МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ М. В. ЛОМОНОСОВА
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

МЕТОДЫ ПРОГРАММИРОВАНИЯ

В ЗАДАЧАХ И ПРИМЕРАХ

В. Д. ВАЛЕДИНСКИЙ,
А. А. КОРНЕВ

Московский государственный университет имени М. В. Ломоносова
Механико-математический факультет

В. Д. Валединский, А. А. Корнев

Методы программирования в задачах и примерах на C/C++

Учебное пособие

Издательство Московского университета
2023

УДК 519.68+004.43
ББК 32.97
В15

Рецензент:
доцент кафедры вычислительной математики
механико-математического факультета МГУ имени М. В. Ломоносова,
кандидат физико-математических наук
М. А. Ложников

Корректор:
старший научный сотрудник кафедры вычислительной математики
механико-математического факультета МГУ имени М. В. Ломоносова,
кандидат физико-математических наук
А. Е. Пентус

Валединский, В. Д., Корнев, А. А.

В15 Методы программирования в задачах и примерах на C/C++ : учебное пособие / В. Д. Валединский, А. А. Корнев. — Москва : Издательство Московского университета, 2023. — 413, [1] с. — (Электронное издание сетевого распространения).

ISBN 978-5-19-011886-5 (e-book)

Учебное пособие является обобщением опыта преподавания университетского курса «Работа на ЭВМ и программирование» для студентов механико-математического факультета МГУ имени М. В. Ломоносова и школьного курса «Информатика» в классах при механико-математическом факультете на базе 54-й школы г. Москвы и в математических классах Университетской гимназии. В книге собраны и систематизированы задачи, предлагавшиеся для самостоятельного решения во время семинарских занятий, лабораторных работ, на зачетах и экзаменах.

Книга предназначена учащимся старших классов с углубленным изучением математики и информатики, студентам высших учебных заведений, осваивающим методы программирования и язык C, а также в помощь преподавателям для проведения практических занятий.

УДК 519.68+004.43
ББК 32.97

ISBN 978-5-19-011886-5
(e-book)

Оглавление

ПРЕДИСЛОВИЕ

Учебное пособие написано на основе многолетнего опыта преподавания университетских курсов «Работа на ЭВМ и программирование», «Численные методы», «Практикум на ЭВМ» на мехмате МГУ, а также школьного курса «Информатика» в классах при мехмате и в математических классах Университетской гимназии.

Основной целью данного пособия является изучение методов программирования на основе решения прикладных задач из различных (около)научных областей. В каждом разделе предварительно даются все необходимые определения и обсуждаются идеи построения алгоритмов. Далее для ключевых задач приводятся либо полные решения, либо отдельные фрагменты программного кода. Большинство постановок задач совершенно конкретны: нужно выполнить ту или иную обработку данных по определенному алгоритму. Другие, напротив, сформулированы нечетко и обрисовывают скорее круг идей и возможных подходов к решению.

Процесс решения задачи с применением ЭВМ можно разделить на две части: разработка формального алгоритма и его последующая реализация на конкретном языке. Поэтому в предлагаемой вашему вниманию книге последовательно в форме упражнений разбираются наиболее востребованные (типичные) алгоритмические конструкции и приемы, а также приводятся необходимые сведения для их реализации на языке С.

На наш взгляд, язык С наиболее подходит для изучения основ программирования. На данный момент С — это один из трех «древнейших и здравствующих по сей день» языков программирования. Это небольшой, четко формализованный и прозрачный язык, допускающий операции низкого уровня. На его примере можно не только освоить базовые операторы и конструкции большинства языков высокого уровня, но и детально разобрать процессы, происходящие в памяти компьютера при их исполнении. Четкая структурированность С позволяет ясно реализовывать небольшие алгоритмические задачи, а востребованность в среде профессиональных программистов гарантирует перспективность его изучения. Почему для начального изложения выбран язык С, а, казалось бы, более популярный и современный язык С++ поставлен на второе место? Дело в том, что базовые отличия

этих двух языков заключаются в принципах построения программ — процедурном и объектно ориентированном. А точка зрения авторов состоит в том, что знакомство с программированием, выработку умения составлять алгоритмы, оперирующие с простейшими объектами — числами и символами, нужно начинать именно с освоения процедурного подхода. И только после наработки навыков составления алгоритмов, решающих простейшие задачи, можно переходить к задачам программирования более сложных отношений между объектами, в чем и заключается главная задача объектно-ориентированного подхода. Конечно, на языке C++ вполне можно программировать и оставаясь в рамках процедурного подхода, и иногда это дает некоторые удобства. В таких случаях в тексте книги будут сделаны соответствующие замечания.

Первое издание задачника, вышедшее двадцать лет назад, в первую очередь предназначалось преподавателям и продвинутым слушателям курса «Работа на ЭВМ и программирование». Все разделы текущего издания значительно пополнились новыми задачами, проектными темами, алгоритмами, примерами программ. Добавлена информация для начинающих пользователей, в том числе позволяющая изучать программирование и язык C со школьниками старших классов в рамках уроков информатики в режиме 4 часов в неделю. Также затронуты некоторые популярные в повседневной научной деятельности численные алгоритмы.

Отдавая себе отчет, что стиль программирования — неисчерпаемая тема для споров и дискуссий, авторы все же предлагают конкретные реализации как пример для подражания, надеясь, что при критическом отношении к приведенному коду читатель сможет вынести для себя много полезного из этой книги.

Замечания и комментарии по содержанию пособия просьба сообщать авторам на кафедру вычислительной математики механико-математического факультета МГУ им. М. В. Ломоносова.

Авторы

Правдивая история (*из письма выпускника мехмата МГУ*). «*Помнишь, как на последнем курсе я каждое утро пару часов занимался английским и французским? Большая ошибка! Надо было учить C и Fortran...*»

Глава 1
АЛГОРИТМИЧЕСКИЕ ЗАДАЧИ

Все задачи данной главы можно условно разделить на следующие группы по своему назначению: освоение базовых конструкций и возможностей языка C; изучение однопроходных, рекуррентных и индуктивных алгоритмов обработки последовательностей; получение навыков работы с динамически выделяемой памятью и усвоение разнообразных алгоритмов перестановок; знакомство с популярными методами и приемами, используемыми при реализации реальных прикладных задач.

Прежде чем перейти к изложению материала, отметим несколько простых и полезных правил, сформулированных классиками методов программирования. Всегда следует помнить, что основной целью программиста является разработка кода, правильно решающего поставленную задачу. Если программа работает неверно, то ее эффективность, компактность, изящность не имеют значения. Применяя известное высказывание Андрея Николаевича Колмогорова «Каждая теорема либо очевидна, либо неверна», получаем, что программа либо «прозрачна», либо содержит ошибки. Любые «мутные» места в алгоритме или коде, замечания компилятора на этапе сборки, неоднозначности при тестировании указывают на наличие потенциального источника ошибок.

