dbase[i].name[strlen(name)] == ' ') {
                    strcpy(name, dbase[i].name);
                    CharToOem(name, name);
                    cout << 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;
}
if (n_man > 0) cout << " Средний оклад: " << mean_pay / n_man <<
endl;
return 0;
}

```

Рассмотрим приведенную выше программу подробно. В операторе 1 заданы именованные константы, в которых хранится формат входного файла, то есть длина каждого из полей записи (строки файла). Такой подход позволяет при необходимости легко вносить в программу изменения. Длина буфера, в который будет считываться каждая строка файла, вычисляется как сумма длин указанных полей.

В операторе 2 определяется структура Map для хранения сведений об одном сотруднике. Длина поля, в котором будет находиться фамилия, задана с учетом завершающего нуль-символа. В операторе 3 определяется массив структур dbase для хранения всей базы. Его размерность также задается именованной константой. В операторах 4 и 5 задаются промежуточные переменные: буфер buf для ввода строки из файла и строка name для фамилии запрашиваемого сотрудника.

В операторе 6 выполняется открытие файла dbase.txt для чтения. Предполагается, что этот файл находится в том же каталоге, что и текст программы, иначе следует указать полный путь. Входной файл следует создать в любом текстовом редакторе¹ до первого запуска программы в соответствии с форматом, заданным в условии задачи. Файл для целей тестирования должен состоять из нескольких строк, причем необходимо предусмотреть случай, когда одна фамилия является частью другой (например, Иванов и Ивановский). Не забудьте проверить, выдается ли диагностическое сообщение, если файл не найден.

Цикл 7 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 копирования строк strncpy, преобразования из строки в целое число atoi и преобразования из строки в вещественное число atof. Они были рассмотрены на предыдущем семинаре (см. с. 94). Обратите внимание на то, 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.

Обратите внимание на то, что при каждом проходе цикла выполняется проверка, не превышает ли считанное количество строк размерность массива. При тестировании программы в этот цикл следует добавить контрольный вывод на экран считанной строки, а также сформированных полей структуры. Для проверки выдачи диагностического сообщения следует временно задать константу l_dbase равной, а затем меньшей фактического количества строк в файле.

ВНИМАНИЕ

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

¹ О проблемах, связанных с различной кодировкой кириллицы в текстовых редакторах, работающих в среде MS-DOS или в среде Windows, будет сказано ниже.

Кстати, можно записать эту проверку и так:

```

while (fin.getline(buf, 1_buf) && i < l_dbase) {
    ...
    i++;
}
if (i >= l_dbase) { cout << " Слишком длинный файл "; return 1; }

```

В операторе 8 определяются две переменные: n_record для хранения фактического количества записей о сотрудниках и n_man — для подсчета сотрудников, о которых будут выдаваться сведения. Следует также не забыть обнулить переменную mean_pay, в которой в следующем цикле будет накапливаться сумма окладов.

Цикл поиска сотрудников по фамилии организован как бесконечный (оператор 9) с принудительным выходом (оператор 11). 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, поскольку пользователь вынужден для окончания работы с программой ввести слово end в нижнем регистре. Более удобным был бы выход из цикла по нажатию, например, клавиши Esc. К сожалению, в рамках стандарта это сделать невозможно, однако в большинстве библиотек есть функции типа getch и kbhit, позволяющие анализировать нажатие клавиш. В общем случае, несомненно, следует выбирать интерфейс, наиболее удобный для пользователя, поскольку именно для него и пишутся все без исключения программы.

В операторе 12 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

```

if (not_found) cout << " Такого сотрудника нет " << endl;

```

хорошо понятен без дополнительных комментариев.

В операторе 13 организуется цикл просмотра массива структур (просматриваются только заполненные при вводе элементы). Проверка совпадения фамилии сотрудника производится в два этапа. В операторе 14 с помощью функции strstr поиска подстроки определяется, содержится ли в поле базы name искомая последовательность букв, а в операторе 15 проверяется, есть ли непосредственно после фамилии пробел (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 каждой строки.

Когда вы дойдете до отладки этой части программы, то, вполне возможно, столкнетесь с непонятной на первый взгляд проблемой: программа не может найти запись с заданной фамилией. Впрочем, этот эффект проявляется, только если вы работаете в среде Windows и только на фамилиях, записанных в базе на русском языке; стоит только перейти на латиницу, как все начинает работать нормально. Здесь есть одна тонкость, связанная с разной кодировкой букв русского алфавита (кириллицы) в операционных системах MS DOS и Windows, о чем уже говорилось на первом семинаре (см. с. 17).

Напомним, что если вы работаете в интегрированной среде Visual C++ в режиме консольных приложений, то весь ввод-вывод осуществляется в кодировке ASCII. Поэтому поведение данной программы будет различным в зависимости от кодировки текста в файле `dbase.txt`, а последняя определяется тем, в каком текстовом редакторе вы создавали этот файл.

Допустим, что вы использовали один из редакторов под MS DOS, к примеру Dos Navigator или Far. В этом случае кодировка в файле и в окне вывода одна и та же, и программа работает нормально. Но если вы заполнили файл `dbase.txt` в редакторе NotePad или WordPad (кодировка ANSI), то программа не будет работать с русскоязычными фамилиями.

Если материал первого семинара еще не окончательно улетучился из вашей памяти, то вы знаете, что в таких случаях надо использовать функцию `OemToChar()`, переводящую символы из кодировки ASCII в кодировку ANSI для введенной с консоли фамилии (переменная `name`). Для этого достаточно раскомментировать оператор 10 в нашей программе. Аналогично, для нормального вывода текста в консольное окно необходимо обратное преобразование с помощью функции `CharToOem()` — раскомментируйте оператор 16. Применение указанных функций требует подключения заголовочного файла `<windows.h>` (раскомментируйте оператор 0).

Для подобных программ в инструкции для пользователя должно быть четко указано, при помощи каких текстовых редакторов можно произвести первоначальное заполнение файла базы данных.

Продолжим анализ программы. Мы остановились на том, что алгоритм составлен в предположении, что фамилия начинается с первой позиции записи в базе данных. Измените программу так, чтобы это ограничение можно было снять. Для этого придется проверить, стоит ли перед фамилией пробел в том случае, если она не начинается с первой позиции. Аналогичная проблема рассматривалась на предыдущем семинаре (задача 5.2). Внесите в тестовый пример изменения, необходимые для тестирования этой части программы.

Проверка переменной `n_man` в операторе 17 необходима для того, чтобы в случае, если пользователь не введет ни одной фамилии, совпадающей с фамилией в базе, не выполнялось деление на 0.

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(c)` и `toupper(c)`, переводящие переданный им символ с нижний и верхний регистр соответственно, аналогичные функции для символов русского алфавита придется написать самостоятельно.

Если в базе есть несколько сотрудников с одной и той же фамилией, программа выдаст сведения обо всех.

А теперь давайте рассмотрим вариант записи этой же программы с помощью библиотечных функций ввода-вывода:

```
#include <stdio.h>
#include <string.h>
```

```
// #include <windows.h> // 0
int main(){
    const int l_name = 30;
    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
        // OemToChar(name, name); // 4

        bool not_found = true;
        for (i = 0; i < n_record; i++) {
            if (strstr(dbase[i].name, name))
                if (dbase[i].name[strlen(name)] == ' ') {
                    strcpy(name, dbase[i].name);
                    CharToOem(name, name); // 5
                    printf("%30s%5i%10.2f\n", name, dbase[i].birth_year,
                           dbase[i].pay);
                    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);
    return 0;
}
```

Из всех именованных констант осталась одна, задающая длину поля фамилии (`l_name`, оператор 1). Все остальные константы определять нет смысла, потому что ввод осуществляется не через буферную переменную, а непосредственно в поля структуры с помощью функции чтения строки `fgets` и форматного ввода `fscanf` (оператор 2). Эта функция сама выполняет действия по преобразованию подстроки в число, которые мы явным образом задавали в предыдущей программе.

Как и в предыдущей версии, программу необходимо «настроить» на ожидаемую кодировку символов. Если вы работаете в среде Visual C++ и готовите текстовый файл `dbase.txt` с помощью Windows-ориентированного текстового редактора, то раскомментируйте операторы 4 и 5 для преобразования символов из кодировки ASCII в кодировку ANSI и обратно, а также оператор 0 для подключения заголовочного файла `<windows.h>`.

Мы упростили выход из цикла ввода запросов, теперь для завершения цикла достаточно нажать клавишу `Enter` (оператор 3). Для вывода сведений о сотруднике мы использовали функцию `printf` (оператор 6). Как видите, иногда старые добрые функции могут сделать программу более простой и наглядной! Это происходит, когда при вводе и выводе требуется форматирование разнотипных данных.

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 файл с тем же именем, что исходный. Для сортировки применим описанный в Учебнике на с. 59 метод выбора. При всей своей простоте он достаточно эффективен. Надеемся, что вы еще помните основную идею этого метода (мы его использовали в задаче 4.3): из массива выбирается наименьший элемент и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, наименьший из них меняется местами со вторым элементом, и так далее. Для упорядочивания требуется количество просмотров, на единицу меньшее, чем количество элементов в массиве (при последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива).

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
int main(){
    const int l_name = 30, l_year = 5, l_pay = 10.
    l_buf = l_name + l_year + l_pay;

    struct Man {
        int birth_year:
```

```
char name[l_name + 1];
float pay;
};

const int l_dbase = 100;
Man dbase[l_dbase];
char buf[l_buf + 1];
ifstream fin("dbase.txt", ios::in | ios::nocreate);
if (!fin) { cout << "Ошибка открытия файла " << 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 " << endl; return 1; }
}
int n_record = i;
fin.close();
ofstream fout("dbase.txt");
if (!fout){ cout << "Ошибка открытия файла " << endl; return 1; }

for (i = 0; i < n_record - 1; i++) {
// принимаем за наименьший первый из рассматриваемых элементов:
    int imin = i;
    // поиск номера минимального элемента из неупорядоченных:
    for (int j = i + 1; j < n_record; j++)
        if (dbase[j].birth_year < dbase[imin].birth_year) imin = j;
    // обмен двух элементов массива структур:
    Man a = dbase[i];
    dbase[i] = dbase[imin];
    dbase[imin] = a;
}
for (i = 0; i < n_record - 1; i++) {
    fout << dbase[i].name << dbase[i].birth_year << ' ' << dbase[i].pay <<
        endl;
}
fout.close();
cout << "Сортировка базы данных завершена" << endl;
return 0;
}
```

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.

Для того чтобы записать результаты в файл с тем же именем, файл, открытый для чтения, закрывается, а затем открывается файл с тем же именем для записи (говоря более строго, создается объект выходного потока *ostream* с именем *fout*). При этом старый файл на диске уничтожается и создается новый, пустой файл, в который 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, без сомнения, медитативное, но пользы от него немного. Преимущество бинарных файлов состоит в том, что, во-первых, при чтении/записи не тратится время на преобразование данных из символьной формы представления во внутреннюю и обратно, а во-вторых, при этом не происходит потери точности вещественных чисел. Кроме того, при работе с бинарными файлами широко применяется прямой доступ к информации путем установки текущей позиции указателя. Это дает возможность быстрого получения и изменения отдельных данных файла. Например, в данной задаче мы будем изменять оклад отдельных сотрудников, не затрагивая другие записи базы.

Бинарный файл открывается в двоичном режиме, а чтение/запись в него выполняются с помощью функций *fread* и *fwrite*.

Обе программы не представляют алгоритмических сложностей, поэтому мы не будем разбирать процесс их создания по этапам, а сразу приведем тексты. Отметим только, что хранить в памяти весь входной файл нет необходимости, вполне достаточно одной переменной структурного типа, в которой будет содержаться в каждый момент времени запись об одном сотруднике.

```
// Создание бинарного файла из текстового
#include <stdio.h>
#include <string.h>
```

```
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) {
        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); // отладочная печать
        fwrite(&man, sizeof(man), 1, fout);
    }
    fclose(fout);
    printf("Бинарный файл записан\n");
    return 0;
}
```

Для формирования записей в бинарном файле здесь применяется функция *fwrite*:

```
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 во второй программе 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 считанных элементов, которое может быть меньше, чем запрошенное.

```
// Корректировка бинарного файла
#include <stdio.h>
#include <string.h>
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);
int n_record = ftell(fout) / sizeof(man); // 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 %5i%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 %5i%10.2f\n", man.name, man.birth_year, man.pay);
}
fclose(fout);
printf("Корректировка завершена.\n");
return 0;
}

```

В операторе 1 открывается сформированный в предыдущей задаче бинарный файл. Обратите внимание на режим открытия: `r+` обозначает возможность чтения и записи, `b` — двоичный режим (мы говорили о нем на пятом семинаре, см. с. 106). Чтобы проконтролировать введенный номер записи, в переменную `n_record` заносится длина файла в записях (оператор 2). До этого указатель текущей позиции файла устанавливается на его конец с помощью функции `fseek`, следовательно, в результате вызова `ftell` будет получен размер файла в байтах.

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: меньше нуля или больше, чем их количество (при написании программы мы приняли, что первая запись в файле имеет номер 0).

Задача 6.4. Структуры в динамической памяти

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

Не тратя времени на предварительное обсуждение¹, сразу приведем текст программы.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
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 *fbn;
    if ((fbn = fopen("dbase.bin", "rb")) == NULL) {
        puts("Ошибка открытия файла\n"); return 1; }
    fseek(fbn, 0, SEEK_END);
    int n_record = ftell(fbn) / sizeof(Man);

    Man *man = new Man [n_record]; // 2

    fseek(fbn, 0, SEEK_SET);
    fread(man, sizeof(Man), n_record, fbn); // 3
    fclose(fbn);

    qsort(man, n_record, sizeof(Man), compare); // 4

    for (int i = 0; i < n_record; i++)
        printf("%s %5i%10.2f\n", man[i].name, man[i].birth_year,
               man[i].pay);
    return 0;
}

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

```

Рассмотрим моменты, которые отличают эту программу от предыдущих. Во-первых, это чтение из бинарного файла. После открытия файла мы, как и в предыдущей программе, заносим в переменную `n_record` его размер в записях, а затем выделяем в динамической памяти место под весь массив структур (оператор 2). Функция `fread` позволяет считать весь файл за одно обращение (оператор 4), после чего файл уже больше не требуется, поэтому лучше его сразу же закрыть.

¹ Ведь вы все равно его пропустите, правда?

Для сортировки мы в образовательных целях и для разнообразия использовали стандартную функцию `qsort` (оператор 5). Ее прототип находится в заголовочном файле `<stdlib.h>`. Функция может выполнять сортировку массивов любых размеров и типов. У нее четыре параметра:

- 1) указатель на начало области, в которой размещается упорядочиваемая информация;
- 2) количество сортируемых элементов;
- 3) размер каждого элемента в байтах;
- 4) имя функции, которая выполняет сравнение двух элементов.

На следующем семинаре мы подробно рассмотрим правила оформления функций и передачи имен функций в качестве параметров, а сейчас необходимо уяснить следующее: раз функция `qsort` универсальна, мы должны дать ей информацию, 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`. 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 правильной работы `qsort` требуется, чтобы наша функция имела два параметра — указатели на сравниваемые элементы. Они должны иметь тип `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 мы хотим упорядочить массив по убыванию, нужно 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`, введенную в стандарт C++ относительно недавно. Старые компиляторы могут ее не поддерживать. 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`, мы перенесли описание структуры, а вместе с ней и описание необходимой ей константы `l_name` 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. 36). 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 объем программы, но требует некоторых усилий по изучению их интерфейса и правильному оформлению вызова. В качестве небольшого заключительного упражнения измените эти функции так, чтобы они использовали операцию `reinterpret_cast`.

Давайте повторим основные моменты этого семинара.

1. Структуры применяются для логического объединения связанных между собой данных различных типов.
2. После описания структурного типа ставится точка с запятой.
3. Элементы структуры называются полями. Поля могут быть любого основного типа, массивом, указателем, объединением или структурой.
4. Для обращения к полю используется операция выбора: «точка» при обращении через имя структуры и «->» при обращении через указатель.
5. Структуры одного типа можно присваивать друг другу.
6. Ввод/вывод структур выполняется поэлементно.
7. 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 размещены 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 AEROFLLOT, 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 AEROFLLOT; 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 AEROFLLOT, 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 AEROFLLOT; 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 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

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 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

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 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

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 – это группа операторов, выполняющая законченное действие. К функции можно обратиться по имени, передать ей значения и получить из нее результат.

Функции нужны для упрощения структуры программы. Разбив задачу на подзадачи и оформив каждую из них в виде функций, мы поступаем по принципу, известному еще с древних времен: «Разделяй и властвуй». Передача в функцию различных аргументов позволяет записав ее один раз, использовать многократно для разных данных. Чтобы использовать функцию, не требуется знать, как она работает – достаточно знать, как ее вызвать. Точно так же мы включаем телевизор или пользуемся стоп-краном в самолете. Страшно подумать, сколько информации пришлось бы держать в голове, если бы требовалось в деталях знать устройство всех приборов, которые мы ежедневно применяем! В реальной жизни мы успешно ограничиваемся знанием интерфейса, то есть правил обращения (и общения).