Разработку даже небольшой программы следует начинать с технического задания: четкой формализации, что является набором входных данных, что — итогом работы. Алгоритм необходимо выбирать самым тщательным образом: пятнадцать минут предварительных размышлений часто помогают избежать многих часов программирования, не приводящего к желаемому результату. Если на начальном этапе нет четверти часа на детальный анализ алгоритма, то будьте готовы выделить сутки на переделку и отладку программы.

При оформлении кода не следует забывать, что программы читаются людьми. Правильный выбор имен переменных — залог удобочитаемости программы. Для каждой функции пишите комментарии к заголовку с описанием решаемой задачи и входных/выходных параметров. В тело функции включайте пояснения, дополняющие программный код. Однако помните, что неточные комментарии хуже, чем их отсутствие. При оформлении кода используйте отступы и скобки для наглядности — скобки

и пробелы обходятся дешевле, чем ошибки. Одного оператора в строке достаточно. Каждый логический шаг алгоритма выделяйте в отдельную функцию — не стоит зубочистку и расческу крепить к одной ручке.

По возможности всегда используйте стандартные библиотеки — не нужно изобретать колесо. Перед началом работы с библиотечной функцией изучите ее описание и примеры использования, а по окончании вызова проверьте, что получили правильный результат.

Каждое предупреждение, выдаваемое компилятором, — это указание на наличие потенциальной ошибки, хотя для ее проявления может потребоваться время.

Приступая к тестированию, помните, что программа без ошибок есть абстрактное теоретическое понятие. Поэтому тестирование призвано выявить ошибки, а не подтвердить правильность работы программы. Так как исчерпывающее тестирование невозможно (число потенциальных ошибок много больше числа проводимых проверок), выбирайте наиболее эффективные тесты — каждый следующий вариант должен соответствовать новому классу входных данных. Выполняйте эхо-проверку считанных данных. Испытывайте программу для нормальных, экстремальных и исключительных (но, конечно, строго допустимых) входных условий. В процессе написания программы продумывайте варианты unit-тестов, позволяющих в дальнейшем проверить правильность работы отдельных модулей. Проводите полное тестирование после каждого внесения изменений.

Программная ошибка на жаргоне «баг» (bug, клоп), и ловить ее нужно соответствующим образом. Загоняем bug в угол: на максимально простом тесте шаг за шагом локализуем место ошибки в программном коде; прижимаем багу хвост: находим переменную с неправильным значением; вытаскиваем баг: последовательно отслеживая операторы, отвечающие за формирование данной ячейки, находим ошибочную языковую либо логическую конструкцию. При отладке нужно исходить из постулата: если ваша программа выдает неправильный ответ, то либо в коде, либо во входных данных, либо в выбранном алгоритме имеется ваша ошибка (в том числе алгоритм может быть банально несовместим с конкретными входными данными). Голословно утверждать, что проблема связана с системными библиотеками, компилятором,

железом, бесперспективно — вероятность такого события слишком мала.

При решении задач придерживайтесь правила:

«Сделай сам! И разберись, как это делают другие».

1.1. Введение в язык С

При решении задач данного раздела условимся, что ввод необходимых входных данных выполняется с клавиатуры после выдачи программой соответствующих указаний на экран. Далее программа вычисляет искомые величины и сохраняет соответствующие значения в отдельных переменных, а затем с пояснениями выводит на экран полученный результат.

Задача 1-1-1. Найдите минимум из двух целых чисел a и b.

Решение. Приведем формальное решение.

```
/* Выбор минимума из двух целых чисел
   Автор(ы): User
   Copyright (c):  1996-2020 User
   Лицензия: GNU Lesser General Public License */
#include <stdio.h>
int main(void) {
  int a, b, min;
  //  ввод данных
  printf("Введите целое число a ");
  scanf("%d", &a);
  printf("Введите целое b ");
  scanf("%d", &b);
  //  вычислительная часть
  if (a < b) {
    min = a;
  } else {
    min = b;
  }
  //  вывод результата
  printf("Минимум из a = %d и b = %d ", a, b);
  printf("равен %d\n", min);
  return 0;
}
```

Предложенный вариант программы формально решает поставленную задачу, поэтому на «нулевом» шаге обучения (т. е. на этапе освоения редактора, компилятора и элементарных конструкций языка) его условно можно принять за образец. Разберем код подробнее. Заключенный между символами `/*...*/` текст представляет собой многострочный комментарий и при компиляции игнорируется. А точнее, заменяется на пустые строки на этапе препроцессирования (об этапах компиляции см. п. 5.1.3). Следующая строка (т. е. строка 7) начинается с символа `#` и поэтому по определению является командой препроцессора. В результате выполнения команды `include` содержимое указанного в «уголках» файла, хранящегося в специальной системной директории, будет включено в текст программы вместо данной строки. В файле `stdio.h` (от англ. standard input/output header) содержатся, в частности, прототипы (т. е. заголовки) основных библиотечных функций стандартного ввода/вывода. Эта информация позволит на этапе компиляции организовать правильное взаимодействие главной функции `main()` с вызываемыми из нее библиотечными функциями `printf()` и `scanf()`.

В строке 8 записан заголовок функции `int main(void)`, согласно которому функция не получает при запуске из командной строки входной информации и возвращает в точку вызова (в командную строку) целое число. Далее в фигурных скобках содержатся операторы, составляющее тело функции `main()`. В 9-й строке создаются три целочисленные переменные. В общем случае имена переменных могут содержать строчные и прописные латинские буквы, цифры и символ подчеркивания; начинать рекомендуется с маленькой буквы и запрещается с цифры. В строках 11—14 (и далее в 22 и 23) вызываются библиотечные функции, отвечающие за вывод информации на экран и считывание данных с клавиатуры. Обращаем ваше внимание, что в простейшем варианте (см. строки 11 и 13) первым и единственным параметром функции вывода `printf()` является заключенный в двойные кавычки текст, подлежащий печати. Если, помимо текста, требуется выдать на экран содержимое некоторой переменной (строка 23), то через запятую вторым параметром указывается ее имя — это обеспечивает передачу *содержимого* переменной в функцию `printf()`. В этом случае заключенный в двойные кавычки текст должен обязательно содержать последовательность символов, начинающуюся с управляющего символа `%` и заканчиваю-

щуюся символом — ключом формата, которая указывает позицию для печати и регламентирует правила вывода соответствующей переменной. В данном примере `%d` означает, что содержимое передаваемой переменной необходимо проинтерпретировать как целое число и распечатать его в стандартном десятичном (decimal integer) формате. В текст также добавлен управляющий символ `\n`; выдача его на экран переводит курсор печати в начало следующей строки. При желании можно распечатать содержимое сразу нескольких переменных (строка 22).