Для использования функции тоже требуется знать только ее интерфейс. Интерфейс грамотно написанной функции определяется ее заголовком, потому что в нем указывается все, что необходимо для ее вызова: имя функции, тип результата, который она возвращает, а также 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 возвращается она одно значение типа `int` (код завершения). Функция может и не возвращать никакого значения, в этом случае должен быть указан тип `void`. Вот, к примеру, заголовок стандартной библиотечной функции, вычисляющей синус угла:

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

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

```
void *memcp(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 *memcp(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 и ее текст. Представьте, что прежде чем позвонить по телефону, вам нужно было бы разобрать его и рассмотреть все внутренности, чтобы убедиться, что красный проводочек не подключен 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. *Определение* функции, кроме заголовка, включает ее *тело*, то есть те операторы, которые выполняются при вызове функции, например:

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

В тексте программы может содержаться произвольное количество объявлений одной и той же функции и только одно определение (в этом функции не отличаются от других программных объектов). 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 и 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 того чтобы вызвать функцию, надо указать ее имя (тут прослеживается полная аналогия с реальной жизнью, 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

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

функции указано, сколько величин и какого типа ей требуется для успешной работы, значит, надо их ей передать. Если мясоперерабатывающий конвейер рассчитан на то, что на вход поступает корова, а на выходе получается колбаса, трудно рассчитывать на колбасу, подав на вход даже самый современный автомобиль.

Вызов функции, возвращающей значение определенного типа (то есть не имеющей тип `void`), может быть записан в любом месте, где по синтаксису допустимо выражение — в правой части оператора присваивания, в составе выражения, в цепочке вывода и так далее. Вот, например, как можно вызвать функции, приведенные выше:

```
double y, x1 = 0.34, x2 = 2;
y = sin(x1);
cout << y << ' ' << sin(x2) << endl;
y = sin(x1 + 0.5) - sin(x1 - 0.5);
char *cite = "Never say never";
char b[100];
memcp(b, cite, strlen(cite) + 1);
int summa, a = 2;
summa = sum(a, 4);
```

ВНИМАНИЕ

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

Теперь, когда мы получили общее представление о назначении и правилах записи функций, перейдем к рассмотрению задач.

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 уже 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);
```

Обратившись к задаче 2.5, мы видим, что для вычисления суммы ряда понадобятся две промежуточные переменные — для хранения очередного члена ряда и его номера. Эти переменные должны быть описаны внутри функции, поскольку вне ее они не нужны (вспомните, что еще на первом семинаре мы рекомендовали вам описывать переменные так, чтобы их область действия была минимальной из возможных).

Давайте рассмотрим текст программы:

```
#include <stdio.h>
#include <math.h>
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("----- \n");
    printf("|      X      |      Y      | \n");
    printf("----- \n");

    for (double x = Xn; x <= Xk; x += dX)
        printf("|%9.2lf      |%14.6g      | \n", x, cosh(x, eps));
    printf("----- \n");
    return 0;
}

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;
}
```

Как видите, за счет использования функции программа получилась более ясной и компактной, потому что задача была разделена на две: вычисление функции и печать таблицы. Кроме того, написанную нами функцию можно при необходимости без изменений 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 используемых в программе 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.

В этой программе для ввода-вывода мы применили не классы, а функции, унаследованные из библиотеки языка С, поскольку с их помощью, на наш взгляд, форматированный вывод записывается более компактно. Обратите внимание на спецификацию формата *g*. Она применяется для вывода вещественных чисел в широком диапазоне значений. Первое число модификатора (14) задает, как и для других спецификаций, ширину отводимого под число поля, а второе (6) — не точность, как в формате *f*, а количество значащих цифр. При этом число выводится либо в формате *f*, либо в формате *e* (с порядком) в зависимости от того, какой из них получится короче.

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 приведенной выше программе: вывести текстовое сообщение, сформировать какое-либо определенное значение функции (чаще всего это 0) и выйти из функции. Недостаток этого способа — печать диагностического сообщения внутри функции. Это нежелательно, а порой (например, когда функция входит в состав библиотеки) и вовсе недопустимо. Попробуйте задать в качестве исходных данных большие значения аргумента и высокую точность. Вы увидите, что 500 итераций для ее достижения недостаточно, и таблицу результатов «портиг» сообщение о том, что ряд расходится.

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 представляет собой целую величину, ненулевые значения которой 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 это решение выглядит так:

```
#include <stdio.h>
#include <math.h>
double cosh(double x, double eps, int &err);
```

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

```
int main(){
    double Xn, Xk, dx, eps, y;
    int err;
    printf("Enter Xn, Xk, dx, eps \n");
    scanf("%lf%lf%lf%lf", &Xn, &Xk, &dx, &eps);

    printf(" ----- \n");
    printf("|      X      |      Y      | \n");
    printf("----- \n");

    for (double x = Xn; x <= Xk; x += dx) {
        y = cosh(x, eps, err);
        if (err) printf("|%9.2lf | Ряд расходится! |\n", x);
        else    printf("|%9.2lf |%14.6g |\n", x, y);
    }
    printf(" ----- \n");
    return 0;
}

double cosh(double x, double eps, int &err) {
    err = 0;
    const int MaxIter = 500;
    double ch = 1, y = ch;
    for (int n = 0; fabs(ch) > eps; n++) {
        ch *= x * x / ((2 * n + 1)*(2 * n + 2));
        y += ch;
        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! Надеемся, вы обратили внимание на знак *&* перед параметром *err*. Это — признак передачи параметра по ссылке. Такой способ 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 этой 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, например:

```
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:

```
// прототип функции:  
double cosh(double x, double eps, int *err);  
  
// вызов функции:  
y = cosh(x, eps, &err); // & - взятие адреса  
  
// обращение к err внутри функции:  
*err = 0; // * - 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 один серьезный 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 нашей программы передача входных данных по константной ссылке выглядит так:

```
// прототип функции:  
double cosh(const double &x, const double &eps, int &err);  
  
// вызов функции:  
y = cosh(x, eps, err);  
// обращение к x и eps внутри функции не изменяется
```

Поскольку вопрос о передаче параметров очень важен, не поленимся повторить изложенное еще раз в краткой форме:

ВНИМАНИЕ

Входные данные функции надо передавать по значению или по константной ссылке, результаты ее 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 — написать функцию так, чтобы параметр ошибки передавался через возвращаемое значение. Это применяется в основном для функций вывода информации. 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. В этом случае 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 нулю, хотя ноль может и входить в множество допустимых значений результата. В этом случае у программиста нет средств отличить ошибочное значение от правильного. Например, таким образом реализованы уже 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.

Теперь, когда мы обсудили разные способы уведомления об ошибке в функции, подведем итоги:

ВНИМАНИЕ

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

Во второй части практикума мы рассмотрим еще один механизм уведомления о возникновении ошибки — генерацию исключения.

А 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);
```

Как упоминалось на втором семинаре, DBL_EPSILON – это константа, определенная в файле `<float.h>`. Ее значение равно минимальному числу, которое, будучи добавлено к единице, даст не равный единице результат. Теперь нашу функцию можно вызывать с одним параметром, к примеру:

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

ВНИМАНИЕ

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

Вариант прототипа функции с использованием параметра ошибки, а также значением точности по умолчанию выглядит так:

```
double cosh(const double x, int &err, const double eps = DBL_EPSILON);
```

Соответствующим образом изменится и вызов функции. Обратите внимание, что указание перед параметром ключевого слова `const` в данном случае (при передаче по значению) применяется только для того, чтобы четко указать, какие из параметров являются входными. В случае передачи по ссылке указание `const`, кроме того, дает возможность передавать на месте этого параметра константу.

Теперь давайте посмотрим на нашу программу с другой стороны (с рассмотренных она нам уже порядком надоела). Мы оформили в виде функции вычисление суммы ряда, однако задача вывода таблицы значений функции сама по себе достаточно типична и может встретиться в других задачах. Поэтому было бы логично оформить ее решение также в виде функции.

Задача 7.1-а. Передача в функцию имени функции

Назовем функцию вывода таблицы значений `print_tabl`. Прежде всего надо определить ее интерфейс. Для того чтобы вывести таблицу, нашей функции потребуется знать диапазон и шаг изменения значений аргумента, а также какую, собственно, функцию мы собираемся вычислять. Всё? Нет, не все: забыли, что в функцию вычисления суммы ряда надо передавать точность, поэтому точность следует включить в список параметров вызывающей ее функции `print_tabl`. Функция `print_tabl` не возвращает никакого значения, то есть перед ее именем надо указать `void`.

Как передать в функцию имя функции? Точно так же, как и любую другую величину: в списке параметров перед именем параметра указать его тип. До этого момента мы передавали в функцию величины стандартных типов, а теперь нам потребуется определить собственный тип. Тип функции определяется типом ее возвращаемого значения и типом ее 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 умолча-

нию нам, к сожалению, придется отказаться). Часто, если описание типа сложное, с целью улучшения читаемости программы задают для него 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 вид:

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

Запишем теперь текст программы, сведя к минимуму диагностику ошибок (при превышении максимально допустимого количества итераций функция завершается, возвращая 0, а вызывающая программа бесстрастно выводит это значение):

```
#include <stdio.h>
#include <math.h>
```

```
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);
    return 0;
}
```

```
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) {
```

```
    const int Maxiter = 500;

```

```
    double ch = 1, y = ch;

```

```
    for (int n = 0; fabs(ch) > eps; n++) {

```

```
        ch *= x * x / (2 * n + 1) / (2 * n + 2);

```

```
        y += ch;
    }
```

```

    if (n > MaxIter) return 0;
}
return y;
}

```

Функция `print_tab` предназначена для вывода таблицы значений любой функции, принимающей два аргумента типа `double` и возвращающей значение того же типа.

Как видите, наряду с большей общностью мы добились и лучшего структурирования программы, разбив ее на две логически не связанные подзадачи: вычисление функции и вывод таблицы. В главной программе остался только ввод исходных данных и вызов функции.

Задача 7.2. Передача одномерных 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 потребуется подсчитать количество положительных элементов в двух массивах, то есть выполнить для обоих массивов одни и те же действия. Следовательно, эти действия надо поместить в функцию. Интерфейс функции: входные данные — массив и количество его элементов, результат — количество положительных элементов в массиве. Таким образом, заголовок функции должен иметь вид:

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

Имя массива представляет собой указатель на его нулевой элемент, поэтому в функцию массивы передаются через указатели¹. Количество элементов в массиве должно передаваться отдельным параметром, потому что, в отличие от строк символов, использующих признак конца строки, для массивов общего вида никакого признака конца массива не существует.

Аналогичные задачи мы рассматривали на третьем семинаре, поэтому не будем останавливаться на алгоритме и сразу перейдем к написанию программы:

```

#include <iostream.h>
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];
}

```

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

```

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" << endl;
else if(n_posit(a, n) < n_posit(b, n))
    cout << " Во втором положительных больше" << endl;
else
    cout << " Однаковое количество" << endl;
return 0;
}

```

```

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;
}

```

В этой программе место под массивы выделяется в динамической области памяти, поскольку в задании не указано конкретное количество элементов. Однако функцию `n_posit` можно без изменений применять и для «обычных» массивов, потому что для каждого из них имя тоже является указателем на нулевой элемент, только константным. Например, опишем массив из 10 элементов и инициализируем первые шесть из них (оставшимся будут присвоены нулевые значения):

```

int x[10] = {2, 3, -1, -10, 4, -2}; // будет выведено значение 3
cout << n_posit(x, 10);

```

Заслуживает рассмотрения способ анализа результатов работы функции. Как видите, функция вызывается в составе выражения в условном операторе. Для перебора всех трех вариантов результата приходится вызывать ее для каждого массива дважды, что для больших массивов, конечно, нерационально. Чтобы избежать повторного вызова, можно завести две переменные, в которые записываются результаты обработки обоих массивов, а затем использовать эти переменные в условных операторах:

```

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

```

Надо сказать, что современные компиляторы обладают широкими возможностями оптимизации и сами отслеживают подобные ситуации, преобразуя код программы, но это не означает, что на эффективность своих программ вообще не надо обращать внимания. Главным же критерием при выборе варианта написания программы, тем не менее, остается простота ее структуры и читаемость.

Мы не получили выигрыша в длине приведенной выше программы, использовав для вычисления количества положительных элементов функцию, но причина этому — лишь простота задачи. В реальных программах в виде отдельной функции оформляются более крупные законченные фрагменты. Разные авторы рекомендуют различные конкретные цифры, задающие длину функций, но сходятся в одном: функция должна быть не очень короткой, но и не очень длинной. Чаще всего советуют создавать функции длиной в один-два экрана. Впрочем, придерживаться разумной умеренности бывает полезно не только при написании программ...

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.

Отличие передачи в функцию строки от передачи обычного массива состоит в том, что можно не передавать отдельным параметром размерность строки, а определять конец строки внутри функции по нуль-символу. Для простоты предположим, что числа могут быть либо целые, либо вещественные с фиксированной точкой и непустой дробной частью. Распространить действие программы на другие виды чисел предоставляется вам в виде самостоятельного упражнения.

```
#include <fstream.h>
#include <ctype.h>
int num_num(const char *str);
int main(){
    ifstream fin("test.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Нет файла test.txt" << endl; return 0; }
    const int len = 101;
    int i = 1;
    char str[len];
    while (fin.getline(str, len)) {
        cout << "В строке " << i << " содержится " << num_num(str) << " чисел " <<
            endl;
        i++;
    }
    return 0;
}

int num_num(const char *str) {
    int count = 0;
    while (*str) {
        if (isdigit(*str) && !isdigit(*(str + 1)) && *(str + 1) != '.') count++;
        str++;
    }
    return 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 двумерных массивов в функцию

Написать программу, определяющую, в какой строке целочисленной матрицы x находится самая длинная серия одинаковых элементов.

Под серией имеются в виду элементы, расположенные подряд.

В виде функции здесь удобно оформить решение основной задачи, оставив главной программе только ввод исходных данных и вывод результатов. Для удобства отладки ввод массива в программе выполняется из текстового файла. Первая строка файла содержит значения для m и n , каждая следующая строка — набор чисел для одной строки матрицы. Память под массив выделяется в цикле для того, чтобы можно было задавать обе размерности массива в виде переменных (см. материал четвертого семинара).

```
#include <fstream.h>
int ser_equals(int **a, const int m, const int n);
int main(){
    ifstream fin ("matrix.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Нет файла matrix.txt" << endl; return 0; }
    int m, n, i, j;

    fin >> m >> n;
    int **a = new int *[m]; // выделение памяти
    for(i = 0; i < m; i++) a[i] = new int [n];

    for (i = 0; i < m; i++) // ввод массива
        for (j = 0; j < n; j++) fin >> a[i][j];
    int line = ser_equals(a, m, n); // вызов функции
    if (line >= 0) cout << "Самая длинная серия в строке " << line;
    else cout << "Серий одинаковых элементов нет";
    return 0;
}

int ser_equals(int **a, const int m, const int n){
    int i, j, count, line = -1, maxcount = 0;
    for (i = 0; i < m; i++){
        count = 0;
        for (j = 0; j < n; j++){
            if (a[i][j] == a[i][j + 1]) count++;
            else count = 0;
            if (count > maxcount) maxcount = count;
        }
    }
    return maxcount;
} // 1
```

```

for (j = 0; j < n - 1; j++) {
    if (a[i][j] == a[i][j + 1]) count++; // 2
    else {
        if (count > maxcount) { // 3
            maxcount = count; line = i; }
        count = 0;
    }
}
if (count > maxcount) { // 4
    maxcount = count; line = i; }
}
return line;
}

```

Алгоритм работы функции прост: в каждой строке выполняется сравнение соседних элементов (оператор 2). Если они равны, мы находимся внутри серии, при этом увеличиваем ее текущую длину. Она накапливается в переменной count, которая обнуляется перед обработкой каждой строки (оператор 1). Если же элементы не равны, это означает либо окончание серии, либо просто одиничный элемент (оператор 3). В этом случае надо посмотреть, не является ли данная серия самой длинной из рассмотренных и, если да, то запомнить ее длину и номер строки, в которой она встретилась (оператор 4). Для подготовки к анализу следующих серий в этой же строке надо обнулить счетчик count. Аналогичная проверка после цикла просмотра строки (оператор 5) выполняется для серии, которая расположена в конце строки, поскольку в этом случае ветвь else выполняться не будет.

Если в массиве нет ни одной серии одинаковых элементов, функция вернет значение, равное -1.

Задача 7.5. 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 добавление этой информации в бинарный файл, поэтому в нашей программе будет две функции. Первая функция 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 напишем еще одну функцию, которая будет по введенному номеру записи выводить ее на экран.

Ниже приведен текст программы.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
// #include <windows.h>
const int l_name = 30; // 0

```

```

struct Man {
    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[2];
    char filename[] = "dbase.bin";
    do {
        contin = false;
        if (append2binfile(read_data(), filename) != 0) {
            puts("Ошибка при записи в файл "); return 1; }
        puts("Продолжить (y/n)?"); gets(y_n);
        if ((y_n[0] == 'y') || (y_n[0] == 'Y')) contin = true;
    } while (contin);
    print_from_bin(filename);
    return 0;
}

int append2binfile(const Man &man, const char* filename) {
    FILE *fout;
    if ((fout = fopen(filename, "ab")) == NULL) return 1;
    int success = fwrite(&man, sizeof(man), 1, fout);
    fclose(fout);
    if (success == 1) return 0; else return 2;
}

int print_from_bin(const char * filename) {
    int num;
    Man man;
    FILE *f;
    if ((f = fopen(filename, "rb")) == NULL) return 1;
    fseek(f, 0, SEEK_END);
    int n_record = ftell(f) / sizeof (man);
    while (true) {
        puts("Введите номер записи или -1: "); scanf("%i", &num);
        if (num < 0 || num > n_record) break;
        fseek(f, num * sizeof(man), SEEK_SET);
        fread(&man, sizeof(man), 1, f);
        // CharToOem(man.name, man.name); // 1
        printf("%30s%5i%10.2f\n", man.name, man.birth_year, man.pay);
    }
}

```

```

    return 0;
}

Man read_data(){
    Man man;
    char buf[80];
    char name[1_name + 1];
    puts("Введите фамилию И.О. "); gets(name);
    if (strlen(name) < 1_name)
        for (int i = strlen(name); i < 1_name; i++) name[i] = ' ';
    name[1_name] = 0;
    // OemToChar(name, name);
    // 2
    strcpy(man.name, name, 1_name + 1);
    do {
        puts("Введите год рождения "); gets(buf);
        // 3
        while ((man.birth_year = atoi(buf)) == 0);
        do {
            puts("Введите оклад "); gets(buf);
            // 4
            while (!(man.pay = atof(buf)));
        }
        return man;
    }
}

```

В функции ввода `read_data` предусмотрено заполнение пробелами оставшейся части строковой переменной `name`, чтобы формат имени был идентичен формату ввода в текстовом файле.

ПРИМЕЧАНИЕ

Здесь так же, как и в задаче 6.1, необходимо учесть возможные различия в кодировке символов. Если вы работаете в среде Visual C++, а бинарный файл был создан из текстового файла, подготовленного в текстовом редакторе с кодировкой ANSI, то вам нужно раскомментировать операторы `0, 1 и 21`.

Обратите внимание на то, как в этой функции выполняется 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 3 и 4 записано в двух разных вариантах, чтобы вы сами могли оценить, какой из них вам более понятен (профессионалы предпочли бы второй, более лаконичный вариант).

Как видите, структура, в отличие от массива, может быть возвращаемым значением функции. В этой программе структура передавалась в функцию по константной ссылке; можно передавать ее и по значению, что несколько хуже, потому что в этом случае затрачивается время на копирование и требуется дополнительное место в стеке параметров.

¹ См. пояснения к задаче 6.1.

Задача 7.6. Рекурсивные функции

Написать программу упорядочивания массива методом быстрой сортировки, используя рекурсию.

Пришло время выполнить обещание, данное на третьем семинаре, — показать рекурсивную реализацию метода быстрой сортировки. Если вы не знаете, что такое рекурсивные функции, загляните в Учебник на с. 82. Говоря кратко, рекурсивной называется функция, в которой имеется обращение к ней самой. Освежите в памяти алгоритм быстрой сортировки, который мы рассматривали, решая задачу 3.3. Там использовалась «процедура разделения», применяемая к фрагменту массива (изначально — ко всему массиву). На каждом шаге образовывались две половины текущего фрагмента, и к ним снова нужно было применять процедуру разделения. То есть по своей сути алгоритм является рекурсивным, и если не здесь применять рекурсивные функции, то где?..

К счастью, 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 регистрах переменных, поэтому предыдущее состояние выполняемой функции сохраняется, и к нему впоследствии можно вернуться (так и произойдет, если только ваша программа где-нибудь не зависнет).

Одна из возможных версий программы сортировки приведена ниже.

```

#include <iostream.h>

void qsort(float* array, int left, int right);

int main(){
    const int n = 10;
    float arr[n];
    int l, r;

    cout << "Введите элементы массива: ";
    for (i = 0; i < n; i++) cin >> arr[i];

    l = 0; r = n - 1; // левая и правая границы начального фрагмента
    qsort(arr, l, r); // 1
    for (i = 0; i < n; i++) cout << arr[i] << ' ';
    return 0;
}

void qsort(float* array, int left, int right) {
    int l = 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;
        i++;
        j--;
    }
}
if (left < j) qsort(array, left, j); // 2
if (i < right) qsort(array, i, right); // 3
}

```

Мы не будем подробно анализировать эту программу, поскольку алгоритм вам знаком по задаче 3.3. Отметим только, что 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 половинки текущего фрагмента, и в операторе 3 – для сортировки его правой половинки. Надеемся, что сравнение этой программы с предыдущей нерекурсивной версией (задача 3.3) вызовет у вас истинное эстетическое наслаждение¹.

Однако, у рекурсии есть и недостатки: во-первых, такую программу труднее отлаживать, поскольку требуется контролировать глубину рекурсивного обращения, во-вторых, при большой глубине стек может переполниться, а в-третьих, использование рекурсии повышает накладные расходы (например, в данном случае в стеке сохраняются отнюдь не два числа, представляющие собой границы 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

Описанные в приложениях интегрированные среды Visual C++ 6.0 и Borland C++ 3.1, как и подавляющее большинство других, поддерживают создание, компиляцию и сборку многофайловых проектов. С технологией их создания вы можете ознакомиться по приложениям, 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.

¹ Если этого не произошло, вернитесь к третьему семинару и выполните все упражнения, вплоть до настоящего семинара, повторно.

Напомним, что пользуясь технологией исходящего 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 (как уже указывалось, наиболее предпочтительным считается размер не более одного-двух экранов текстового 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 так называемый *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 указанного набора вызывается из какого-то другого исходного модуля уuu.cpp, то вы обязаны включить в этот модуль заголовочный файл xxx.h с помощью директивы #include¹. Негласное правило стиля программирования на C++ требует включения этого же заголовочного файла (с помощью #include) и в исходный файл xxx.cpp.

Теперь о *глобальных переменных*. В многофайловом проекте возможны два «вида глобальности». Если некоторая глобальная переменная g1var1 объявлена в файле xxx.cpp с модификатором static, то она видима от точки определения до конца этого файла, то есть область ее видимости ограничена файлом. Если же другая глобальная переменная g1var2 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 это объявление поместить 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 имен².

Теперь обратим ваше внимание на проблему *повторного включения заголовочных файлов*. Проблема может возникнуть при иерархическом проектировании структур данных, когда в некоторый заголовочный файл уuu.h включается при помощи директивы #include другой заголовочный файл xxx.h (например, для использования типов, определенных в этом файле). Впрочем, лучше рассмотреть эту проблему на конкретном примере.

¹ Попробуйте не включить и посмотрите на реакцию компилятора.

² Пространства имен рассматриваются в Учебнике на с. 99.

Ниже приведены тексты очень 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 просто создается объект типа «Прямоугольник»¹ и выводятся координаты его левого верхнего и правого нижнего 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 наподобие следующего: «error ...: 'Point': 'struct' type redefinition».

```
////////// Файл Point.h //////////
// Объявления типов
struct Point {
    int x;
    int 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;
    Point 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;
}

void GetLT(Rect& rect, Point& lt) {
    lt = rect.leftTop;
}

void GetRB(Rect& rect, Point& rb) {
    rb = rect.rightBottom;
}
#include <stdio.h>

#include "Point.h"
#include "Rect.h"

////////// Файл Main.cpp //////////
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);
    return 0;
}
```

Каков 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
```

¹ Под «объектом» мы здесь понимаем переменную типа struct.

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

Применительно к нашему примеру файл Point.h должен содержать следующий текст:

```
////////// Файл Point.h //////////
#ifndef POINT_H
#define POINT_H
// Объявления типов
struct Point {
    int x;
    int y;
};
// Прототипы функций
void SetXY(Point& point, int x, int y);
int GetX(Point& point);
int GetY(Point& point);

#endif /* POINT_H */
```

Рекомендуем вам проверить рассмотренный пример на вашем компиляторе.

Задача 7.7. Многофайловый проект — форматирование текста

Написать программу форматирования текста, читаемого из файла unformat.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, то есть результирующий текст должен находиться в позициях margin + 1 .. margin + max1_line. Кроме этого, программа должна подсчитывать общее количество слов в тексте.

На примере этой задачи мы показываем технологию 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 определяем спецификации нужных нам функций:

- void DefInter (const char* pline, int & base_int, int & add_int, int & inter) определяет для строки, на которую указывает pline, количество межсловенных 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 межсловенных 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.

Ниже приводится содержимое этих файлов.

```
////////// Файл Task7_7.cpp
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "edit.h"

// Глобальные переменные
const int max1_line = 63;
const int margin = 5;
```

```

int main(){
FILE* finp;
FILE* fout;
char line[maxl_line + 1];
int b_i, a_i, start, next, inter;
int nword = 0;

printf("Работает программа Task7_7.\n");
if(!(finp = fopen("unformat.txt", "r"))) {
    printf("Файл unformat.txt не найден.\n"); exit(0);
}
printf("Читается файл unformat.txt.\n");
if(!(fout = fopen("formatd.txt", "w"))) {
    printf("Файл formatd.txt не создан.\n"); exit(0);
}
printf("Выполняется запись в файл formatd.txt.\n");

while(GetLine(finp, line)) {
    DefInter(line, b_i, a_i, inter);
    PutInterval(fout, margin);
    next = PutWord(fout, line, 0, nword);

    for (int i = 0; i < inter; i++) {
        start = SearchNextWord(line, next);
        PutInterval(fout, b_i);
        if (a_i) { a_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
// Прототипы функций
void DefInter(const char* pline, int& base_int, int& add_int,
    int& inter);
int GetLine(FILE*, char*);
void PutInterval(FILE*, const int);

```

```

int PutWord(FILE*, const char*, const int, int&);
int SearchNextWord(const char*, const int);

// Глобальные переменные
extern const int maxl_line;

/////////////////////////////////////////////////////////////////////////
// Файл Edit.cpp
#include <stdio.h>
#include <string.h>
#include "edit.h"

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;

    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 + max_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;
}
///////////////////////////////////////////////////////////////////

```

Обратите внимание, что имена функций мы здесь записали (для разнообразия) в стиле Microsoft, с использованием прописных букв для выделения осмысленных частей имени. Константу `max_line` следует задавать большей, чем максимальная длина строки исходного файла. В качестве самостоятельного упражнения измените программу так, чтобы можно было форматировать текст и в более узкую колонку, чем в исходном файле.

Приведем результаты тестирования этой программы.

Содержимое входного файла `unformat.txt`¹:

¹ Текст взят из афоризмов Б. Грасиана («Карманный оракул», 1647 г.).

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 добре славе злоба людская сразу подметит - и упорно в них метит. Особенно ценно искусство скрывать свой недостаток, обращая его в преимущество. Так. Цезарь скрывал свою плесть 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 добре славе злоба людская сразу подметит - и упорно в них метит. Особенно ценно искусство скрывать свой недостаток, обращая его в преимущество. Так. Цезарь скрывал свою плесть лавровым венком.

Протестируйте эту программу на других текстах.

Давайте повторим наиболее важные моменты этого семинара.

1. Функция — это именованная последовательность операторов, выполняющая законченное действие. Функции нужны для упрощения структуры программы.
2. Интерфейс грамотно написанной функции определяется ее заголовком.
3. Для вызова функции надо указать ее имя и набор аргументов.
4. В определении, в объявлении и при вызове функции типы и порядок следования аргументов и параметров должны совпадать.
5. Передача параметров в функцию может выполняться по значению или по адресу.
6. 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.
7. При написании функции нужно предусмотреть все возможные ошибки и обеспечить пользователя функции средствами их диагностики.
8. Печать диагностических сообщений внутри функции нежелательна.
9. Функция может иметь несколько параметров со значениями по умолчанию. Они должны находиться в конце списка параметров.

10. Массивы всегда передаются в функцию по адресу. Количество элементов в массиве должно передаваться отдельным параметром.
11. Рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь. При использовании рекурсии следует учитывать возникающие при этом проблемы и накладные расходы.
12. В многофайловых проектах важно грамотно разбить задачу на подзадачи и распределить функции по файлам.
13. Для предотвращения ошибок компиляции, связанных с повторным включением заголовочных файлов, следует использовать так называемые 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 и 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, руководствуясь списком аргументов.

Если же 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 генерирует текст функции для конкретных типов данных, с которыми выполняется вызов, и программисту не придется поддерживать несколько практически одинаковых функций.

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 своем комфорте при отладке программы, а что может быть важнее для «настоящего программиста»? Поэтому проще сразу оформить печать в виде функций, например таких:

```
void print( char* str, const int i, const int j ) {
```

6 Зад. 784

```

cout << str << '|' << oct << setw(4) << i << '|' << setw(4) << j <<
'|' << endl;
}

void print(float mas[], const int n) {
    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) {
    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 восьмичной форме, разделенных вертикальными черточками для читаемости. Под каждое число отводится по 4 позиции (действие манипулятора `setw` распространяется только на ближайшее выводимое поле).

Во второй функции для вывода вещественных значений по четыре числа на строке задается вид вывода с фиксированной точкой и точностью в два десятичных 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 шестому семинару структуры так, чтобы они не склеивались между собой. Манипулятор `setw` устанавливает ширину следующего за ним поля. Это приведет к тому, что фамилии будут выведены с отступом от края экрана. Вызов этих функций в программе 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. Также оформим в виде отдельной функции чтение базы из файла — и для лучшего структурирования программы, и для того, чтобы в случае необходимости было легче заменить эту функцию на другую, например на чтение из бинарного файла.

```

#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <iomanip.h>
const int l_name = 30, l_year = 5, l_pay = 10,
         l_buf = l_name + l_year + l_pay;
struct Man {
    int birth_year;
    char name[l_name + 1];
    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(){
    const int l_dbase = 100;
    Man dbase[l_dbase];
    int n_record = 0;
}

```

```

if (read_dbase("txt6.txt", dbase, l_dbase, n_record) != 0) return 1;

int option;
int 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);
return 0;
}

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;
        }
    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, ios::in | ios::nocreate);
    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;
}

```

Добавьте в базу еще одно-два осмысленных поля (например, отдел, стаж или количество детей) и напишите еще одну перегруженную функцию для выборки по новому критерию. Если тип критерия будет такой же, как один из рассмотренных, то, чтобы компилятор мог различать варианты, можно добавить в функцию дополнительный параметр.

Ниже приведены *правила описания перегруженных функций*.

- Перегруженные функции должны находиться в одной области видимости, иначе произойдет скрытие 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

Итак, мы создали несколько перегруженных функций, а теперь предоставим аналогичную возможность компилятору: зададим шаблон функции, а компилятор пусть самостоятельно создает столько перегруженных функций, для скольких ти-

пов данных нам потребуется вызвать шаблон. Конечно, этот фокус пройдет у нас только в том случае, если реализуемый алгоритм независим от типа данных. Таким образом, области применения перегрузки функций и шаблонов отличаются: перегруженные функции мы применяем 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 как в остальной части заголовка, так и в теле функции. Параметризованный тип – это всего лишь 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 шаблона: а) *явный*, когда объявляется заголовок функции, в котором все параметризованные типы заменены на конкретные типы, известные в этот момент в программе, б) *неявный*, когда создание экземпляра функции происходит автоматически, если встречается фактический вызов функции.

Для того чтобы вы прониклись разнообразием возможностей языка 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 саму функцию будет передаваться два аргумента: указатель на массив и длина этого массива.

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

template <class T> T Max(T *b, int n);

int main(){
    const int n = 20;
    int i, b[n];
    cout << "Введите " << n << " целых чисел:" << endl;
    for (i = 0; i < n; i++) cin >> b[i];
    cout << Max(b, n) << endl;
```

```
double a[] = {0.22, 117.2, -0.08, 0.21, 42.5};
cout << Max(a, 5) << endl;
char *str = "Sophisticated fantastic template";
cout << Max(str, strlen(str)) << endl;
return 0;
}
```

```
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));
```

Этот способ применяется в тех случаях, когда тип не определяется по виду оператора вызова функции.

Аналогично обычным параметрам функции, можно задавать значение параметра шаблона по умолчанию.

ВНИМАНИЕ

При работе с многофайловым проектом нужно не забывать, что если какой-то 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`.

Давайте повторим основные моменты этого семинара.

1. Перегрузкой функций называется использование нескольких функций с одним именем и различными типами параметров.
2. Перегрузка применяется, когда одно и то же по смыслу действие реализуется по-разному для различных типов или структур данных.
3. 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.
4. Функции не могут быть перегружены, если описание их параметров отличается только модификатором `const` или использованием ссылки.

5. Шаблоны функций применяются для записи идентичных действий над данными различных типов.
6. Инстанцирование шаблона функции — это создание компилятором конкретного варианта функции.
7. Шаблоны можно перегружать как шаблонами, так и обычными функциями.

Задания

Выполнить задания третьего и четвертого семинара, 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 отдельных 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;
    Node *p;
};
```

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 задаче 3.3. А может быть, вы пойдете дальше и подумаете: «Да сколько же можно!». Или еще что-нибудь подобное. И вот тут мы с вами не согласимся, потому что когда мы решали эту задачу, нам пришлось выделить для хранения границ неотсортированных фрагментов массива два массива, причем большая часть этой памяти пропадала совершенно зря. Но это еще полбеды, а настоящая беда может случиться, если массив для сортировки будет настолько большой длины, что еще два выделить просто не удастся. И вот тут нам на помощь приходит динамическое выделение памяти — когда выделяется нужное количество байт 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 использовать удобнее всего 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 нам требуется.

Для работы со стеком достаточно одной переменной — указателя на его вершину. Назовем ее *top*. Каждый элемент стека должен содержать два целых числа, представляющих собой левую и правую границы фрагмента массива, и указатель на следующий элемент:

```
struct Node {                                // Элемент стека
    int left, right;
    Node* p;
}:
Node* top = 01;                          // Вершина стека
```

Удобно оформить занесение и выборку элемента в виде отдельных функций.

Функция помещения в стек обычно называется *push*, а выборки — *pop*. Все необходимое передается функциям через параметры. В отличие от Учебника (с. 120) мы не будем использовать отдельную функцию для занесения первого элемента в стек, так как если указателю *top* присвоить 0 перед первым обращением к функции *push()*, то функция *push* вполне прилично справится с созданием первого элемента стека:

```
// Занесение в стек
Node* push(Node* top, const int l, const int r) {
    Node* pv = new Node;                      // 1
    pv->left = l;                            // 2
```

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 этой константе нет необходимости.

```
pv->right = r;                           // 3
pv->p = top;                            // 4
return pv;                               // 5
}
// Выборка из стека
Node* pop(Node* top, int& l, int& r) {
    Node* pv = top->p;                   // 6
    l = top->left;                       // 7
    r = top->right;                      // 8
    delete top;                           // 9
    return pv;                           // 10
}
```

Рассмотрим эти функции, самым тщательным образом вникая в детали. 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 тот 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 этим 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 «затакивающий» 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, просто изменив поля, составляющие его информационную часть, и соответствующие параметры.

Теперь можно заменить в задаче 3.3 решение с массивами на «классический» стек:

```
#include <fstream.h>
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);

// -----
int main() {
    int i, n;
    float middle, temp;
    ifstream fin("sort1.txt", ios::in | ios::nocreate);
    if (!fin) { cout << "Нет файла sort1.txt" << endl; return 0; }
    fin >> n;
    float* arr = new float[n];
    for (i = 0; i < n; i++) fin >> arr[i];
    // Сортировка
    int j, left, right;
    Node* top = 0;
    top = push(top, 0, n - 1);

    while (top) {
        top = pop(top, left, right);
        while (left < right) {
            // Разделение { arr[left] .. arr[right] }
            i = left; j = right;
            middle = arr[(left + right) / 2];
            while (i < j) {
                while (arr[i] < middle) i++;
                while (middle < arr[j]) j--;
                if (i <= j) {
                    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
        }
    }
    // Вывод результата
    for (i = 0; i < n; i++) cout << arr[i] << ' ';
    cout << endl;
    return 0;
}

// ----- Занесение в стек
Node* push(Node* top, const int l, const int r) {

```

```

Node* pv = new Node;
pv->left = l;
pv->right = r;
pv->p = top;
return pv;
}

// ----- Выборка из стека
Node* pop(Node* top, int &l, int &r) {
    Node* pv = top->p;
    l = top->left;
    r = top->right;
    delete top;
    return pv;
}

```

Для большей общности программы ввод с клавиатуры заменен на 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 шестом и седьмом семинарах. Специфика этой задачи состоит в том, что размер базы не ограничен, поэтому для ее хранения в оперативной памяти мы воспользуемся односвязным линейным списком (пример работы с двусвязным списком приведен в Учебнике на с. 115). Каждый элемент односвязного списка включает информационную часть (фамилию, год рождения, оклад) и указатель на следующий элемент. Признаком конца списка служит ноль 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;
    Man *next;
};

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 C++ не входят функции позиционирования курсора на экране, управления цветом и работы с экраном в графическом режиме, поскольку они системнозависимые, поэтому наши возможности ограничены. Вы же, наши любезные читатели, с помощью доступных в вашей среде функций вольны украсить свою программу цветистыми рамочками и разнообразными звуками в соответствии со своими эстетическими представлениями. 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;
- 7) выход.