В базовом варианте (см. строки 12 и 14) первым параметром функции `scanf()` является заключенная в двойные кавычки строка, содержащая управляющий символ `%` и ключ формата `d`. А далее после запятой передается *адрес* переменной (т. е. адрес ячейки памяти, где содержимое соответствующей переменной хранится). Вычисление адреса осуществляется с помощью унарной операции `&`. В случае успешного считывания данных функция `scanf()` запишет полученное целое число по указанному адресу.

За поиск минимума отвечает условный оператор **if-else** (см. строки 16—20). Если условие `(a <b)` истинно, то выполняется блок операторов, записанный за данным условием, т. е. строка 17; иначе выполняется блок операторов за ключевым словом **else**, т. е. строка 19.

Логические части программы разделены однострочными комментариями (строки 10, 15 и 21), предназначенными для облегчения чтения кода и также исключаемыми из текста на этапе препроцессирования.

Оператор **return** `0;` обеспечивает выход из функции `main()`, т. е. возвращение в командную строку, откуда соответствующий исполняемый файл был запущен. Принято считать, что число 0 означает штатное завершение программы.

Замечание. В данном случае текст написан без учета принципа процедурного программирования: вся работа реализована в единственной функции `main()`. Это принято считать неправильным и неграмотным даже для простых программ. При решении любой задачи нужно выделить независимые логические и алгоритмические блоки, отвечающие за разные этапы алгоритма, и оформить их в виде отдельных функций. В данном тривиальном примере тоже можно указать две части: интерфейсная часть (т. е. общение с пользователем, ввод данных и вывод результата) и

вычислительная часть (поиск минимума). Поэтому перепишем программу заново, выделив вычислительную часть в отдельную функцию, а заодно продемонстрируем другие конструкции и возможности языка. Решения последующих задач рекомендуется оформлять с учетом данного замечания, т. е. выделяя логически независимые части алгоритма в отдельные функции.

```
1  #include <stdio.h>
2  int Min(int, int);
3  int main(void) {
4    int a, b, abMin;
5    printf("Введите целые a и b \t");
6    scanf("%d%d", &a, &b);
7    abMin = Min(a, b);
8    printf("min(%d,%d) = %d\n", a, b, abMin);
9    return 0;
10 }
11 int Min(int x, int y) {
12   int z;
13   z = (x < y) ? x : y;
14   return z;
15 }
```

Компилятор будет анализировать представленный программный код сверху вниз, исполнение программы начнется с функции `main()`. В первой строке происходит подключение заголовочного файла, необходимого в данном случае для корректной работы функций `printf()` и `scanf()`. Во второй строке на внешнем уровне, т. е. вне тела `main()`, определяется новый объект — функция с именем `Min()`, зависящая от двух целочисленных переменных и возвращающая целое число. Имена функций строятся по тем же правилам, что и имена переменных. Рекомендуется имена функций начинать с заглавной буквы.

Функция — это группа операторов, выделенных в отдельный самодостаточный блок, которому присвоено уникальное имя. В данном случае это операторы, заключенные между открывающей (строка 11) и закрывающей (строка 15) фигурными скобками. Предварительное описание прототипа функции в строке 2 позволяет компилятору организовать из строки 7 ее корректный вызов, т. е. передачу параметров и прием результата. Прототип (заголовок) функции должен содержать перечисленные через за-

пятую типы входных параметров (для наглядности можно указывать и имена соответствующих переменных), а также тип возвращаемого значения. Прототип необходимо описать на внешнем уровне и до первого вызова функции.

Вызов функции — это переход к выполнению операторов соответствующего блока. При исполнении операторов конкретной функции мы находимся внутри ее блока, поэтому нам недоступны переменные иных функций. Следовательно, все необходимые значения требуется передать в функцию в момент вызова.

В данном случае при вызове функции `Min()` из строки 7 содержимое переменных `a`, `b` копируется соответственно в ячейки `x`, `y`, описанные в строке 11. Обращаем внимание, что любое изменение переменных `x`, `y` внутри функции `Min()` не повлечет изменение содержимого `a`, `b`.

В строке 12 создается новая переменная `z`, в строке 13 ее значение вычисляется посредством тернарного оператора «знак вопроса». При выполнении оператора проверяется стоящее в скобках логическое условие; если оно истинно, то переменной `z` присваивается содержимое переменной `x`, иначе — содержимое переменной `y`.

Далее (строка 14) оператор `return z;` обеспечивает возврат в точку вызова (т. е. в строку 7) и присвоение возвращаемого значения `z` переменной `abMin`. Затем начинается выполнение следующих операторов функции `main()`, т. е. строк 8 и 9. С этого момента переменные `x`, `y`, `z` становятся недоступны.

Замечание. Печатаемый функцией `printf()` текст (см. строки 5 и 8) содержит управляющие символы `\n` (новая строка) и `\t` (табуляция). Их использование позволяет организовать наглядный интерфейс. Отметим, что `\n` *следует всегда* добавлять в конец выдаваемых сообщений при поиске ошибок в программе методом отладочной печати. Дело в том, что отвечающие за вывод библиотечные функции предварительно накапливают некоторое достаточно большое количество символов в специальном буфере и только потом передают эти символы на устройство вывода. Если программа аварийно завершается, то содержимое буфера теряется. Символ `\n` почти всегда обеспечивает досрочное проталкивание буфера печати — выдачу его содержимого на экран независимо от степени заполнения.

Замечание. В рассмотренных примерах функция `int main(void)` не имеет входных параметров (на это указывает

ключевое слово **void**), а в качестве результата возвращает в точку своего вызова число типа **int**. Так как функция вызывается из интерпретатора команд, то ему и предназначена возвращаемая оператором **return** величина. Большинство версий интерпретаторов сохраняют код выхода последней команды в системной переменной с именем `$?`, а ее содержимое можно посмотреть, набрав `echo $?` в командной строке.

Задача 1-1-2. Найдите максимум из целых чисел a и b.

Задача 1-1-3. Вычислите модуль целого числа x.

Указание.

```
int imod(int);
int imod(int x) {
  if (x < 0)
    return (-x);
  else
    return x;
}
```

Представленная функция является аналогом функции `int abs(int)` из стандартной библиотеки (заголовочный файл `stdlib.h`).

Задача 1-1-4. Замените содержимое a на |a|.

Указание. Например, `if (a < 0) a = -a;` Предложите решение на основе оператора «знак вопроса».

Задача 1-1-5. Найдите знак целого числа a, т. е. реализуйте функцию `int sign(int a)`, где

$$\mathrm{sign}(a) = \begin{cases} 1, & a > 0, \\ 0, & a = 0, \\ -1, & a < 0. \end{cases}$$

Задача 1-1-6. Для заданных целых чисел a, b, c найдите количество целочисленных решений уравнения $a \cdot x + b = c$. Рассмотрите все возможные варианты.

Указание. При решении удобно использовать операторы сравнения `==` (равно) и `!=` (не равно).

Так как вычисления будут проходить в рамках целочисленной арифметики с «округлением вниз», то, например, для ненулевого значения коэффициента a можно формально вычислить x, а затем проверить, что найденное число является корнем.

Задача 1-1-7. Для заданного действительного параметра a найдите наибольшее из чисел $10a + 7$ и $a^2 - 2a + 1$.

Решение. Для работы с действительными числами в языке C имеются типы **float**, **double** и **long double**, отличающиеся количеством памяти, выделяемой для их хранения, и как следствие, диапазоном допустимых значений. Общие правила работы с указанными типами похожи, поэтому в подобных задачах мы обычно будем использовать тип **double** как наиболее популярный. Приведем центральную часть «спартанского варианта» решения:

```
double a, f1, f2, fMax;
scanf("%lf", &a);
f1 = 10 * a + 7;
f2 = a * a - 2 * a + 1;
if (f1 > f2){
  fMax = f1;
}
else{
  fMax = f2;
}
printf("%lf\n",fMax);
```


и полное решение с выделением каждого логического шага в отдельную функцию:

```
#include <stdio.h>
/*--- Прототипы функций ---*/
double Max(double, double);
double F1(double);
double F2(double);
/*--- Главная функция ---*/
int main(void) {
  double a, fMax;
  printf("Введите a ");
  scanf("%lf", &a);
  fMax = Max(F1(a), F2(a));
  printf("Максимум из F1(a) и F2(a) ");
  printf("для a = %lf равен %lf\n", a, fMax);
  return 0;
}
```

```
/*--- Вспомогательные функции ---*/
double F1(double a) {
  return (10 * a + 7);
}
double F2(double a) {
  return (a * a - 2 * a + 1);
}
/*--- Рабочая функция ---*/
double Max(double y1, double y2) {
  return (y1 > y2) ? y1 : y2;
}
```

Задача 1-1-8. Для заданного действительного x выберите наибольшее из следующих чисел:

$$x^2 - 3, \quad 5(x-12)(x+2), \quad 17x+1, \quad (x+1)(x+2)(x-3) - 10.$$

Задача 1-1-9. Проверьте, принадлежит ли вещественное число x отрезку [a, b].

Решение. Приведем только искомую функцию. Так как ответом формально является не число, а утверждение — принадлежит или нет, то нужно представить результат работы функции в виде условных числовых значений — кода возврата. Например, в нашем случае мы можем считать, что число 0 соответствует «попаданию» в отрезок, а значения ∓ 1 показывают, слева или справа от отрезка расположено число x.

```
int Inside(double x, double a, double b);
int Inside(double x, double a, double b) {
  int answer = 0;
  if (x > b) answer = 1;
  if (x < a) answer = -1;
  return answer;
}
```

Отметим, что функция из задачи 1-1-5 имеет похожую структуру.

Замечание. В большинстве современных ЭВМ действительные числа хранятся в формате *с плавающей точкой*, что имеет свою специфику (см. главу 2). Например, значения, вычисленные по математически эквивалентным, но разным по структуре формулам, могут отличаться в младших разрядах, т. е. окажут-

ся формально разными. Как следствие, сравнение на равенство двух переменных типа **double** (а также **float** и **long double**) не является корректным действием: обычно компилятор выдает по этому поводу предупреждение, а иногда такая операция вообще запрещается настройками компилятора. Например, в данном случае при $a = 0{,}3$, $b = 0{,}7$ и введенном с клавиатуры $x = 0{,}3$ не гарантируется, что функция вернет `answer = 0`. Отметим также, что для подобных задач не рекомендуется использовать менее универсальную конструкцию

```
if ((a - x) * (b - x) <= 0) {
  // x из [a,b]
} else {
  // x вне [a,b]
}
```

В данном случае, например, при достаточно больших (но допустимых) входных параметрах мы можем в результате умножения получить переполнение.

Задача 1-1-10. Проверьте, принадлежит ли число x объединению отрезков $[1, 11]$, $[101, 1001]$.

Задача 1-1-11. Проверьте, принадлежит ли число x объединению и/или пересечению отрезков $[a1, b1]$, $[a2, b2]$.

Задача 1-1-12. Проверьте, принадлежит ли число x интервалам $(a1, b1)$, $(a2, b2)$, $(a3, b3)$, и если принадлежит, то укажите номер каждого такого интервала.

Указание. Решение нужно оформить в виде трехкратного последовательного вызова функции типа `Inside()` из задачи 1-1-9.

Задача 1-1-13. Вычислите

$$|a| + |b|, \quad \big||a| - |b|\big|, \quad \big||a + b| - |a - b|\big|.$$

Указание. Решение нужно оформить в виде последовательных и вложенных вызовов функции модуля: **int** `abs(int)` для целых чисел (стандартная библиотека, заголовочный файл `stdlib.h`), **double** `fabs(double)` для вещественных чисел (математическая библиотека, заголовочный файл `math.h`, ключ `-lm` при сборке (линковке) исполняемого exe-файла).

Задача 1-1-14. Замените содержимое a на значение $a + b$, содержимое b на значение $a - b$ без использования дополнительных переменных.

Задача 1-1-15. Поменяйте содержимое a и b местами без использования дополнительных переменных.

Указание. См. задачу 1-1-14. Не стоит без необходимости использовать данный прием — это не только делает код «непрозрачным», но и может, например при больших значениях a и b, привести к ошибочному ответу в результате переполнения.

Задача 1-1-16. Найдите сумму $\mathrm{sum} = 1 + \ldots + n$ с использованием конструкции цикла `for()`. Напомним, что $\mathrm{sum} = n(n+1)/2$.

Решение.

```
int NatSum(int);
int NatSum(int n) {
  int sum = 0, i;
  for (i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}
```

Задача 1-1-17. Найдите величину $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Расчеты проведите на основе следующих типов данных, печатая результат по соответствующему формату:

```
short int                "%hd"
unsigned short int       "%hu"
int                      "%d"
unsigned int             "%u"
long int                 "%ld"
unsigned long int        "%lu"
long long int            "%lld"
unsigned long long int   "%llu"
size_t                   "%zu"
```

В зависимости от архитектуры используемой ЭВМ и настроек компилятора для переменных каждого типа выделяется свое количество байт, узнать его можно с помощью оператора `sizeof(имя типа)`, возвращающего переменную типа `size_t`, по сути беззнаковое целое. Определение размера позволяет для каждого типа еще до фактического вычисления факториала найти наибольшее значение n, для которого программа выдаст верный

ответ. Экспериментально проверьте правильность теоретических оценок.

Задача 1-1-18. Найдите сумму первых кратных 7 и не кратных 11 чисел при условии, что значение суммы не превосходит sMax.