Каждый пункт этого меню, кроме последнего, оформим в виде отдельной функции. Попытаемся путем логических 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 потребуется 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, так же, как и функция добавления, возвращает указатель на начало списка на тот 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. Поскольку реального заказчика у нас нет, 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);
void find(Man *beg, int birth_day);
void find(Man *beg, float pay);
```

Вторая и третья функции выполняют поиск и вывод списка сотрудников по заданному году рождения и окладу. Первая функция выполняет поиск по фамилии, выводит сведения о сотруднике и возвращает указатель на найденный элемент или ноль, если элемент не найден. Кроме того, мы предусмотрительно добавили в список параметров указатель на элемент, предшествующий найденному. Он передается по адресу (как указатель на указатель¹), 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. Эта функция аналогична функции удаления за исключением того, что указатель на начало списка измениться не может, поэтому функция возвращает признак успешности выполнения корректировки (просто так, 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:

```
Man read_man();
int menu();
```

¹ Альтернативный способ передачи указателя по адресу – с использованием ссылки на указатель – будет показан в задаче 9.4.

Приведем текст программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*
#include <windows.h>
char bufRus[256];
char* Rus(const char* text) {
    CharToOem(text, bufRus);
    return bufRus;
}
*/
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", ios::in | ios::nocreate);
    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;
}

return 0;
}

// ----- Добавление сотрудника
Man* add(Man* beg, const Man& man) {
    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;
}

// ----- Корректировка сведений
int edit(Man* beg) {
    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 в списке 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;
    }
}

7 Зад. 784
```

```

*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;
    }
}
// ----- Поиск
void find_man(Man* beg) {
    char buf[1_name];
    int birth_day, option;
    float pay;

    do {
        puts("1 - поиск по фамилии, 2 - по году рождения,\n\
            3 - по окладу, 4 - отмена\n");
        gets(buf);
    } while (!(option = atoi(buf)));

    switch (option) {
    case 1: get_name(buf);
        Man *prev;
        Man *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");
    } while (!(pay = (float)atof(buf)));
        find(beg, pay);
        break;
    case 4: return;
    default:
        puts("неверный режим\n");
    }
}
// ----- Запрос фамилии
void get_name(char* name) {
    puts("Введите фамилию И.О. ");
    gets(name);
    // DemToChar(name, name); // 3
}
// ----- Вывод меню
int menu() {
    char buf[10];
    int option;
    do {
        puts("-----");
        puts("1 - добавление сотрудника\t 4 - корректировка сведений");
        puts("2 - удаление сотрудника\t 5 - вывод базы на экран");
        puts("3 - поиск сотрудника\t 6 - вывод базы в файл");
        puts("\t\t\t 7 - выход");
        gets(buf);
        option = atoi(buf);
    } while (!option);
    return option;
}
// ----- Вывод базы на экран
void print_dbase(Man* beg) {
    Man* pv = beg;
    while (pv) {
        printf("%s%5i%10.2f\n", pv->name, pv->birth_day, pv->pay);
        pv = pv->next;
    }
}
// ----- Чтение базы из файла
Man* read_dbase(char* filename) {
    FILE* fin;
    Man man, *beg = 0;

    if ((fin = fopen(filename, "r")) == 0) {
        printf("Нет файла %s\n", filename);
        return 0;
    }
}

```

```

}

while (!feof(fin)) {
    fgets(man.name, l_name, fin);
    fscanf(fin, "%i%f\n", &man.birth_day, &man.pay);
    beg = add(beg, man);
}
fclose(fin);
return beg;
}
// ----- Ввод информации о новом сотруднике
Man read_man() {
    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;
}
// ----- Удаление сотрудника
Man* remove(Man* beg) {
    char name[l_name];
    get_name(name); // Кого удаляем?
    Man* prev;
    Man* pv = find(beg, name, &prev);
    if (pv) { // Если нашли
        if (pv == beg) // Удаление из начала списка
            beg = beg->next;
        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_dbbase(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;
    }

```

ВНИМАНИЕ

Здесь так же, как и в задаче 6.1, необходимо учсть возможные различия в кодировке символов кириллицы. Если вы работаете в среде Visual C++, а текстовый файл базы данных подготовлен в текстовом редакторе с кодировкой ANSI, то вам нужно: а) раскомментировать оператор 3, б) подключить функцию Rus(), убрав скобки комментария в строках 1 и 2, в) заменить все строковые константы в операторах вывода на вызов функции Rus(), подставляя в качестве аргумента заменяемую константу.

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. При отладке вывода в файл (функция write_dbbase) рекомендуем вам задать имя, отличное от исходного файла, с тем, чтобы случайно его не испортить.

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, единственный действием которых является вывод сообщения о том, что эта функция была вызвана.

В качестве самостоятельного упражнения добавьте в эту программу функцию формирования упорядоченного списка. За образец можно взять аналогичную функцию, приведенную в Учебнике на с.119.

Забегая немного вперед, предложим вам 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[l_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. Очередь

Написать программу учета для автосервиса, выполняющего кузовные работы и ремонт двигателей. При записи на обслуживание заполняется заявка, в которой указываются фамилия владельца, марка автомобиля, вид работы, дата приема заказа и стоимость ремонта. После выполнения работы распечатывается квитанция.

Интерфейс программы организуем, как и в предыдущей задаче, в виде меню. Предоставим пользователю программы следующие возможности:

- 1) добавление заявки;
- 2) распечатка квитанции о выполнении работы;
- 3) вывод списка заявок на экран;
- 4) вывод списка заявок в файл (поскольку ремонт автомобиля – дело сложное и может затянуться на месяц-другой);
- 5) выход.

Если для простоты предположить, что все клиенты обслуживаются в порядке очереди, то для хранения информации лучше всего использовать одноименную динамическую структуру данных – очередь. Очередь и стек являются частными случаями линейного списка. Для очереди определены всего две операции – помещение в конец и выборка из начала. При выборке элемент удаляется из очереди (аналогично стеку и в противоположность списку). Таким образом, используемая в программе структура данных не позволит выбрать для выполнения заявку, если до нее еще не дошла очередь¹.

Сделаем еще одно наивное предположение о том, что в автосервисе существует специализация, то есть работы выполняет не мастер Левша в одиночку, а две бригады. Одна бригада занимается кузовными работами, другая – ремонтом двигателей. Наша программа должна помочь им организовать работу без простояев, поэтому представляется логичным организовать отдельную очередь для каждого вида работ. В этих очередях будут храниться элементы одного и того же типа – заявки на выполнение работы:

```

const int l_name = 20, l_model = 20, l_work = 80;
struct Order {

```

¹ Тем фактом, что такую программу сегодня не приобретет ни один автосервис, придется с прискорбием пренебречь. Но после построения правового государства этот код пойдет нарасхват.

```

    char name[l_name]; // Фамилия И.О.
    char model[l_model]; // Марка автомобиля
    char work[l_work]; // Описание вида работы
    time_t time; // Дата и время приема заявки
    float price; // Стоимость работ
    Order *next; // Указатель на следующий элемент
};

```

Для задания длины строк мы определили символические константы с тем, чтобы программа легче читалась и можно было при необходимости легко их изменить, не перемещаясь по всему тексту.

Для работы с каждой очередью потребуется по два указателя, ссылающиеся на их начало и конец. Назовем их `beg_body` и `end_body` – для очереди на кузовные работы и `beg_engine` и `end_engine` – для очереди на ремонт двигателей.

При составлении этой программы мы так же, как и в предыдущей, разбили ее на независимые функции и передаем им всю информацию через параметры:

```

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <iomanip.h>
const int l_name = 20, l_model = 20, l_work = 80;
struct Order {
    char name[l_name], model[l_model], work[l_work];
    time_t time;
    float price;
    Order* next;
};
Order* add(Order* end, const Order& order);
Order* first(const Order& order);
Order* get(Order* beg);
int get_type();
Order input_Order();
int menu();
void print(const Order& order);
void print_dbase(Order* beg);
int read_dbase(char* filename, Order** beg, Order** end);
int write_dbase(char* filename, Order* pv);

// ----- Главная функция
int main(){
    Order* beg_body = 0, *end_body = 0;
    *beg_engine = 0, *end_engine = 0;
    char *file_body = "order_body.txt";
    *file_engine = "order_engine.txt";
    read_dbase(file_body, &beg_body, &end_body);
    read_dbase(file_engine, &beg_engine, &end_engine);
    Order* p = beg_body;
    while(p != 0) {
        cout << p->name << " " << p->model << " " << p->work << endl;
        p = p->next;
    }
    Order* p2 = beg_engine;
    while(p2 != 0) {
        cout << p2->name << " " << p2->model << " " << p2->work << endl;
        p2 = p2->next;
    }
}

```

```

read_dbase(file_engine, &beg_engine, &end_engine);
while (true) {
    switch (menu()) {
        case 1: // Добавление заявки (помещение в очередь)
            switch (get_type()) {
                case 1: if (beg_body) end_body = add(end_body,
                    input_Order());
                else {
                    beg_body = first(input_Order());
                    end_body = beg_body;
                }
                break;
            case 2: if (beg_engine) end_engine = add(end_engine,
                input_Order());
                else {
                    beg_engine = first(input_Order());
                    end_engine = beg_engine;
                }
                break;
            }
            break;
        case 2: // Распечатка квитанции (выборка из очереди)
            switch (get_type()) {
                case 1: beg_body = get(beg_body); break;
                case 2: beg_engine = get(beg_engine); break;
            }
            break;
        case 3: // Вывод заявок на экран
            switch (get_type()) {
                case 1: print_dbase(beg_body); break;
                case 2: print_dbase(beg_engine); break;
            }
            break;
        case 4: // Вывод заявок в файл
            write_dbase(file_body, beg_body);
            write_dbase(file_engine, beg_engine);
            break;
        case 5: // Выход
            return 0;
        default:
            cout << "Надо вводить число от 1 до 5" << endl;
            break;
    }
}
return 0;
}

```

Задача 9.3. Очередь

```

// ----- Добавление заявки
Order* add(Order* end, const Order& order) {
    Order* pv = new Order;
    *pv = order; // Копирование элемента
    pv->time = time();
    pv->next = 0;
    end->next = pv;
    end = pv;
    return end;
}

// ----- Начальное формирование очереди
Order* first(const Order& order) {
    Order* beg = new Order;
    *beg = order; // Копирование элемента
    beg->time = time();
    beg->next = 0;
    return beg;
}

// ----- Выборка из очереди
Order* get(Order* beg) {
    if (!beg) { cout << "Очередь пуста" << endl; return 0; }
    print (*beg);
    beg = beg->next;
    return beg;
}

// ----- Определение типа работ
int get_type() {
    int type;
    char buf[10];
    do {
        cout << "Введите вид работы: 1 - кузовные, 2 - двигатель" <<
            endl;
        cin >> buf;
        type = atoi(buf);
    }
    while (type != 1 && type != 2);
    cin.get();
    return type;
}

// ----- Ввод заявки
Order input_Order() {
    Order order; char buf[10];
    cout << "Введите фамилию И.О. " << endl;
    cin.getline(order.name, 1_name);
    cout << "Введите марку а/м " << endl;
    cin.getline(order.model, 1_model);
    cout << "Введите описание работ " << endl;
}

```

Задача 9.3. Очередь

```

while (pv) {
    print(*pv);
    pv = pv->next;
}
// ----- Чтение базы из файла
int read_dbase(char* filename, Order** beg, Order** end) {
    ifstream fin(filename, ios::in | ios::nocreate);
    if (!fin) { cout << "Нет файла " << filename << endl; return 1; }

    Order order;
    *beg = 0;
    while (fin.getline(order.name, 1_name)) {
        fin.getline(order.model, 1_model);
        fin.getline(order.work, 1_work);
        fin >> order.time >> order.price;
        fin.get();
        if (*beg) *end = add(*end, order);
        else   { *beg = first(order); *end = *beg; }
    }
    return 0;
}
// ----- Вывод базы в файл
int write_dbase(char* filename, Order* pv) {

```

```
ofstream fout(filename);
if (!fout) { cout << "Ошибка открытия файла" << endl; return 1; }

while (pv) {
    fout << pv->name << endl;
    fout << pv->model << endl;
    fout << pv->work << endl;
    fout << pv->time << ' ' << pv->price << endl;
    pv = pv->next;
}
return 0;
}
```

У этой программы много общего с предыдущей, поэтому обратим внимание на отличия.

Работу с очередью выполняют три функции. Функция `first` формирует первый элемент очереди и возвращает указатель на него. Функция `add` выполняет добавление в конец, поэтому ей передается указатель на конец очереди и элемент, который следует добавить, а возвращает она измененный указатель на конец очереди. Выборка выполняется из начала очереди, при этом указатель на ее начало изменя-

ется, поэтому функция `get` получает этот указатель через параметры и возвращает его новое значение.

Вспомним, что для структур определена операция присваивания, выполняющая позлементное копирование, поэтому при занесении в очередь нет необходимости копировать отдельно каждое поле (см. строки, помеченные комментарием «Копирование элемента», в функциях `add` и `first`).

В этой программе для расширения знаний о стандартной библиотеке функций продемонстрирована *работа с датами*. Средства, необходимые для этого, определены в заголовочном файле `<time.h>`. Текущую дату и время можно получить с помощью функции `time`. Она возвращает ее в виде количества секунд, прошедших с полуночи 1 января 1970 года. Функция `localtime` преобразует эту величину в стандартную структуру `tm`, определенную в том же заголовочном файле, и возвращает указатель на эту структуру. Поля структуры содержат все, что может потребоваться для работы с датами: год (точнее, количество лет, прошедших с 1900 года), месяц (от 0 до 11), день (от 1 до 31), час (от 0 до 23), минуту (0–59) и секунду (0–59). Ввод-вывод организован с помощью классов. Обратите внимание на использование метода `get()` класса `iostream` в функциях `get_type` и `read_dbase`. Он необходим для считывания из входного потока признака конца строки с тем, чтобы следующий ввод производился из новой строки. Для хранения заявок из каждой очереди используются отдельные файлы.

ПРИМЕЧАНИЕ

В функции `print`, предназначенной для вывода отдельной заявки, использован манипулятор `setiosflags`, с помощью которого устанавливается выравнивание по левому краю. Для этого в манипулятор передается в качестве параметра константа `left`, определенная в классе `ios` (базовом для всех классов ввода-вывода). С помощью этой константы устанавливается соответствующий флаг состояния потока. Это, так сказать, информация «для любознательных».

Задача 9.4. Бинарное дерево

Написать программу учета нарушений правил дорожного движения. Для каждой автомашины необходимо хранить в базе список нарушений. Для каждого нарушения фиксируется дата, время, вид нарушения и размер штрафа. При оплате всех штрафов автомашине удаляется из базы.

В первую очередь, как обычно, опишем внешнюю спецификацию нашей программы. Программа должна позволять вносить нарушение в базу, удалять его при поступлении сведений об оплате штрафа, получать справку о нарушениях, совершенных на заданной автомашине, сохранять информацию в файле и считывать ее из файла. Позаимствуем принцип организации меню из предыдущих программ. 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 автомашине с заданным номером;
- 4) выход.

Теперь займемся определением внутренней формы 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*. Однаковых номеров автомашин не бывает, поэтому логично использовать номера в качестве ключей. Кроме ключа, узел должен содержать две ссылки: на свое левое и правое поддерево. У всех узлов левого поддерева ключи меньше, чем ключ данного элемента, 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 элементов¹.

Информационная часть каждого узла должна содержать ссылку на список нарушений. Опишем элемент списка нарушений (штраф по-английски — «fine») и узел дерева поиска `Node`:

```
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 автомашины
    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 автомашины и выполняется поиск в базе. Если такая автомашина уже фигурирует в базе, в список ее нарушений добавляется новое, а если нет, то создаются новый узел дерева и первый элемент списка нарушений.

При вводе информации об уплате штрафа выполняется поиск заданной автомашины, а затем — поиск соответствующего нарушения. При этом запрашивается

¹ Сравните с линейным списком, в котором время поиска пропорционально количеству его элементов.

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 существующие данныечитываются из файла, а при окончании работы с программой файл обновляется.

Как и в предыдущих примерах, после описания интерфейса программы, используемых структур данных и общего алгоритма следует разбить ее на функции и определить интерфейс каждой из них.

Программа начинает работу с формирования двоичного дерева в динамической памяти из файла базы нарушений. Назовем функцию чтения `read_dbase`. Интерфейс функции прост: 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[1_number]; // Номер автомашины
    char time[1_time]; // Время нарушения
    char type[1_type]; // Вид нарушения
    float price; // Размер штрафа
};
```

Функция с незамысловатым именем `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 отличается: если для внесения данных в базу требуется вся информация (номер автомашины, время и вид нарушения, размер штрафа), то для удаления оплаченного штрафа необходимы только номер автомашины 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 этот режим 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 процедур 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 осуществлять спуск по дереву. Назовем функцию `search_insert`. Для того чтобы ее можно было использовать и при удалении узла из дерева, нам пришлось 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.

ВНИМАНИЕ!

В отличие от задач 9.2 и 9.3, где для передачи указателя по адресу соответствующий параметр функции объявлялся как указатель на указатель, здесь используется другой, более удобный, на наш взгляд, механизм: параметр функции объявляется как *ссылка на указатель*. При этом упрощается и обращение к функции (отпадает необходимость в операции взятия адреса &), и кодирование тела функции (не нужна операция разадресации).