Решение.

```
int Sum7Mult11NonMult(int sMax);
int Sum7Mult11NonMult(int sMax) {
  int sum = 0, i;
  for (i = 1; ; i++) {
    if (i % 7 != 0) continue;
    if (i % 11 == 0) continue;
    if (sum + i > sMax) break;
    sum += i;
  }
  return sum;
}
```

Задача 1-1-19. Для заданных $n > 1$ и $nkMax > 0$ найдите величину $sum = 0 + n^0 + n^1 + \ldots + n^k$ для максимального k, удовлетворяющего условию $n^k < nkMax$.

Указание. Считая, что допустимо решение со сложностью O(k) действий, реализуем алгоритм на основе цикла **while()**:

```
double BoundSum(double n, double nkMax);
double BoundSum(double n, double nkMax) {
  double sum = 0;
  double nk = 1;
  while (nk < nkMax) {
    sum += nk;
    nk *= n;
  }
  return sum;
}
```

Отметим, что, используя тождество

$$sum = \frac{(1 + n + \ldots + n^k) \cdot (1 - n)}{1 - n} = \frac{1 - n^{k+1}}{1 - n}, \quad n \neq 1,$$

с помощью функций математической библиотеки

double log(double) (*натуральный* логарифм),

double pow(double) (возведение в степень),

int round(double) (округление к ближайшему целому)

можно аналитически вычислить соответствующее значение k и искомую сумму. Напомним, что для использования функций из математической библиотеки следует подключить заголовочный файл **math.h** и указать ключ **-lm** при сборке исполняемого exe-файла.

Задача 1-1-20. Обеспечьте в задаче 1-1-19 ввод с клавиатуры положительного значения nkMax.

Указание. Например,

```
do {
  printf("Введите положительное число nkMax ");
  scanf("%lf", &nkMax);
} while (nkMax <= 0);
```

Задача 1-1-21. Реализуйте «калькулятор» для выполнения базовых действий с целыми числами. Функция **main()** должна отвечать за ввод кода арифметической операции (целого числа) и необходимых данных, за вызов соответствующей функции и печать результата.

Указание. Работу «калькулятора» можно оформить в виде бесконечного цикла следующего вида:

```
while (1) {
  scanf("%d", &code);
  if (code < 0) break;
  if (code == 0) PrnMenu();
  if (code == 1) {
    scanf("%d%d", &x, &y);
    printf("%d + %d  = %d\n", x, y, Sum(x, y));
  }
  if (code == 2) {
     .....
  }
  .....
}
```

Задача 1-1-22. По заданным параметрам изображений (например, высоте и ширине рисунка) выведите на экран псевдографические картинки следующего вида:

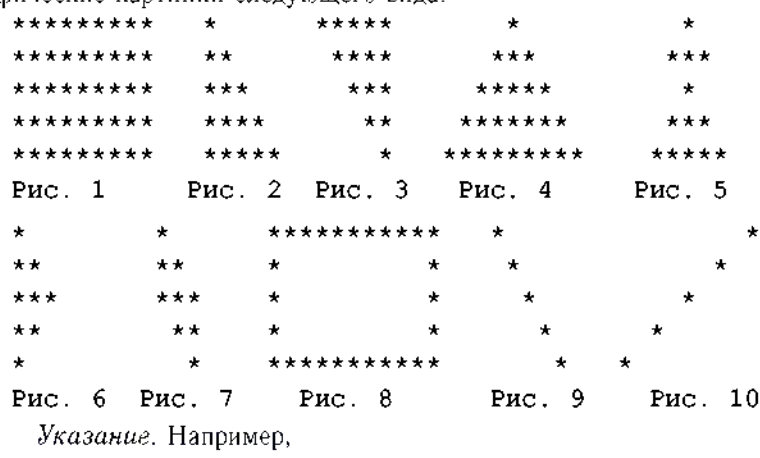


Рис. 1 Рис. 2 Рис. 3 Рис. 4 Рис. 5

Рис. 6 Рис. 7 Рис. 8 Рис. 9 Рис. 10

Указание. Например,

```
void Pict8(int nx, int ny);
void Pict8(int nx, int ny) {
  int i, j;
  for (j = 0; j < nx; j++) printf("*");
  printf("\n");
  for (i = 1; i < ny - 1; i++) {
    printf("*");
    for (j = 1; j < nx - 1; j++) printf(" ");
    printf("*\n");
  }
  for (j = 0; j < nx; j++) printf("*");
  printf("\n");
}
```

Как отработает функция для $nx, ny \leqslant 0$?

Замечание. Если реализована функция `Pict4()` для рис. 4, то `Pict5()` может быть получена на основе многократного (в данном случае двукратного) вызова `Pict4()`.

Замечание. Для обеспечения в задаче 1-1-22 ввода положительных значений nx, ny можно использовать конструкцию вида

```
do {
  printf("Введите два натуральных числа \n");
  scanf("%d %d", &nx, &ny);
```

```
} while ((nx <= 0) || (ny <= 0));
```

Замечание. Функция `printf()` поддерживает механизм escape-последовательностей, позволяющий «раскрашивать» выдаваемые на консоль символы, менять цвет фона и управлять координатами позиции печати, что, по сути, является стандартным упрощенным аналогом библиотеки `ncurses`. Например:

```
/* выбираем один из восьми цветов для символов
 30 (черный); ...  37 (белый); */
int color = 32;  // зеленый
// устанавливаем цвет символов:
printf("\033[%dm", color);
// устанавливаем координаты печати:
printf("\033[%d;%dH", y, x);
// печатаем требуемый текст:
printf("**Green text**\n");
// обязательный сброс цветов
// до системных установок:
printf("\033[0m");
// очистка экрана и сброс координат печати:
printf("\033[2J");
```

Задача 1-1-23. Для заданных $sMax > 0$, m, n найдите, используя конструкции циклов **for**, **while**, **do-while**, максимальное значение $sum = 0 + 1 + (m + n) + \ldots + (m + k \cdot n)^k$, удовлетворяющее условию $sum < sMax$. Корректность неравенства $sMax > 0$ необходимо контролировать на этапе ввода данных.

Задача 1-1-24. Реализуйте конструкции, обеспечивающие ввод двух целых чисел `x1`, `x2` при условии, что в результате

1) оба числа должны оказаться положительными;
2) первое число неотрицательно;
3) хотя бы одно число неотрицательно;
4) ровно одно число положительно.

Для каждого пункта предложите три решения (на основе циклов **for**, **while** и **do-while**).

Указание. Например, можно использовать сложные условия с логическими операторами `!` (не), `&&` (и), `||` (или).

Задача 1-1-25. Вычислите значение функции n!!, т. е. для нечетного n найти произведение $1 \cdot 3 \cdot \ldots \cdot n$, а для четного — произведение $2 \cdot 4 \cdot \ldots \cdot n$.

Задача 1-1-26. Для положительного вещественного числа n найдите ближайшее к нему целое число вида $7k + 5$, распечатайте его значение и значение соответствующего k. Реализуйте решение и с использованием цикла, и по явным формулам.

Задача 1-1-27. Для заданного неотрицательного целого числа n найдите его наибольший делитель m, отличный от n.

Указание. Для случая $n \geqslant 2$ имеем

```
for (m = n / 2; m > 1; m--) {
  if (n % m == 0) break;
}
if (m == 1) {
  printf("Число %d является простым\n", n);
} else {
  printf("Наибольший делитель ");
  printf("числа %d равен %d\n", n, m);
}
```

Модифицируйте код для всех $n \geqslant 0$.

Задача 1-1-28. Определите, является ли введенное число n простым.

Указание. Нужно последовательно проверить делимость n на числа $2, 3, \ldots, k$, пока $k * k \leqslant n$, т. е. на числа от 2 до $\sqrt{n}$.

Задача 1-1-29. Распечатайте все делители числа n.

Задача 1-1-30. Распечатайте все простые делители числа n.

Задача 1-1-31. Распечатайте разложение числа n на простые множители.

Задача 1-1-32. Найдите n-е простое число.

Задача 1-1-33. Найдите все простые числа, не превосходящие n.

Идеи реализации. Простейший способ решения — последовательно проверять числа $2, 3, \ldots$ на простоту (см. задачу 1-1-28). Сложность алгоритма $O(n^{3/2})$. Для больших значений n лучше воспользоваться алгоритмами типа «решето Эратосфена» (см. задачи 1-7-17, 1-7-18).

Задача 1-1-34. Вычислите, сколько различных упорядоченных наборов длины k можно составить из n различных предметов, т. е. найдите значение функции $A_n^k = n!/(n-k)!$. При реализации учтите возможность переполнения величины n! для корректных значений A_n^k.

Задача 1-1-35. Вычислите, сколько различных наборов длины k можно составить из n различных символов, т. е. найдите $C_n^k = n!/(n-k)!/k!$. Учтите возможность переполнения n! для корректных значений C_n^k.

Задача 1-1-36. Найдите N-е число Фибоначчи (Леонардо Пизанского), определяемое рекуррентной формулой

$$f_1 = 1, \quad f_2 = 1, \quad f_n = f_{n-1} + f_{n-2}, \quad n = 3, 4, \ldots, N.$$

Указание. Возможны реализации на основе цикла, рекурсии (см. задачу 1-9-6) либо явной формулы, основанной на величине $(1+\sqrt{5})/2$, называемой золотым сечением.

```
unsigned long int FibonacciRecurrent(int N);
unsigned long int FibonacciRecurrent(int N) {
  unsigned long fn;
  unsigned long fn_2 = 1, fn_1 = 1;
  int n;
  for (n = 3; n <= N; n++) {
    fn = fn_1 + fn_2;
    fn_2 = fn_1;
    fn_1 = fn;
  }
  return fn;
}

#include <math.h>
unsigned long int FibonacciExplicit(int N);
unsigned long int FibonacciExplicit(int N) {
  double sq5 = sqrt(5.), fn;
  fn = 1. / sq5 *
       (pow((1. + sq5) / 2., (double)N) -
        pow((1. - sq5) / 2., (double)N));
  return (unsigned long int) lround(fn);
}
```

Если на вашей ЭВМ поддерживается полноценный тип **long double**, занимающий 16 байт, то явная формула также может быть реализована с использованием **long double** `sq5, fn;` и библиотечных функций `sqrtl()`, `powl()`, `lroundl()`.

Полезно сравнить эффективность данных подходов, т. е. найти диапазон значений N, при которых функции выдают верный

ответ, и сравнить время работы при больших N. В ОС Linux имеется системная утилита `time`, вызов которой из командной строки в формате `time ./a.exe` позволяет по завершении работы получить информацию о временных ресурсах, затраченных на выполнение программы `a.exe`.

Задача 1-1-37. Проверьте, могут ли три введенных числа a, b, c являться длинами сторон одного треугольника.

Задача 1-1-38. Проверьте, являются ли три введенных числа a_1, a_2, a_3 последовательными элементами арифметической прогрессии, т. е. удовлетворяют ли они формуле $a_n = a_{n-1} + d$, $n = 2, 3$.

Задача 1-1-39. Проверьте, являются ли три введенных числа b_1, b_2, b_3 последовательными элементами геометрической прогрессии, т. е. удовлетворяют ли они формуле $b_n = qb_{n-1}$, $n = 2, 3$.

Задача 1-1-40. Проверьте, являются ли три введенных числа a_i, a_j, a_k последовательными (возможно, неупорядоченными) элементами арифметической прогрессии.

Задача 1-1-41. Проверьте, являются ли три введенных числа b_i, b_j, b_k последовательными (возможно, неупорядоченными) элементами геометрической прогрессии.

Задача 1-1-42. Найдите с помощью алгоритма Евклида наибольший общий делитель положительных чисел m и n, т. е. $nod = нод(m, n)$.

Указание. Алгоритм Евклида основан на равенстве $нод(m, n) = нод(m - n, n)$, $m > n > 0$.

```
unsigned int E1(unsigned int m, unsigned int n);
unsigned int E1(unsigned int m, unsigned int n){
  unsigned int nod;
  while (1) {
    if (m == n) break;
    if (m > n) {
      m = m - n;
    } else {
      n = n - m;
    }
  }
  nod = n;
  return nod;
}
```

Следующая реализация является оптимизированной версией E1(). В данном случае мы также считаем, что $m, n > 0$.

```
unsigned int E2(unsigned int m, unsigned int n);
unsigned int E2(unsigned int m, unsigned int n){
  unsigned int c, nod;
  if (m < n) {
    c = m;
    m = n;
    n = c;
  }
  while (1) {
    c = m % n;
    if (c == 0) break;
    m = n;
    n = c;
  }
  nod = n;
  return nod;
}
```

Задача 1-1-43. Найдите хотя бы одно решение (x, y) уравнения в целых числах $x \cdot m + y \cdot n = \text{нод}(m, n)$.

Указание. Например, можно реализовать перебор по целым значениям $x = 0, \pm1, \pm2, \ldots$. В общем случае решение ищется методом расширенного алгоритма Евклида (см. задачу 1-7-14).

Задача 1-1-44. Найдите наименьшее общее кратное чисел m и n, т. е. $\text{нок}(m, n)$.

Указание. Примените формулу $\text{нод}(m, n)\text{нок}(m, n) = m \cdot n$.