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 единобразия 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 поддерева, 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 автомашине с заданным номером, служит простая функция `print_node` с соответствующим узлом дерева в качестве параметра:

```
void print_node(const Node* node);
```

По окончании работы с базой она выводится в файл, для этого служит 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 удается 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, а затем дадим необходимые пояснения по алгоритмам работы ее функций.

```
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <iomanip.h>

const char* filename = "dbase";
enum Action {INSERT, DEL, INFO};
enum Dir {LEFT, RIGHT};
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 автомашины
    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 автомашины
    char time[l_time]; // Время нарушения
    char type[l_type]; // Вид нарушения
    float price; // Размер штрафа
};
```

¹ Хотя, положа руку на сердце, мы сами не всегда следуем своим советам.

```
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(){
    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 << endl; return 1;
    }
    write_dbase(fout, root);
    return 0;

case 5: // Отладка
    print_dbase(root);
    break;
default:
    cout << " Надо вводить число от 1 до 4" << endl;
    break;
}
return 0;
}

// ----- Спуск по дереву
Node* descent(Node* p) {
    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;
}

// ----- Формирование корневого элемента дерева
Node* first(Data data) {
    // Создание записи о нарушении:
    Fine* beg = new Fine;
    strcpy(beg->time, data.time, l_time);
    strcpy(beg->type, data.type, l_type);
    beg->price = data.price;
    beg->next = 0;
    // Создание первого узла дерева:
    Node* root = new Node;
    strcpy(root->number, data.number, l_number);
    root->beg = beg;
    root->left = root->right = 0;
    return root;
}

```

```

// ----- Ввод нарушения
Data input(Action action) {
    Data data;
    char buf[10], templ[3], temp2[3];
    int day, month, hour, min;
    cout << "Введите номер а/м" << endl;
    cin.getline(data.number, l_number);
    if (action == INFO) return data;

    do {
        cout << "Введите дату нарушения в формате ДД.ММ.ГГ:" << endl;
        cin >> buf;
        strcpy(templ, buf, 2); strcpy(temp2, &buf[3], 2);
        day = atoi(templ); 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;
        strcpy(templ, buf, 2); strcpy(temp2, &buf[3], 2);
        hour = atoi(templ); 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;
}

// ----- Вывод меню
int menu() {
    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;
}

// ----- Вывод на экран сведений об а/н
void print_node(const Node& node) {
    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;
}

// ----- Вывод на экран всего дерева
void print_dbase(Node *p) {
    if (p) {
        print_node(*p);
        print_dbase (p->left);
        print_dbase (p->right);
    }
}

// ----- Чтение базы из файла
Node * read_dbase (char* filename) {
    Node *parent;
    Dir dir;
    Data data;
    ifstream f(filename, ios::in | ios::nocreate);
    if (!f) { cout << "Нет файла " << filename << endl; return 0; }

    f.getline(data.number, 1_number); // Номер а/м
}

```

```

if(f.eof()) { cout << "Пустой файл (0 байт)" << endl; return 0; }
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, 1_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, 1_time);
    if(data.time[0] == '-') return 1; // Конец списка нарушений
    f.getline(data.type, 1_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 следующему элементу списка
        }
    }
    if (!found) {
        cout << "Сведения о нарушении отсутствуют." << endl;
        return 1;
    }

    if (pf == p->beg) // Удаление из начала списка
        p->beg = pf->next;
    else // Удаление из середины или конца списка
        prev->next = pf->next;
    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;
    strcpy(pf->time, data.time, l_time);
    strcpy(pf->type, data.type, l_type);
    pf->price = data.price;
    pf->next = 0;

    if (!found) {
        // Создание нового узла:
        p = new Node;
    }
}

```

```

strcpy(p->number, data.number, l_number);
p->beg = pf;
p->left = p->right = 0;
if (dir == LEFT) // Присоединение к левому поддереву предка:
    parent->left = p;
else // Присоединение к правому поддереву предка:
    parent->right = p;
}
else { // Узел существует. добавление нарушения в список
    Fine* temp = p->beg;
    while (temp->next) temp = temp->next; // Поиск конца списка
    temp->next = pf;
}
return p;
}
// ----- Вывод базы в файл
void write_dbase(ofstream f, const Node *p) {
    if (p) {
        write_node(f, *p);
        write_dbase(f, p->left);
        write_dbase(f, p->right);
    }
}
// ----- Вывод в файл сведений об а/м
void write_node(ofstream f, const Node& node) {
    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 будет 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 одной автомашины 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 первой автомашины и ее первом нарушении и зано-

сится в корень дерева с помощью функции `first`, затем организуется цикл по всем остальным нарушениям данной автомашины, если они есть. Нарушения добавляются в список с помощью функции `search_insert`.

Далее аналогичные действия производятся для всех остальных автомашин. 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 на то, 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 удобства отладки была написана рекурсивная функция вывода всей базы `print_dbase`. Она аналогична функции вывода базы в файл.

Функция поиска с включением `search_insert` исчерпывающе, на наш взгляд, документирована в программе. Кроме того, описание аналогичной функции приведено в Учебнике на с. 125, поэтому мы не будем рассматривать ее подробно.

Лучше 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). Если узел имеет только одного потомка, то этот потомок становится на место удаляемого узла, а в остальном дерево не изменяется (рис. 9.2). 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 звучит не очень понятно, поэтому рассмотрите этот случай на рис. 9.3.

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 на рис. 9.4.

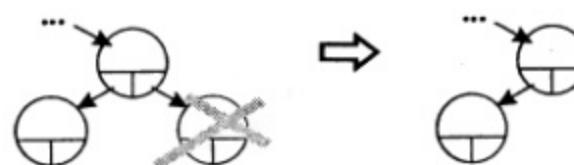


Рис. 9.1. Удаление узла, не имеющего потомков

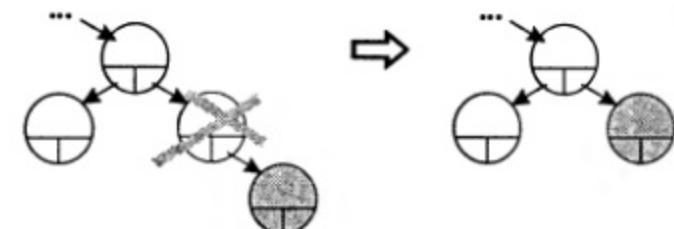


Рис. 9.2. Удаление узла с одним потомком

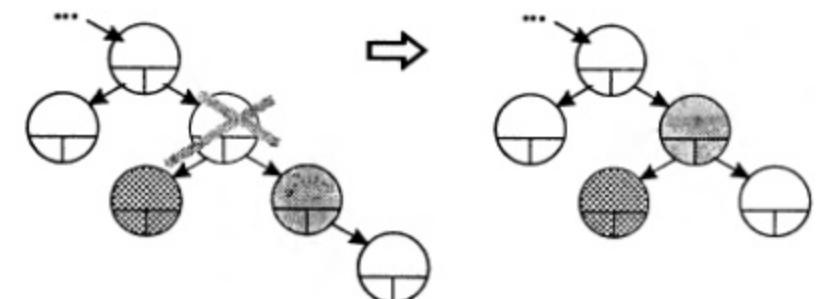


Рис. 9.3. Удаление узла с двумя потомками

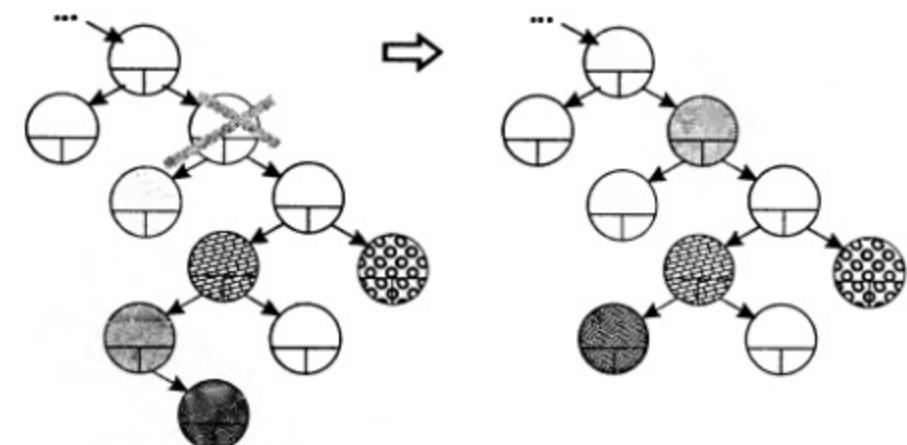


Рис. 9.4. Удаление узла (общий случай)

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 *r* нет левого поддерева, на его место будет поставлена вершина (возможно, пустая) его правого поддерева (оператор 11).

Иначе, если у узла *r* 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 *rgrev*, 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!

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 дерева, надо обновить указатель на корень (оператор 14), иначе — присоединить этот указатель к соответствующему поддереву предка удаляемого узла (оператор 15).

После того как узел удален из дерева, освобождается занимаемая им память (оператор 16).

ПРИМЕЧАНИЕ

Для дерева большой размерности глубина рекурсии может быть весьма значительной, что может привести к переполнению стека. Более безопасно, хотя и более сложно, реализовывать рекурсию самостоятельно с помощью структуры данных «стек».

Как вы могли убедиться, работа с динамическими структурами данных требует внимания и тщательности. Однако можно заметить, что в различных задачах изменяется только информационная часть компонент, а принципы работы со стеком, очередью или списком остаются неизменными, поэтому фактически функции для работы с каждой из этих структур пишутся и отлаживаются один раз. Разработчики языка тоже заметили этот факт и включили в стандартную библиотеку C++ так называемые *шаблоны*, реализующие основные динамические структуры данных. Во второй части практикума мы рассмотрим применение этих шаблонов. Для их понимания вам придется изучить многие средства и возможности C++, но предпринятые усилия быстро себя окупят.

Давайте повторим наиболее важные моменты этого семинара.

1. Динамическими структурами данных называются блоки в динамической памяти, связанные друг с другом с помощью указателей.

2. Динамические структуры различаются способами связи отдельных элементов и допустимыми операциями над ними.
3. Элемент любой динамической структуры данных состоит из информационных полей и полей указателей.
4. Наиболее распространенными структурами являются линейный список (односвязный или двусвязный), стек, очередь и бинарное дерево.
5. Для стека определены операции помещения элемента в вершину и выборки элемента из вершины.
6. Для очереди определены операции помещения элемента в конец очереди и выборки элемента из ее начала.
7. Допускается вставлять и удалять элементы в произвольное место линейного списка.
8. 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.
9. Каждый узел дерева характеризуется уникальным ключом.
10. Допускается вставлять и удалять элементы в произвольное место дерева.
11. 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.
12. Динамические структуры в некоторых случаях более эффективно реализовывать с помощью массивов (см. Учебник, с. 126).

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;
- 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 обо всех книгах в библиотеке 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 единственный выезд находятся в одном конце полосы. Если владелец автомашины приходит забрать свой автомобиль, который не является ближайшим к выходу, то все автомашины, 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 автомашины задается командной строкой, которая содержит признак прибытия или отъезда и номер машины. Программа должна выводить сообщение при прибытии или выезде любой машины. При выезде автомашины 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: следует создать список занятых участков и список свободных участков памяти на диске.

Вариант 10

В файловой системе каталог файлов организован в виде линейного списка.

Для каждого файла в каталоге содержатся следующие сведения:

- имя файла;
- дата создания;
- 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 при вводе.

Вариант 11

Предметный указатель организован в виде линейного списка.

Каждая компонента указателя содержит слово и номера страниц, на которых это слово встречается. Количество номеров страниц, относящихся к одному слову, лежит в диапазоне 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 при вводе.

Вариант 12

Текст помощи для некоторой программы организован в виде линейного списка.

Каждая компонента текста помочи 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 помочи;
- вывод текста помочи;
- вывод поясняющего текста для заданного термина.

Программа должна обеспечивать диалог с помощью меню и контроль ошибок при вводе.

Вариант 13

Картотека в бюро обмена квартир организована в виде линейного списка.

Сведения о каждой квартире включают:

- количество комнат;
- этаж;
- площадь;
- 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 при вводе.

Вариант 14

Англо-русский словарь построен в виде двоичного дерева.

Каждая компонента 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 компоненте 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 компонента с наибольшим зна-

чением счетчика обращений; б) найденная компонента 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 при вводе.

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; а) сколько женщин моложе 30 лет, имеющих среднее образование, ответили НЕТ на вопрос анкеты; а) 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 при вводе.

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 при вводе.

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 при вводе.

Вариант 19

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов дальнего следования.

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 при вводе.

Вариант 20

Автоматизированная информационная система на железнодорожном вокзале содержит сведения об отправлении поездов дальнего следования.

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 при вводе.

ПРИЛОЖЕНИЕ 1

Интегрированная среда Visual C++ 6.0

Integrated Development Environment (интегрированная среда разработки), или, сокращенно, *IDE* – это программный продукт, объединяющий текстовый редактор, компилятор, отладчик и справочную систему. Мы предполагаем, что пакет Microsoft Visual Studio 6.0, в состав которого входит IDE Microsoft Visual C++ 6.0, уже установлен на вашем компьютере. В приложении приводятся минимально необходимые сведения для начала работы с интегрированной средой. Более подробную информацию можно извлечь из справочной системы Visual Studio 6.0.

Любая программа, создаваемая в среде Visual C++, даже такая простая, как «Hello, World!», всегда оформляется как отдельный *проект* (*project*). Проект – это набор взаимосвязанных исходных файлов и, возможно, включаемых (заголовочных) файлов, компиляция и компоновка которых позволяет создать исполняемую программу. Однако, разработчики Visual Studio пошли еще дальше, стремясь удовлетворить потребности не только программистов-одиночек, но и больших коллективов разработчиков программных продуктов. Так появилось понятие *рабочей области проекта* (*project workspace*). Рабочая область может содержать любое количество различных проектов, сгруппированных вместе для согласованной разработки: от отдельного приложения до библиотеки функций или целого программного пакета. Очевидно, что для решения наших учебных задач каждая программа будет воплощаться в виде одного проекта, поэтому рабочая область проекта у нас всегда будет содержать ровно один проект.

Запуск IDE. Типы приложений

Вызов Visual C++ осуществляется или через меню Пуск ▶ Программы ▶ Microsoft Visual Studio 6.0 ▶ Microsoft Visual C++ 6.0¹, или щелчком мыши по пиктограмме с соответствующим именем, например VC6, – если вы позаботились о ее размещении на рабочем столе компьютера.

¹ На вашем компьютере путь к исполняемой команде меню может быть другим.

После запуска Visual C++ появляется главное окно программы, показанное на рис. П1.1. (В зависимости от настроек для вашего рабочего стола Visual C++ его вид может несколько отличаться от показанного на рисунке.)

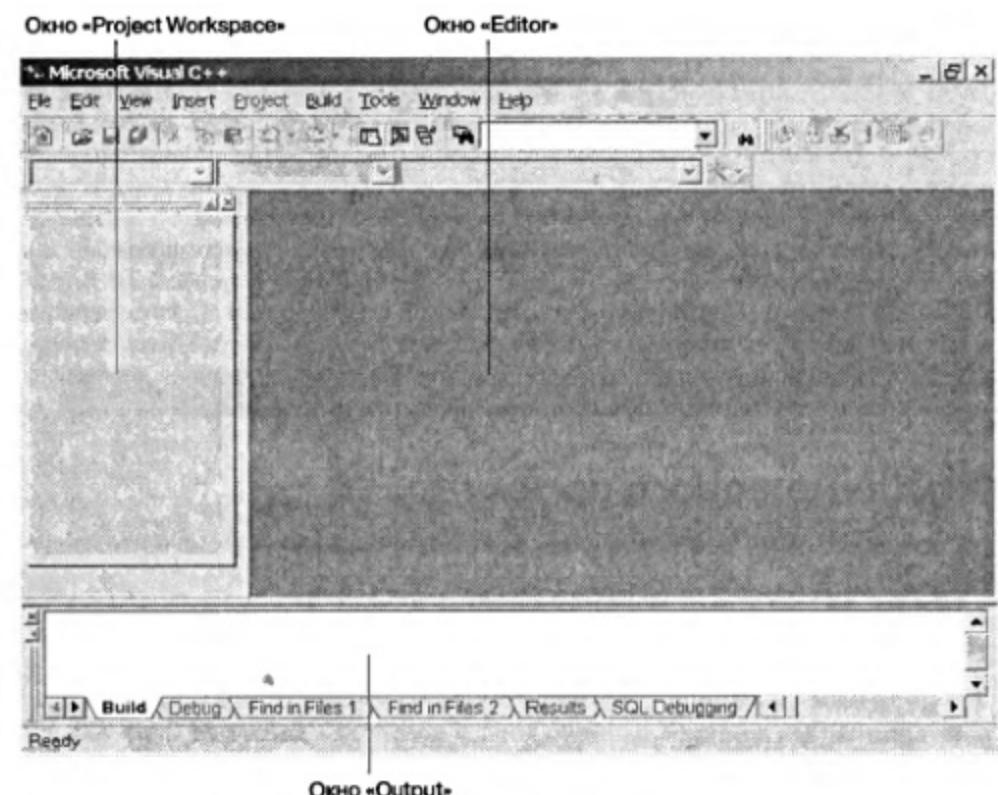


Рис. П1.1. Главное окно (рабочий стол) Visual C++

На самом деле главное окно (рабочий стол) Visual C++ принадлежит студии разработчика Microsoft Developer Studio – интегрированной среде, общей для Visual C++, Visual J, MS Fortran Power Station и некоторых других продуктов.

Рабочий стол Visual C++ включает в себя три окна:

Окно *Project Workspace* (окно рабочей области) предназначено для оказания помощи при написании и сопровождении больших многофайловых программ. Пока что (на рис. П1.1) оно закрыто, но после создания нового проекта (или загрузки сохраненного ранее проекта) одна из вкладок этого окна будет содержать список файлов проекта. Окно *Editor* (окно редактора) используется для ввода и проверки исходного кода.

Окно *Output* (окно вывода) служит для вывода сообщений о ходе компиляции, сборки и выполнения программы. В частности, сообщения о возникающих ошибках появляются именно в этом окне.

Под заголовком главного окна, как и во всех Windows-приложениях, находится строка меню. Назначение команд меню и кнопок панелей инструментов мы будем рассматривать по мере необходимости, разбирая основные приемы работы в IDE. Пока же только заметим, что для кнопок панелей инструментов предусмотрена удобная контекстная помощь: если вы наведете курсор мыши на кнопку и задержитесь на секунду-другую, то всплывет подсказка с назначением кнопки.

Developer Studio позволяет строить проекты разных типов, ориентированные на различные сферы применения. Так как эта студия спроектирована на Windows-платформе, то почти все типы проектов являются оконными Windows-приложениями с соответствующим графическим интерфейсом. В то же время разработчики Developer Studio предусмотрели работу и с так называемыми *консольными приложениями*. 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 С/C++, так как компилируемые программы не «покрываются» толстым слоем промежуточного Windows-кода.

Создание нового проекта

Для создания нового проекта типа «консольное приложение» выполните следующие действия:

- Выберите в строке меню главного окна команду **File > New...**
- В открывшемся диалоговом окне **New** выберите вкладку **Projects**:
 - выберите тип **Win32 Console Application**;
 - введите имя проекта в текстовом поле **Project Name**, например *First*;
 - введите имя каталога размещения файлов проекта в текстовом поле **Location**¹ (если указанный вами каталог отсутствует, то он будет создан автоматически);
 - щелкните левой кнопкой мыши на кнопке **OK**.
- После щелчка запускается так называемый *мастер приложений* Application Wizard, который открывает диалоговое окно Win32 Console Application — Step1 of 1 с предложением определиться, какой подтип консольного приложения вам требуется создать:
 - выберите тип **An empty project**;
 - щелкните на кнопке **Finish**.
- После щелчка появится окно **New Project Information** со спецификациями проекта и информацией о каталоге, в котором будет размещен создаваемый проект:
 - щелкните на кнопке **OK**.

Допустим, что в качестве **Project Name** вы ввели имя *First*. Тогда после выполненных шагов вы увидите на экране примерно то, что показано на рис. П1.2.

¹ Рекомендуется для размещения проектов выделить специальную папку, например D:\MyProjects.

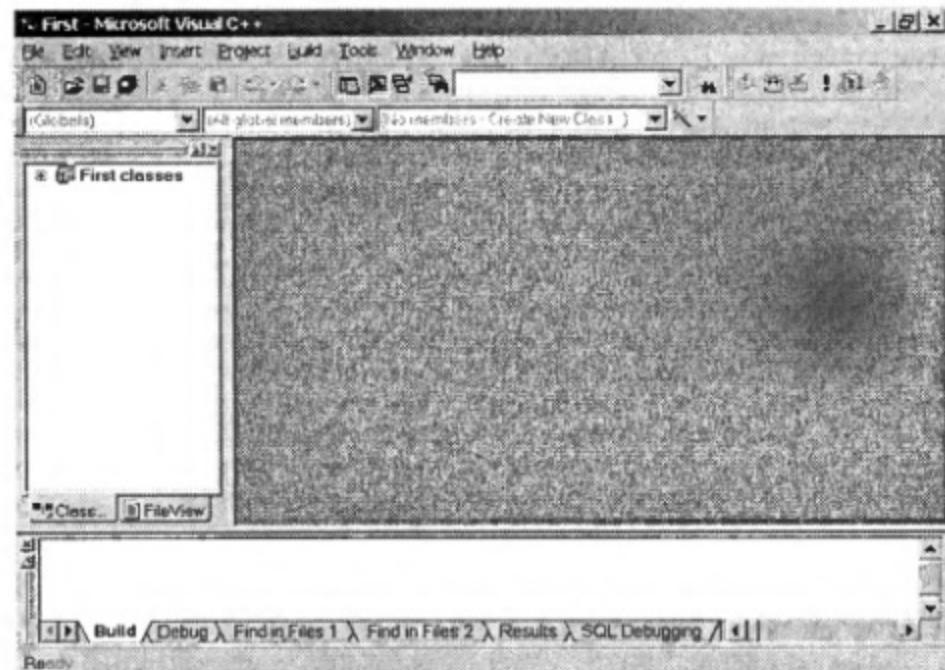


Рис. П1.2. Рабочий стол Visual C++ после создания проекта First

Прежде чем продолжать работу, свернем временно главное окно Visual C++ на панель задач и заглянем в папку *First*, созданную мастером приложений для нашего проекта, а точнее – для нашей рабочей области. Там мы найдем файлы *First.dsw*, *First.dsp*, *First.opt*, *First.ncb*, а также папку *Debug* (или *Release* – в зависимости от конфигурации проекта). Дадим краткое описание каждого из файлов:

- *First.dsw* – файл рабочей области проекта, используемый внутри интегрированной среды разработки. Он объединяет всю информацию о проектах, входящих в данную рабочую область.
- *First.dsp* – проектный файл, используемый для построения (building) отдельного проекта или подпроекта (в ранних версиях Visual C++ этот файл имел расширение *.mak*).
- *First.opt* – файл, содержащий опции рабочей области проекта. Благодаря этому файлу при каждом открытии рабочей области проекта все параметры Developer Studio, выбранные во время последнего сеанса работы с данной рабочей областью, будут восстановлены.
- *First.ncb* – служебный файл. Он создается компилятором и содержит информацию, которая используется в инструменте интегрированной среды под названием *ClassView*. Панель *ClassView* находится в окне *Project Workspace* и показывает все классы C++, для которых доступны определения в рамках данного проекта, а также все элементы этих классов.¹

¹ В задачах первой книги Практикума эта панель будет всегда пустой, так как классы C++ здесь не используются.

- Debug — папка, в которую будут помещаться файлы, формируемые компилятором и сборщиком. Из этих файлов нас будет интересовать, в общем-то, только один — исполняемый файл, имеющий расширение .exe.

Развернем главное окно Visual C++ с открытой рабочей областью First (рис. П1.2), чтобы продолжить работу по созданию нашей первой программы. Первое, что бросается в глаза — окно Project Workspace «оживилось», в нем появились две вкладки: Class View и File View. С вкладкой Class View в ближайшее время мы работать не будем, поэтому щелчком мыши переключимся на вкладку File View. Она предназначена для просмотра списка файлов проекта. Откроем список First files, щелкнув мышью на значке <+. Появится дерево списка файлов, содержащее пиктограммы трех папок: Source Files, Header Files, Resource Files. Так как в консольных приложениях файлы ресурсов не используются, то про последнюю папку сразу забудем. Попробуем заглянуть (щелчком мыши) в первые две папки. Попытка окажется неудачной — папки пусты. Это и неудивительно: ведь мы выбрали в качестве подтипа консольного приложения пустой проект — опцию An empty project. Так что наполнение проекта конкретным содержанием у нас еще впереди.

Добавление к проекту файлов с исходным кодом

Рассмотрим две ситуации: а) добавление существующего файла, б) создание нового файла.

Добавление существующего файла

В этом случае файл с исходным кодом (пусть это будет файл first.cpp) вы уже подготовили ранее в каком-то текстовом редакторе, скачали по сети или принесли от друга на дискете. Продолжение должно быть следующим:

- Скопируйте исходный файл (first.cpp) в папку рабочей области проекта (в данном случае — First).
- Вернитесь к списку First files в окне Project Workspace вашего проекта и щелкните правой кнопкой мыши на папке Source Files.
- В появившемся контекстном меню щелчком мыши выберите команду добавления файлов Add Files to Folder....
- В открывшемся диалоговом окне Insert Files... выберите нужный файл (first.cpp) и щелкните на кнопке OK.

Добавление нового файла

В этом случае необходимо выполнить следующие действия:

- Выберите в строке меню главного окна команду File > New.... В результате откроется диалоговое окно New.
- На вкладке Files:
 - выберите тип файла (в данном случае: C++ Source File);
 - в текстовом поле File Name введите нужное имя файла (в данном случае: first.cpp);

- флагок Add to project должен быть включен;
- щелкните на кнопке OK.

После предпринятых шагов можно наблюдать следующие результаты:

- 1) в окне Project Workspace папка Source Files списка файлов проекта раскроется, и в нее будет помещен файл first.cpp;
- 2) окно Editor засветится мягким белым светом, а в левом верхнем углу его замерцает черный, как графитовый стержень, текстовый курсор, ненавязчиво предлагая ввести какой-нибудь текст.

Что мы и сделаем. Введите, например, такой текст:

```
#include <iostream.h>
int main()
{
    char str1[80];
    cout << "Welcome to C++ !" << endl;
    cout << "Enter the string:" << endl;
    cin >> str1;
    cout << "The value str1 = " << endl;
    cout << str1 << endl;
    return 0;
}
```

На рис. П1.3 показан вид главного окна Visual C++ после завершения проделанной работы.

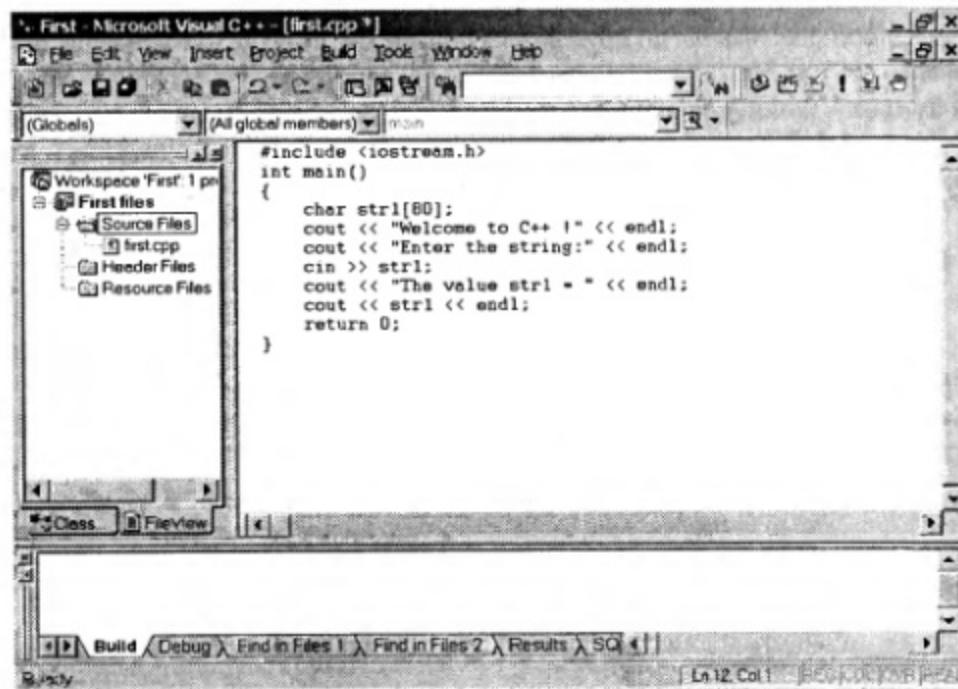


Рис. П1.3. Вид главного окна после ввода исходного текста в файл first.cpp

Многофайловые проекты

Никаких особых усилий при создании многофайловых проектов вам прилагать не придется: разработчики Developer Studio уже обо всем позаботились. Надо будет просто несколько раз повторить процедуру создания/добавления исходных файлов, описанную выше. В многофайловых проектах обычно присутствуют и заголовочные файлы — они создаются/добавляются после щелчка правой кнопкой мыши на пиктограмме папки Header Files в окне Project Workspace; при этом на вкладке Files диалогового окна New выбирается тип файла C/C++ Header File, а при вводе имени файла используется расширение .h.

ПРИМЕЧАНИЕ

Папки Source Files и Header Files, пиктограммы которых вы видите в окне Project Workspace, на самом деле физически не существуют, то есть все файлы помещаются в основную папку рабочей области проекта, имя которой было задано при создании проекта в окне *Project Name*. Но, согласитесь, такое упорядочение дерева списка файлов в окне Project Workspace очень удобно.

Компиляция, компоновка и выполнение проекта

Эти операции могут быть выполнены или через меню Build главного окна, или с помощью кнопок панели инструментов. Опишем кратко основные команды меню Build:

- Compile** — компиляция выбранного файла. Результаты компиляции выводятся в окно Output.
- Build** — компоновка проекта. Компилируются все файлы, в которых произошли изменения с момента последней компоновки. После компиляции происходит сборка (link) всех объектных модулей, включая библиотечные, в результирующий исполняемый файл. Сообщения об ошибках компоновки выводятся в окно Output. Если обе фазы компоновки завершились без ошибок, то созданный исполняемый файл с расширением .exe может быть запущен на выполнение.
- Rebuild All** — то же, что и Build, но компилируются все файлы проекта независимо от того, были ли в них произведены изменения.

СОВЕТ

Если при компоновке многофайлового проекта посредством команды Build вы получаете сообщения об ошибках компиляции или сборки, которые вы не можете объяснить, то мы настоятельно рекомендуем вам воспользоваться командой Rebuild All.

- Execute** — выполнение исполняемого файла, созданного в результате компоновки проекта.

Операции Compile, Build и Execute удобнее выполнять через соответствующие кнопки панели инструментов Build MiniBar, которая в увеличенном виде показана на рис. П1.4.

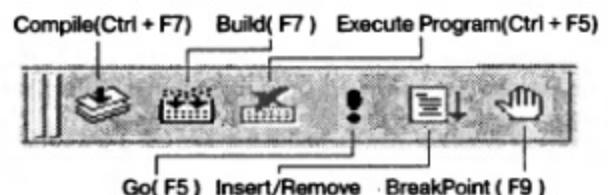


Рис. П1.4. Панель инструментов Build MiniBar

Чтобы пощелкать по этим кнопкам, вернемся к нашему проекту First.

Откомпилируйте проект, щелкнув на кнопке Build (F7). Диагностические сообщения компилятора и сборщика отображаются в окне вывода Output. Мы надеемся, что у вас все будет в порядке¹ и последняя строка в окне вывода будет выглядеть так:

First.exe - 0 error(s). 0 warning(s)

Теперь запустите приложение на выполнение, щелкнув на кнопке Execute Program (Ctrl+F5).

Появится окно приложения First, изображенное на рис. П1.5.



Рис. П1.5. Консольное окно приложения First.exe

На приглашение ввести строку введите любую строку, например Hello. World!, и нажмите Enter. Программа должна вывести на экран то, что показано на рис. П1.6.

Мы не будем здесь разбираться, почему программа вывела «обрезанную» строку Hello.. Для этого есть Учебник (см. с. 268), кроме того, проблема ввода-вывода строк рассматривается на семинаре 5.

Попробуйте заменить в программе оператор вывода

```
cout << "Welcome to C++ !" << endl;
```

следующим оператором:

```
cout << "Добро пожаловать в C++ !" << endl;
```

¹ В противном случае тщательно сличите набранный вами текст с приведенным в книге.

```
D:\MyProjects\First\Debug\FirstExe
Welcome to C++ !
Enter the string:
Hello, world!
The value str1 =
Hello,
Press any key to continue...
```

Рис. П1.6. Вид окна после ввода строки

Откомпилируйте (точнее говоря, скомпонуйте) программу и запустите ее на выполнение. Результат будет настолько безобразным, что рисунок-копию экрана мы здесь приводить не станем.

Проблемы с вводом-выводом кириллицы

Работа в среде Visual C++ 6.0 (в режиме консольных приложений) сопряжена с определенными неудобствами, вызванными различными стандартами кодировки символов кириллицы в операционных системах MS DOS и Windows. Этот вопрос подробно рассматривается на семинаре 1. Здесь мы только напомним, что весь ввод-вывод в консольном окне идет в кодировке стандарта ASCII, а текст в исходных файлах, набираемый в текстовом редакторе Visual C++, имеет кодировку в стандарте ANSI. Поэтому для нормального вывода строки, содержащей буквы русского алфавита, эту строку надо сначала «пропустить» через функцию `CharToOem()`, а уже потом отправлять на консольный вывод. Аналогично, если в программе есть консольный ввод текста и этот текст в дальнейшем надо сохранять в документах (файлах) с кодировкой ANSI, то перед сохранением его надо «пропустить» через функцию `OemToChar()`.

С учетом сказанного выше можно предложить следующую адаптацию нашей первой программы к местным (российским) условиям:

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

char* Rus(const char* text);

int main(){
    char str1[80];
    // cout << "Welcome to C++ !" << endl;
    cout << Rus("Добро пожаловать в C++ !") << endl;
    // cout << "Enter the string:" << endl;
    cout << Rus("Введите строку:") << endl;
    cin >> str1;
```

Как открыть проект, над которым вы работали ранее

```
// cout << "The value str1 = " << endl;
cout << Rus("Значение строки str1 = ") << endl;
cout << str1 << endl;
return 0;
}
////////////////////////////////////////////////////////////////////////
char bufRus[256];
char* Rus(const char* text) {
    CharToOem(text, bufRus);
    return bufRus;
}
////////////////////////////////////////////////////////////////////////
```

Ничего не принимайте на веру. Немедленно, прямо сейчас (!) откомпилируйте эту программу и убедитесь, что она дает ожидаемый результат.

Конфигурация проекта

Visual C++ позволяет строить проект либо в *отладочной конфигурации* (Win32 Debug), либо в *выпускной конфигурации* (Win32 Release). Мы рекомендуем вам всегда работать с проектами в отладочной конфигурации. Обычно она установлена по умолчанию. Все же не мешает проверить, с какой конфигурацией на самом деле идет работа. Для этого выберите в меню *Project* пункт *Settings...* Откроется диалоговое окно *Project Settings*. Проверьте, какое значение установлено в окне комбинированного списка *Settings For...*. Если это не Win32 Debug, то переключитесь на нужное значение через команду меню *Build > Set Active Configuration...*

Как закончить работу над проектом

Можно выбрать меню *File*, пункт *Close Workspace*. А можно просто закрыть приложение Visual C++.

Как открыть проект, над которым вы работали ранее

1. Способ первый:
 - Запустите на выполнение Visual C++.
 - Выберите меню *File*, пункт *Open Workspace...*
 - В открывшемся диалоговом окне найдите папку с вашим проектом, а в ней — файл *ProjectName.dsw*.
 - Откройте этот файл, щелкнув по нему мышью.
2. Способ второй:
 - Запустите на выполнение Visual C++.