Язык C, являясь языком общего назначения, предоставляет возможность прямого доступа к ячейкам памяти ЭВМ. Посредством унарного оператора **sizeof()** можно определить размер каждой переменной, т. е. количество байт, выделенных для ее хранения. Унарный оператор **&** позволят вычислить адрес размещения содержимого переменной в оперативной памяти. Если известен адрес размещения переменной и ее тип, то имеется возможность непосредственного доступа к записанным по адресу данными — за это отвечает унарный оператор *. Отметим, что размер переменной определяется ее типом и зависит от архитектуры ЭВМ и настроек компилятора. Адрес переменной в некото-

ром смысле случайная величина: загрузчик исполняемых модулей помещает код программы в не занятое другими процессами адресное пространство, размещая там все необходимые данные.

Следующий код показывает базовые правила работы с адресами переменных целого типа. Работа с указателями других типов реализуется аналогично.

```
1  #include<stdio.h>
2  int main(void){
3    size_t sv, sp; // создать sv, sp типа size_t
4    int v;         // создать v типа int
5    int *p;        // создать p типа int*
6    p = &v;        /* вычислить адрес ячейки v
7                      и присвоить переменной p */
8    *p = 3;        /* ячейке, адрес которой
9                      записан в переменной p,
10                     присвоить 3 */
11   *p = *p + 14;  /* увеличить содержимое ячейки,
12                     адрес которой записан в p,
13                     на 14 */
14   sv = sizeof(v);// сколько байт занимает v
15   sp = sizeof(p);// сколько байт занимает p
16   printf("v = %d, &v = %p", v, (void *)p);
17   printf("sizeof(v) = %zu\n", sv);
18   printf("sizeof(&v) = %zu\n", sp);
19   return 0;
20 }
```

Разберем представленный код. Результатом применения оператора **sizeof()** к произвольному объекту является число, имеющее тип **size_t** — базовый беззнаковый целочисленный тип, предназначенный для хранения размера произвольного объекта, размещенного в оперативной памяти. В строке 3 создаются переменные `sv`, `sp` типа **size_t** для последующего хранения таких величин.

В строке 5 создается переменная с именем `p` типа **int** *. Описание переменной в том числе указывает, что величина `*p` имеет тип **int**. В данную переменную можно записать адрес доступной целочисленной ячейки, как это сделано в строке 6. Здесь в результате применения оператора `&` к переменной `v` типа **int** получена величина типа **int** *, т. е. адрес размещения `v`

в оперативной памяти. По определению результатом применения оператора * к переменной типа **int** * является содержимое соответствующей ячейки типа **int**. Следовательно, в строках 8 и 11 происходит запись в ячейку v. Унарные операторы *, & имеют наивысший приоритет, поэтому в строках 6, 8, 11 скобки опущены.

В строках 14 и 15 вычисляется количество байт, занимаемых соответствующими объектами в оперативной памяти. Оператор **sizeof**() может применяться как к переменной, так и к типу, т. е. допустимо обращение `sp = sizeof(int *)`.

В строках 16—18 полученные величины в соответствии с указанным форматом печатаются на экран. Отметим, что при печати адрес **int** * преобразуется к безликому адресу, т. е. к адресу на так называемый неопределенный тип **void** *. Таким образом, ключевое слово **void** указывает на отсутствие параметра у функции, а описание `void *t;` означает создание переменной с именем t, предназначенной для хранения адреса переменной неопределенного типа (т. е. адреса как безликого числа).

Формально и размер произвольного объекта, и адрес его размещения в памяти ЭВМ являются положительными целыми числами. Однако для правильной работы программного кода подобные переменные необходимо описывать указанным выше способом.

Таким образом, к переменной, предназначенной для хранения адреса, допустимо применять оператор разадресации *, обращаясь с его помощью к соответствующей ячейке с данными. Помимо этого, разрешено прибавлять к адресу и вычитать из него целое число и находить разность двух адресов одного типа (см. раздел 1.3). Получаемый результат определяется типом указателя. Адреса разрешено сравнивать на совпадение (т. е. на равенство).

Реализованные в рассмотренных ранее задачах функции возвращали не более одного значения — это обусловлено правилами языка С. Для «обхода» данного ограничения удобно использовать переменные указательного типа, предназначенные для хранения адресов ячеек. Если в функцию передается адрес некоторой переменной (т. е., условно, номер байта, начиная с которого в памяти ЭВМ хранится ее содержимое), то появляется возможность из функции напрямую записать по указанному адресу нужное

значение. Напомним, что данный механизм используется при взаимодействии с библиотечной функцией `scanf()`.

Задача 1-1-45. Вычислите сумму и произведение целых чисел от 0 до n.

Решение.

```
#include <stdio.h>
int SP(int *pSum,   // адрес переменной для суммы
  int *pProd,       /* адрес переменной
                       для произведения */
  int n);           // длина последовательности
int main(void) {
  int sum, prod;    /* ячейки для хранения
                       суммы и произведения */
  int n, key;       /* длина последовательности
                       и ключ ошибки */
  scanf("%d", &n); // передаем адрес n
  key = SP(&sum, &prod, n); /* передаем адреса
                    ячеек sum, prod и значение n */
  if (key != 0) {
    printf("Ошибка данных: n < 0\n");
  } else {
    printf("sum = %d, prod = %d\n", sum, prod);
  }
  return 0;
}
/* Функция принимает адреса двух целочисленных
   ячеек и одно целое число; возвращаемое
   значение отвечает за обработку ошибок
   входных данных */
int SP(int *pSum, int *pProd, int n) {
  int sum = 0, prod = 1, i;
  if (n <= 0) return -1;
  for (i = 1; i <= n; i++) {
    sum += i;
    prod *= i;
  }
  *pSum = sum;    /* записываем сумму в ячейку,
                  адрес которой хранится в pSum */
  *pProd = prod; /* записываем  произведение
```

```
                        в ячейку, адрес которой
                        хранится в pProd */
  return 0;
}
```

Функция `SP()` подсчитывает сумму и произведение элементов последовательности. Возвращаемое оператором **return** значение равно нулю в случае успешного завершения либо коду ошибки (в данном случае -1). Искомые величины `sum`, `prod` вычисляются внутри функции `SP()`, но становятся известными в `main()`, так как записываются по полученным адресам.