- Выберите меню File, наведите курсор мыши на пункт Recent Workspaces.
 - Если в появившемся меню со списком последних файлов, с которыми шла работа, вы найдете интересующий вас файл ProjectName.dsw, то щелкните по нему мышью.
3. Способ третий:
- Не вызывая Visual C++, найдите папку с вашим проектом, а в ней – файл ProjectName.dsw.
 - Щелкните мышью на файле ProjectName.dsw.

Встроенная справочная система

В IDE Visual C++ имеется обширная справочная система, доступная через меню Help главного окна¹.

Кроме этого, очень удобно пользоваться интерактивной справкой: если вы находитесь в окне Editor, поставьте текстовый курсор на интересующий вас оператор или библиотечную функцию C++ и нажмите клавишу F1. Тотчас будет вызвана справочная система MSDN с предоставлением необходимой информации. Если запрошенный термин встречается в разных разделах MSDN, то сначала появится диалоговое окно «Найденные разделы». В списке разделов выберите тот, в котором упоминается «Visual C++».

Работа с отладчиком

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 Visual C++. Проще всего это сделать, написав программу, заведомо содержащую несколько ошибок, а затем показав, как с помощью отладчика можно найти и исправить эти ошибки.

В частности, мы научимся устанавливать в программе *точки прерывания* и выполнять ее до заданной точки. Когда во время выполнения встречается точка прерывания, программа останавливается, а на экране появляется отлаживаемый код. Это дает возможность детально выяснить, что происходит в программе.

Кроме того, программу можно выполнять последовательно, строку за строкой – такой процесс называется *пошаговым выполнением*. Этот режим позволяет следить за тем, как изменяются значения различных переменных. Иногда он помогает понять, в чем заключается проблема: если обнаруживается, что переменная принимает неожиданное значение, то это может послужить отправной точкой для выявления ошибки. После обнаружения ошибки ее можно исправить и выполнить программу заново в отладочном режиме.

Назовем нашу программу (проект) именем Buggy. Программа должна вычислять среднее арифметическое первых пяти натуральных чисел: 1–5. Нетрудно догадаться,

¹ При условии установки Visual Studio в полном объеме, включая справочную систему MSDN.

ся, что ответ должен быть равен 3, однако из-за специально сделанных ошибок программа первоначально будет выдавать неправильный ответ¹. Для создания программы выполните следующие действия:

- Создайте проект типа «консольное приложение» с именем Buggy.
- Добавьте к проекту файл buggy.cpp и заполните его следующим текстом:

```
#include <iostream.h>
int main()
{
    const N = 5;
    int a[N] = {1, 2, 3, 4, 5};
    float sum, average;
    int i;
    for (i = 1; i < N; i++)
        sum += a[i];
    average = sum / N;
    cout << "average = " << average << endl;
    return 0;
}
```

- Откомпилируйте проект.
- Запустите программу на выполнение.

Вы должны увидеть в консольном окне приложения нечто вроде следующего результата:

average = -2.14748e+007.

т. е. программа вычислила, что среднее арифметическое первых пяти целых чисел равно -21474800 (на вашем компьютере может быть и другое число), а это мало похоже на число 3.0.

Начнем отладку нашей злополучной программы.

Установка точки прерывания

Точка прерывания позволяет остановить выполнение программы перед любой выполняемой инструкцией (оператором) с тем, чтобы продолжать выполнение программы либо в пошаговом режиме, либо в непрерывном режиме до следующей точки прерывания.

Чтобы задать точку прерывания перед некоторым оператором, необходимо установить перед ним текстовый курсор и нажать клавишу F9 или щелкнуть мышью на кнопке Insert/Remove BreakPoint на панели инструментов Build MiniBar. Точка прерывания обозначается в виде коричневого² кружка на левом поле окна редактирования. Повторный щелчок на указанной кнопке снимает точку прерывания. В программе может быть несколько точек прерывания.

¹ Пожалуйста, закройте глаза на ошибки, если вы сразу увидите их в тексте программы. Приводимый пример несложен и предназначен только для изучения возможностей отладчика.

² На вашем компьютере цвет кружка может быть несколько иным.

Выполнение программы до точки прерывания

Программа запускается в отладочном режиме с помощью команды **Build > Start Debug > Go** (или нажатием клавиши **F5**).

В результате код программы выполняется до строки, на которой установлена точка прерывания. Затем программа останавливается и отображает в окне **Editor** ту часть кода, где находится точка прерывания, причем желтая стрелка на левом поле указывает на строку, которая будет выполняться на следующем шаге отладки.

Продолжим демонстрацию описываемых средств на примере программы **Buggy**:

- Установите точку прерывания перед оператором **for**.
- Запустите программу в отладочном режиме, нажав клавишу **F5**.

Обратите внимание — в Visual C++ меню **Build** заменилось на меню **Debug**. Заглянем в него из любопытства (рис. П1.7).

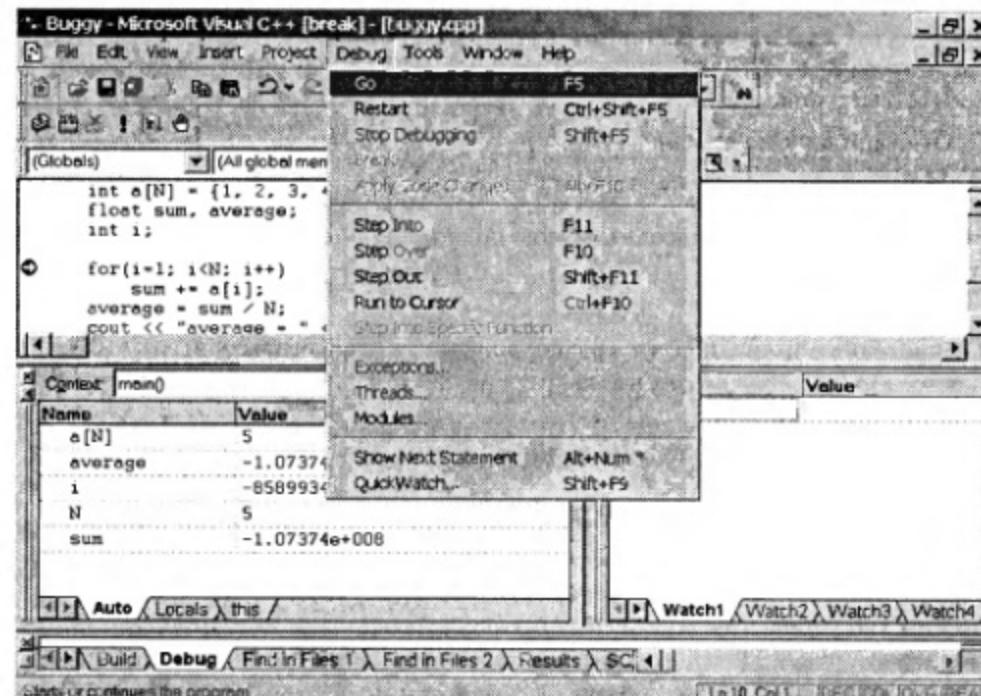


Рис. П1.7. Меню **Debug** в отладочном режиме

Среди различных команд этого меню особый интерес представляют команды **Step Into** (**F11**), **Step Over** (**F10**), **Step Out** (**Shift+F11**), **Run To Cursor** (**Ctrl+F10**) и **Stop Debugging** (**Shift+F5**).

Выбор последней команды (или нажатие комбинации клавиш **Shift+F5**) вызывает завершение работы с отладчиком.

Пошаговое выполнение программы

Нажимая клавишу **F10**, можно выполнять один оператор программы за другим.

Предположим, что при пошаговом выполнении программы вы дошли до строки, в которой вызывается некоторая функция **func1()**. Если вы хотите пройти через код вызываемой функции, то надо нажать клавишу **F11**. Если же внутренняя работа функции вас не интересует, а интересен только результат ее выполнения, то надо нажать клавишу **F10**.

Допустим, что вы вошли в код функции **func1()**, нажав клавишу **F11**, но через несколько строк решили выйти из него, т. е. продолжить отладку после возврата из функции. В этом случае надо нажать клавиши **Shift+F11**.

Существует и другая возможность пропустить пошаговое выполнение некоторого куска программы: установите текстовый курсор в нужное место программы и нажмите клавиши **Ctrl+F10**.

Продолжим отладку программы **Buggy**:

- Нажмите клавишу **F10**.

Указатель следующей выполняемой команды переместится на оператор **sum += a[i];**

Проверка значений переменных во время выполнения программы

Чтобы узнать значение переменной **sum**, в которой будет накапливаться сумма элементов массива **a**, задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением этой переменной:

sum = -1.07374e+008

(или с другим произвольным значением).

Ага!!! Ведь еще не было никакого суммирования и, следовательно, переменная **sum**, по идеи, должна быть равна нулю. Вот где собака зарыта! Мы забыли обнулить переменную **sum** до входа в цикл.

- Нажмите комбинацию клавиш **Shift+F5**, чтобы выйти из отладчика и исправить найденную ошибку. Найдите строку с объявлением

float sum, average;

и добавьте в нее инициализацию переменной **sum**:

float sum = 0, average;

- Откомпилируйте заново проект — кнопка **Build (F7)**.

- Запустите на выполнение — кнопка **Execute Program (Ctrl+F5)**.

Вы получите новый результат:

average = 2.8

Это как бы теплее, но все равно еще неправильно. Нажмите любую клавишу для завершения работы приложения. Необходимо продолжить отладку:

- Установите точку прерывания перед оператором `for`.
- Запустите программу в отладочном режиме, нажав клавишу `F5`.
- Нажмите клавишу `F10`.

Указатель следующей выполняемой команды переместится на оператор

```
sum += a[i];
```

- Нажмите клавишу `F10`.

Указатель следующей выполняемой команды переместится на оператор

```
for (i = 1; i < N; i++)
```

- Задержите теперь над переменной `sum` указатель мыши. Рядом с именем переменной на экране появится подсказка со значением этой переменной
`sum = 2.`

Стоп!!! Позади 1-я итерация цикла, и в переменной `sum` должно находиться значение первого элемента массива `a`, т. е. число 1. А мы имеем число 2, то есть значение второго элемента массива `a`. Пришло время вспомнить, что в языке C++ нумерация элементов массива начинается с 0! Поэтому ошибка находится в заголовке цикла:

```
for (i = 1; i < N; i++)
```

и ее нужно немедленно исправить. Повторите действия, направленные на исправление ошибки:

- выйти из отладчика,
- исправить текст программы в операторе `for`:
`for (i = 0; i < N; i++)`
- откомпилировать,
- запустить на выполнение.

Если в процессе исправления вы не внесли новых ошибок, то должен получиться результат:

```
average = 3.0.
```

Итак, все OK! – программа работает правильно.

В заключение отметим, что отладчик предоставляет и другие возможности для наблюдения за значениями переменных во время выполнения программы.

Окна Auto и Watch1

Помимо экранной подсказки, переменная `sum` со своим значением отображается в окне `Auto`, расположенному в левом нижнем углу экрана (см. рис. П1.7). В этом окне приведены значения последних переменных, с которыми работал `Visual C++`.

Кроме этого, в окне `Watch1`, которое находится в правом нижнем углу, можно задать имя любой переменной, за значениями которой вы хотите наблюдать.

Более подробно о работе с этими окнами можно почитать в справочной системе через меню `Help` главного окна `Visual C++`. То, что она написана на английском языке, ни в коей мере не должно вас останавливать!

ПРИЛОЖЕНИЕ 2

Интегрированная среда Borland C++ 3.1

Установка интегрированной среды `Borland C++ 3.1` с дистрибутива достаточно проста, но вряд ли этот дистрибутив доступен кому-то из читателей «за давностью лет». К счастью, в то золотое время для установки у себя на машине программного продукта часто достаточно было найти уже установленную работающую версию, заархивировать ее, скопировать на дискеты, переписать к себе на компьютер и разархивировать. Предполагаем, что читатель справится с этими действиями самостоятельно или с помощью коллег. Есть еще один вариант: подключить к вашему компьютеру винчестер, одолженный у друзей, на котором уже установлен `BC`, и скопировать необходимые каталоги на ваш винчестер.

Запуск IDE

Для запуска IDE `Borland C++ 3.1` надо открыть каталог (папку), в котором расположен файл `bc.exe` (обычно это `...\\BORLANDC\\BIN`), затем запустить этот файл на исполнение либо щелчком левой кнопки мыши, либо нажатием клавиши `Enter` в оболочке типа `Far` или `Norton Commander`.

После запуска появляется рабочий экран `BC`, содержащий четыре основных части:

- строка меню,
- окно редактирования,
- окно сообщений,
- строка состояния.

Строка меню предоставляет доступ к командам интегрированной среды. Для активизации строки меню нужно нажать клавишу `F10`, после чего один из элементов меню выделяется подсвеченным курсором. Перемещение курсора для выбора нуж-

ногого элемента меню осуществляется с помощью клавиш со стрелками. После выбора и нажатия клавиши **Enter** появляется либо выпадающее меню со списком команд, либо окно диалога, соответствующее выбранному элементу.

Окно редактирования предназначено для ввода и редактирования текста в одном из исходных (source) файлов программы. Система ВС позволяет одновременно держать в памяти несколько открытых окон, при этом активным является только одно окно, на которое установлен так называемый фокус ввода. Каждое окно имеет **рамку**, в верхней части которой расположены **заголовок окна** (имя файла, возможно с указанием пути к нему) и **номер окна** (окна нумеруются с 1 по 9, в эту нумерацию включается также окно сообщений (Message)). Переключение фокуса ввода с одного окна на другое осуществляется нажатием клавиши **F6**.

Приведем краткое описание назначения каждого элемента меню:

- ? – системное меню;
- File – загрузка и создание файлов, сохранение внесенных в программу изменений, выход из системы;
- Edit – реализация различных режимов редактирования текста в активном окне;
- Search – поиск фрагментов текста, контекстная замена и другие операции;
- Run – компиляция, компоновка и запуск программы на выполнение¹;
- Compile – компиляция программы²;
- Debug – управление возможностями отладчика;
- Project – организация проектов (многофайловых программ);
- Options – установка параметров компиляции, компоновки и других настроек интегрированной среды;
- Window – управление окнами;
- Help – обращение к системе оперативной подсказки.

Работа с меню

Опишем некоторые команды меню, достаточные для начала работы с интегрированной средой. Более полную информацию вы можете получить через меню **Help**.

Меню File

После выбора элемента меню **File** и нажатия клавиши **Enter** на экране появляется выпадающее меню, содержащее группу команд. Рассмотрим основные команды из этой группы.

- Команда **File > New** открывает новое окно редактирования со стандартным именем **NONAMExx.CPP** (где вместо букв **xx** используется целое число в диапазоне

¹ Команда **Run** из меню **Run** относится либо к активному окну – в случае однофайловой программы, либо ко всему проекту в целом – в случае многофайловых программ, имеющих в своем составе проектный файл.

² То же – для команды **Compile** из меню **Compile**.

от 00 до 99). Это имя считается временным (на время ввода нового текста). Если вы попытаетесь сохранить набранный текст с помощью команды **File > Save**, то система ВС вызовет диалоговое окно «Save File As», в котором пользователю предлагается ввести нужное имя файла. Если же на это предложение вы бездумно нажмете **Enter** (не изменяя имя файла), то файл с именем вида **NONAME00.CPP** будет сохранен в каталоге **BORLANDC\BIN**. Указанный каталог вообще-то предназначен для хранения исполняемых файлов и динамически подключаемых библиотек системы ВС, поэтому засорение его какими-либо не относящимися к делу файлами крайне нежелательно. Отсюда первая практическая рекомендация начинающему программисту:

СОВЕТ

Заведите специальную папку (каталог) для размещения ваших программ (проектов), создаваемых в среде ВС, например, **D:\BC_WORK**, а уже внутри этой папки вы будете создавать отдельные каталоги для каждой новой программы (проекта). О создании такого каталога надо позаботиться еще до начала работы над новой программой. В него вы будете помещать файлы с исходными текстами программы.

После завершения ввода текста в новый файл вызовите команду **File > SaveAs**.

- Команда **File > SaveAs** вызывает окно диалога «Save File As». В этом окне выделим следующие элементы:
 - текстовое поле **Save File As**, предназначенное для ввода имени файла;
 - поле списка **Files**, содержащий список файлов для текущего каталога; в этом поле возможна навигация по списку файлов и каталогов с помощью клавиш со стрелками и клавиши **Enter** – аналогично тому, как это делается в оболочках типа **Far** или **Norton Commander**, причем строка с символами ..\ обозначает переход в каталог верхнего уровня (родительский каталог);
 - строка состояния (внизу диалогового окна), в которой отображается полный путь к текущему каталогу и текущее имя файла;
 - три кнопки: **OK**, **Cancel** и **Help**.

Заметим, что переход от одного из перечисленных элементов диалогового окна к другому осуществляется нажатием клавиши **Tab**.

После того как вы установили нужный каталог путем навигации в поле **Files**, перейдите, используя клавишу **Tab**, в текстовое поле **Save File As** и введите требуемое имя файла (не забывайте контролировать это по строке состояния). Осталось перейти на кнопку **OK** и нажать клавишу **Enter**.

- Команда **File > Save (F2)**¹ в зависимости от состояния файла выполняется одним из двух способов:
 - либо сохраняет на диске текущее состояние файла (после тех изменений, которые вы в нем сделали) – если это файл «не первой свежести», то есть вы ранее с ним работали;

¹ В скобках после обозначения команды меню указывается так называемая горячая клавиша (или сочетание клавиш), нажатие которой эквивалентно данной команде.

- либо вызывает окно диалога «Save File As» — если файл абсолютно новый, то есть создан командой **File > New**.
- Команда **File > Open** вызывает окно диалога «Open a File», которое по составу элементов имеет много общего с окном «Save File As». Основную работу по поиску нужного файла вы проводите в поле **Files**, затем переходите на кнопку **Open** и нажимаете клавишу **Enter**. В результате текст файла появляется в окне редактирования. После завершения ввода/редактирования текста в файле вызовите команду **File > Save**.
- Команда **File > Quit** вызывает завершение работы с IDE BC. Выйти из системы можно и другим способом — нажав комбинацию клавиш **Alt+X**.

Меню Edit

Меню **Edit** позволяет выполнять вырезание, копирование и вставку выделенных фрагментов текста, с которым вы работаете в окне редактирования.

Выделение фрагмента осуществляется разными способами. Самый простой — с применением 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, то, удерживая нажатой клавишу **Shift**, можно с помощью клавиши **↓** («стрелка вниз») выделить всю строку.

Другой способ выделения фрагментов текста, а также операции по вырезанию, копированию и вставке этих фрагментов реализуются последовательностями команд, задаваемыми сочетаниями клавиш, как показано в таблице:

Действие	Последовательность команд
Отметить начало блока	Ctrl+K B ¹
Отметить конец блока	Ctrl+K K
Скопировать блок в буфер	Ctrl+Ins
Вставить блок из буфера	Shift+Ins
Вырезать блок, поместив его в буфер	Shift+Del
Скрыть / отобразить блок	Ctrl+K H

Мы не будем здесь описывать отдельно команды меню **Edit**, поскольку они реализуют те же самые действия, которые приведены в таблице.

Меню Run

Меню **Run** содержит команды, предназначенные для выполнения программы как в обычном, так и в отладочном режиме.

- Команда **Run > Run (Ctrl+F9)** вызывает выполнение откомпилированной ранее программы. Если с момента последней компиляции исходный код был моди-

¹ Запись **Ctrl+K B** означает следующее: вы должны, удерживая клавишу **Ctrl**, нажать сначала клавишу **K**, а затем, отпустив ее, клавишу **B**.

фицирован, то команда последовательно выполнит следующие действия: компиляция, компоновка, выполнение.

Остальные команды меню используются при отладке программы:

- Команда **Run > Program reset (Ctrl+F2)** останавливает текущий сеанс отладки, освобождает память, выделенную для программы, и закрывает все открытые файлы.
- Команда **Run > Go to cursor (F4)** вызывает выполнение программы до того оператора, перед которым установлен текстовый курсор.
- Команда **Run > Trace into (F7)** осуществляет пошаговое (оператор за оператором) выполнение программы, при этом если встречается вызов функции, то трассировка продолжается с заходом в тело функции и пошаговым выполнением операторов внутри функции.
- Команда **Run > Step over (F8)** осуществляет пошаговое выполнение программы, при этом если встречается вызов функции, то функция выполняется как один оператор (без захода в тело функции).
- Команда **Run > Arguments** позволяет задавать аргументы командной строки точно так же, как если бы они вводились при запуске программы из командной строки DOS.

Меню Compile

Меню **Compile** содержит команды, предназначенные для компиляции программы, находящейся в активном окне, а также для полной или выборочной перекомпиляции всех файлов текущего проекта.

- Команда **Compile > Compile (Alt+F9)** вызывает компиляцию исходного файла (с расширением .c или .cpp) в активном окне редактора. Сообщения компилятора об ошибках и предупреждениях выводятся в окно **Message**, которое при этом становится активным. Если компиляция прошла успешно, то создается одноименный файл с расширением .obj.
- Команда **Compile > Make (F9)** вызывает избирательную перекомпиляцию тех файлов проекта, в которых произошли изменения с момента последней компиляции.
- Команда **Compile > Link** использует текущие .obj и .lib файлы, либо задаваемые в файле проекта, либо используемые по умолчанию, для компоновки в результатирующий .exe файл.
- Команда **Compile > Build all** осуществляет полную перекомпиляцию всех файлов проекта, независимо от того, вносились ли в них изменения с момента последней компиляции.

Меню Debug

Меню **Debug** содержит команды, управляющие работой встроенного отладчика.

- Команда **Debug > Inspect (Alt+F4)** открывает окно **Inspector**, с помощью которого можно анализировать значения объектов¹.

¹ Используется при отладке программ на C++ с использованием классов.

- Команда Debug > Evaluate/modify (Ctrl+F4) открывает диалоговое окно с тремя полями: Expression, Result, New value, — с помощью которых можно отображать значения переменных (выражений), а также модифицировать значения переменных.
 - Команда Debug > Call stack (Ctrl+F3) открывает диалоговое окно, отображающее стек вызовов — последовательность функций, которые вызывались с момента старта программы.
 - Команда Debug > Watches вызывает всплывающее меню, позволяющее оперативно добавлять новые выражения просмотра.
 - Команда Debug > Breakpoints... открывает диалоговое окно «Breakpoints», позволяющее назначать или отменять точки прерывания, на которых будет останавливаться программа в отладочном режиме. Это окно содержит поле списка «Breakpoint List», в котором отображается список установленных точек прерывания, и ряд управляющих кнопок, из которых чаще всего используются OK, Edit и Delete. Чтобы добавить новую точку прерывания, вы должны позиционироваться на кнопке Edit (напомним, что навигация по диалоговому окну производится при помощи клавиши Tab) и нажать клавишу Enter. Появится диалоговое окно «Breakpoint Modify/New», содержащее четыре текстовых поля и четыре кнопки: Modify, New, Cancel и Help. Текстовые поля предназначены для ввода следующей информации:
 - Condition — условие, при котором произойдет останов (любое допустимое С-выражение, в котором нет: а) вызовов функций, б) макросов, в) локальных или статических переменных, лежащих вне области видимости выполняемой функции).
 - Pass Count — сколько раз точка прерывания пропускается, прежде чем произойдет останов.
 - File Name — имя файла с «исходником».
 - Line Number — номер строки в этом файле, на которой произойдет останов.
- Введя все это и нажав на кнопку New, вы получите новую точку прерывания, а строка с указанным номером подсветится красным фоном.

Меню Project

Меню Project содержит команды, необходимые для создания, модификации, открытия и закрытия проекта.

Проект — это набор взаимосвязанных исходных файлов и, возможно, объектных файлов, компиляция и компоновка которых приводит к созданию исполняемой программы. Использование проекта обязательно при разработке многофайловых программ. Однако мы рекомендуем вам создавать проект и для однофайловой программы, поскольку в так называемом проектном файле (имеющем расширение .prj) запоминаются все установки интегрированной среды, что существенно облегчит вашу программистскую жизнь при долговременном сопровождении программы.

Меню Options

Меню Options содержит команды, позволяющие просматривать и модифицировать различные установки (опции) интегрированной среды. Для большинства из этих настроек можно оставить значения, заданные по умолчанию. Но вот команда Options > Directories требует особого внимания, поскольку в ней задаются пути к заголовочным файлам (Include Directories) и библиотекам (Library Directories), а также к каталогу, в который будут помещаться файлы с расширениями .obj, .exe и .map, создаваемые средой (Output Directory).

Например, если ВС установлен в каталоге C:\BORLANDC, то в поле Include Directories необходимо указать путь C:\BORLANDC\INCLUDE, а в поле Library Directories — путь C:\BORLANDC\LIB. В поле Output Directory целесообразно указать текущий каталог проекта — это делается вводом символа точки «..».

Если вы собираетесь пользоваться услугами встроенного отладчика, то проверьте также опцию Source Debugging. Для этого выполните команду Options > Debugger и в открывшемся диалоговом окне «Debugger» установите переключатель Source Debugging в положение On.

После того как вы закончили работу с настройками среды, их следует сохранить, выполнив команду Options > Save...

Меню Window

Меню Window содержит команды управления окнами. Назначение большинства из них понятно из их названий. Назначение можно уточнить, выделив команду и нажав клавишу F1. Будет вызвана встроенная справочная помощь.

Создание нового проекта

Для создания нового проекта выполните следующие действия:

- Выберите в строке меню команду Project > Open project...
- Откроется окно диалога «Open Project File», напоминающее по своему устройству и приемам работы рассмотренное выше окно диалога «Save File As».
- В открывшемся диалоговом окне выберите нужный каталог, пользуясь полем списка «Files».
- Введите в текстовое поле Open Project File требуемое имя проекта *ProjName.prg* и нажмите клавишу Enter.

Откроется окно проекта «Project: *ProjName*» — пока что пустое, без файлов, а строка состояния главного окна ВС изменит свой вид: в ней появятся команды для добавления (Ins) файлов в текущий проект и их удаления (Del) оттуда.

В проект могут добавляться как исходные файлы (с расширением .c или .cpp), так и объектные файлы (с расширением .obj), полученные ранее путем компиляции исходных файлов. Все добавляемые файлы должны находиться в текущем каталоге проекта. Заголовочные файлы (с расширением .h) не должны

добавляться в проект, но должны находиться в текущем каталоге проекта, чтобы соответствующие директивы `#include` в исходных файлах работали нормально.

- Для добавления файла к проекту нажмите клавишу *Insert*.

Откроется окно диалога «Add to Project List». Пользуясь полем «Files», найдите нужный файл, перейдите на кнопку *Add* и нажмите клавишу *Enter*.

После того как будут добавлены все нужные файлы, перейдите на кнопку *Done* и нажмите клавишу *Enter*.

Модификация существующего проекта

Для добавления файлов в существующий проект (или удаления из него файлов) действуйте по той же схеме, что и при создании нового проекта.

Если после команды *Project > Open project...* окно проекта не появится на экране, то выполните команду *Window > Project*.

Добавление файлов выполняется путем нажатия клавиши *Insert*, удаление — путем нажатия клавиши *Delete*. Любая работа должна завершаться нажатием кнопки *Done*.

Открытие проекта

Чтобы открыть проект, с которым вы работали ранее, выполните следующие шаги:

- Выберите в строке меню команду *Project > Open project...*

Откроется окно диалога «Open Project File».

- В открывшемся диалоговом окне найдите нужный каталог, пользуясь полем «Files», а в этом каталоге — нужный проектный файл *ProjName.prg*.

- Перейдите на кнопку *OK* и нажмите клавишу *Enter*.

Работа с проектом

Ведите нужный текст в нужные файлы или отредактируйте существующий текст. Нажмите клавиши *Ctrl+F9* (команда *Run > Run*), чтобы запустить компиляцию, сборку и выполнение. При этом безразлично, какой файл находится в активном окне. Менеджер проектов, входящий в среду BC, сам разберется, какие файлы надо компилировать, и создаст исполняемый файл *ProjName.exe*. Если не будет ошибок компиляции и сборки, то программа будет запущена на выполнение. Если ошибки будут, вам придется заняться отладкой.

Завершение работы с проектом

Чтобы закрыть проект, выберите в строке меню команду *Project > Close project*.

Работа с отладчиком

Рекомендуем вам ознакомиться с использованием встроенного отладчика Visual C++, описанным в Приложении 1, даже если у вас нет среды VC. Дело в том, что все отладчики в подобного рода пакетах строятся на одних и тех же общих принципах: установка точек прерывания, режим пошагового выполнения, наблюдение за значениями переменных и т. д., и т. п. Поэтому мы не будем повторно описывать технологию отладки программ применительно к отладчику пакета BC. Предоставляем вам возможность самостоятельно повторить в среде BC отладку проекта *Buggy*, который рассматривался в Приложении 1, пользуясь командами меню *Debug* и *Run*, описанными выше.

Алфавитный указатель

&

&, 21
&&, 32

А

ANSI, 220
Application Wizard, 214
ASCII, 220
atof, 94
atoi, 94
atol, 94

В

bool, 32
break, 37

С

case, 37
char, 90, 94
CharToOem, 113, 220
cin, 14
close, 104
conio.h, 36
const, 165
cosh, 42
cout, 14
ctype.h, 101

Д

DBL_EPSILON, 36
default, 37
delete, 60
do while, 37
double, 15

Е

Editor, 213
endl, 15
error, 17

Ф

false, 32
fgets, 118
FILE, 106

fixed, 162
float, 15
FLT_EPSILON, 36
fopen, 106
for, 37
fread, 106, 120
fscanf, 118
fseek, 106
fstream.h, 80
fwrite, 120

Г

get, 91, 94
getch, 36
getchar, 95
getline, 91
gets, 92

Н

heap, 23

И

IDE, 212
if, 30
ifstream, 80, 97
int, 14, 24
iomanip.h, 75
ios, 104
iostream.h, 14
isalpha, 95
isdigit, 95
ispunct, 95
isspace, 95

Л

left, 188
localtime, 188
long, 24

М

main, 14, 132
malloc, 59
math.h, 26
memcpy, 133
Microsoft Developer Studio, 213

Н

new, 59

О

OemToChar, 113, 220
ostream, 80, 104
Output, 213

Р

precision, 162
printf, 21, 92
project, 212
Project Workspace, 212
putchar, 95
puts, 92

Q

qsort, 124

Р

read, 104
reinterpret_cast, 76, 124
return, 133

С

scanf, 21, 92
SEEK_CUR, 106
SEEK_END, 106
SEEK_SET, 106
seekg, 104
setf, 162
setiosflags, 188
setw, 75, 162
short, 24
signed, 24
sin, 133
static, 23
static_cast, 40
stdio.h, 21
stdlib.h, 124
strcpy, 93
string, 89
string.h, 93
strlen, 93
strncpy, 93, 114
strstr, 97
strtok, 101
struct, 110
success, 17
switch, 30, 36

Т

tellg, 104
time, 188

time.h, 188
tm, 188
tolower, 116
toupper, 116
true, 32

У

unsigned, 24

W

warning, 17
while, 37

Б

бинарное дерево, 189
бинарный файл, 120
блок, 35
быстрая сортировка, 63, 170

В

время жизни, 22
выражение, 16

Г

глобальная переменная, 22, 151

Д

двумерный массив, 71
дерево поиска, 189
динамические структуры данных, 169
динамический массив, 59
директива препроцессора, 14

З

заглушки, 181
заголовок функции, 14, 132
заголовочный файл, 14, 218

И

И, 35
извлечение из потока, 14
ИЛИ, 35
имя переменной, 15
индекс, 71
инстанцирование шаблона, 166
интегрированная среда, 212
 Borland C++ 3.1, 227
 Visual C++ 6.0, 212
исполняемый файл, 218

К

кириллица, 220
ключ, 189

командная строка, 214
 компиляция, 14, 218
 компоновка, 218
 консольное окно, 214
 консольное приложение, 214
 конфигурация проекта, 221
 критерии качества программы, 33
 куча, 22

Л

линейная программа, 13
 линейный список, 173
 логическое И, 32
 логическое ИЛИ, 34
 локальная переменная, 22

М

манипулятор, 16
 массив, 55
 мастер приложений, 214
 метод выбора, 82
 многофайловый проект
 Borland C++ 3.1, 232
 Visual C++ 6.0, 218
 модификаторы формата, 21

Н

неоднозначность, 163
 нуль-символ, 89

О

область действия, 22
 объекты-потоки, 14
 окно вывода, 213
 окно рабочей области, 213
 окно редактора, 213
 операторы цикла, 37
 операции отношения, 32
 операция выбора, 111
 операция присваивания, 16
 описание переменных, 22
 опции интегрированной среды, 233
 отладка программы, 162
 отладчик, 222
 очередь, 182

П

параметры по умолчанию, 165
 перегрузка функций, 161
 передача параметров в функцию, 137
 поле структуры, 111
 помещение в поток, 14
 пошаговое выполнение, 222
 приоритет операций, 16
 проект, 212, 232

прототип функции, 132
 прямой доступ, 120

Р

рабочая область, 212
 разадресация, 72
 размерность массива, 55

С

сборка, 218
 сегмент данных, 22
 сегмент кода, 22
 сегмент стека, 22
 символьный литерал, 15
 сортировка, 63
 спецификация формата, 21, 92
 справочная система, 222
 ссылка на указатель, 191
 статическая переменная, 23
 стек, 64, 170
 строки, 89
 структура, 110

Т

тело функции, 14
 тип, 15
 точка прерывания, 222

У

узел дерева, 189
 указатель, 59
 управляющая последовательность, 21
 условная операция, 36
 утечка памяти, 60

Ф

функции, 14, 132
 перегрузка, 161

Х

хип, 22

Ц

цикл, 37, 39

Ч

читаемость, 15

Ш

шаблон функции, 166

Э

элемент динамической структуры данных, 169

Павловская Татьяна Александровна
Щупак Юрий Абрамович
C/C++. Структурное программирование
Практикум

Главный редактор *Е. Строганова*
 Заведующий редакцией *И. Корнеев*
 Руководитель проекта *А. Васильев*
 Литературный редактор *М. Иванов*
 Художник *Н. Биржаков*
 Корректоры *И. Смирнова, И. Хохлова*
 Верстка *Л. Харитонов*

Лицензия ИД № 05784 от 07.09.01.

Подписано к печати 25.10.02. Формат 70x100/16.

Усл. п. л. 19,35. Доп. тираж 4500. Заказ 872

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

Налоговая льгота — общероссийский классификатор

продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ФГУП ордена Трудового Красного Знамени «Техническая книга»
 Министерства Российской Федерации по делам печати, телерадиовещания
 и средств массовых коммуникаций.
 198005, Санкт-Петербург, Измайловский пр., 29.

**Т. А. Павловская
Ю. А. Щупак**

C/C++

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Структурное программирование

ПРАКТИКУМ

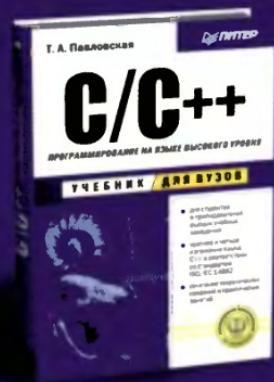
Практикум предназначен для изучения языка C++ на лабораторных работах и практических занятиях по программированию, а также для самостоятельного освоения языка. Он является дополнением к учебнику Т. А. Павловской «C/C++. Программирование на языке высокого уровня», выпущенному издательством «Питер» в 2001 году.

Данная книга является первой частью практикума. В ней рассматриваются возможности C++, используемые в рамках структурной парадигмы написания программ: стандартные типы данных, основные конструкции языка, массивы, строки, функции, шаблоны функций и динамические структуры данных.

В этой книге на примерах рассматриваются различные алгоритмы, методы и приемы написания, отладки и тестирования программ, обсуждаются вопросы качества и стиля. Для каждого семинара приводится от одного до трех комплектов из 20 вариантов заданий.

**Темы, рассмотренные
в книге:**

- основные принципы структурного написания программ
- конструкции языка в соответствии со стандартом ISO/IEC 14882 (1998)
- основные структуры данных — массивы, стеки, списки, очереди, бинарные деревья
- технология создания надежных программ
- модульное программирование
- приемы и методы отладки программ
- описание работы в популярных интегрированных средах



**Не забудьте купить
учебник!**

ISBN 5-94723-447-5



ПИТЕР®
WWW.PITER.COM

Посетите наш web-магазин: <http://www.piter.com>