Следующий пример поясняет механизм обмена данными между функциями в языке C.

```
#include<stdio.h>
void Swap(int *x, int *y);
void SwapNo(int c, int d);
int main(void)
{
  int a=1, b=2;
  printf("Init: a = %d; b = %d\n", a, b);
  SwapNo(a,b);
  printf("SwapNo: a = %d; b = %d\n", a, b);
  Swap(&a, &b);
  printf("Swap: a = %d; b = %d\n", a, b);
  return 0;
}
void SwapNo(int c, int d)
{
  int tmp;
  tmp = c; c = d; d = tmp; /* реализован обмен
                              ячеек c, d */
  return;
}
void Swap(int *x, int *y)
{
  int c;
  c = *x; *x = *y; *y = c; /* реализован обмен
                              ячеек (*x), (*y) */
  return;
}
```

Задача 1-1-46. В чем идейная ошибочность следующих синтаксически верных конструкций?

```
void SwapErr1(int *x, int *y);
void SwapErr1(int *x, int *y)
{
  int *c;
  c = x; x = y; y = c;
  return;
}
void SwapErr2(int *x, int *y);
void SwapErr2(int *x, int *y)
{
  int *c;
  *c = *x; *x = *y; *y = *c;
  return;
}
```

Указание. В первом случае меняется содержимое переменных x, y, что не приводит к обмену содержимого ячеек, адреса которых записаны в x, y. Во второй функции в ячейке c хранится «мусорный» адрес некоторой случайной ячейки. Оператор `*c = x` осуществляет запись в эту ячейку. Это нарушает правила доступа к памяти и приводит (в требовательных системах) к аварийной остановке программы с ошибкой сегментации памяти Segmentation fault (core dumped). Добавление строк типа `int z; c = &z;` делает код рабочим.

Задача 1-1-47. Найдите на отрезке [a, b] два целых числа, имеющих наибольшее и наименьшее количество делителей.

Указание. Реализуйте функцию с прототипом

```
int FindMN(double a, double b,// границы отрезка
  int *pMin,  // адреса ячеек, созданных в main()
  int *pMax); // для записи найденных чисел
```

Возвращаемое значение либо 0 (в случае успеха), либо код ошибки (например, при $b < a$).

Объясните выражение $0! = 1$ (*тест «Вы математик или программист?»*).

1.2. Обработка последовательностей

Общая постановка задач данного раздела выглядит следующим образом: имеется набор (последовательность) однотипных данных, например чисел, и требуется вычислить некоторую характеристику набора (функцию от этих данных). Специфика задачи состоит в том, что мы не можем сохранить всю последовательность в памяти, поскольку общее количество элементов может оказаться существенно больше доступного места. Поэтому нужно построить алгоритм, вычисляющий необходимую характеристику за один проход (просмотр) последовательности. Допускается вычисление и сохранение лишь ограниченного набора промежуточных значений, на основе которых в любой момент можно найти искомую величину. Обычно алгоритмы решения подобных задач сводятся к реализации некоторых рекуррентных соотношений, пересчитываемых при последовательной обработке входных данных.

Задача 1-2-1. С клавиатуры вводятся положительное целое число n, а далее последовательность, состоящая ровно из n целых чисел. Требуется найти сумму элементов последовательности.

Решение.

```
#include <stdio.h>
int main(void) {
  int x, sum;
  int i, n;
  printf("---Вычисление суммы целых чисел---\n");
  printf("\t Введите количество\n");
  printf("элементов последовательности n\n");
  scanf("%d", &n);
  if (n <= 0) {
    printf("Последовательность пустая\n");
    return -1;
  }
  printf("Вводите элементы  по одному\n");
  sum = 0;
  for (i = 0; i < n; i++) {
    printf("Введите элемент x_%d \n", i);
    scanf("%d", &x);
    sum += x;
```

```
  }
  printf("\t Последовательность \n");
  printf("содержит %d чисел \n", n);
  printf("Их сумма равна %d \n", sum);
  return 0;
}
```

Задача 1-2-2. С клавиатуры вводится последовательность целых чисел, длина которой заранее не известна. В конце последовательности добавлен ноль, означающий конец ввода (при этом ноль не считается элементом последовательности). Требуется найти и распечатать количество элементов последовательности и их произведение либо выдать сообщение, что последовательность пустая.

Указание. Например, можно использовать следующую конструкцию:

```
#include <stdio.h>
int main(void) {
  int n, x, prod;
  prod = 1;
  n = 0; // число элементов последовательности
  while (1) {
    scanf("%d", &x);
    if (x == 0) break;
    n++;
    prod *= x;
  }
  printf("Последовательность");
  printf("содержит %d чисел \n", n);
  if (n > 0) {
    printf("Их произведение равно %d\n", prod);
  }
  return 0;
}
```

Задача 1-2-3. С клавиатуры вводится последовательность действительных чисел, длина которой заранее неизвестна. Ввод завершается либо комбинацией клавиш Ctrl + d, либо набором произвольных нечисловых данных. Требуется найти и распечатать произведение элементов последовательности.

Указание. Функция `scanf()` возвращает количество успешно считанных объектов по запрашиваемому формату — это позволяет по крайней мере частично контролировать корректность входных данных. Выделим непосредственное вычисление произведения в отдельную функцию:

```
#include<stdio.h>
int Prod(double *p);
int Prod(double *p) {
  int k;
  double x, prod = 1.;
  int n = 0;//число элементов последовательности
  while (1) {
    k = scanf("%lf", &x);
    if (k == 1) {//x содержит очередной элемент
        n++;
        prod *= x;
    } else {//если k<=0, то x считать не удалось
      break;
    }
  }
  *p = prod;
  return n;
}
```

При решении предлагаемых далее задач рекомендуется чередовать рассмотренные варианты 1-2-1, 1-2-2, 1-2-3 считывания элементов последовательности.

Замечание. Процедура ввода данных с клавиатуры является исключительно трудоемкой и утомительной (особенно на этапе тестирования программы), поэтому далее будем считать, что обрабатываемая последовательность хранится в некотором текстовом файле, например с именем `input.txt`. В простейшем варианте для работы с этими данными можно в момент запуска исполняемого файла `tsk.exe` с кодом нашей программы переопределить поток стандартного ввода, назначив источником данных файл `input.txt`. Это можно сделать следующей консольной командой:

```
./tsk.exe < input.txt
```

В результате при очередном вызове функции `scanf` из кода `tsk.exe` данные будут последовательно поступать из файла

`input.txt`. Если же выдаваемый на экран функцией `printf` результат требуется сохранить в файле `out.res`, то это можно сделать с помощью вызовов

```
./tsk.exe > out.res
```

либо

```
./tsk.exe >> out.res
```

В первом варианте исходное содержимое файла `out.res` затирается, а во втором информация будет последовательно добавляться в конец. Также допустимо одновременное перенаправление каналов ввода и вывода:

```
./tsk.exe < input.txt > out.res
```

Однако такие способы являются достаточно специфическими и частными, и более правильным было бы заставить программу работать непосредственно с требуемым файлом (см. задачу 1-2-28).

Для тестирования предлагаемых далее для самостоятельной реализации однопроходных алгоритмов полезно подготовить набор текстовых файлов с числовыми последовательностями разного типа. Например, в задаче может потребоваться, чтобы входная последовательность была возрастающей, знакопеременной, являлась арифметической или геометрической прогрессией либо последовательностью случайных чисел. Реализация генератора подобной последовательности приводится в задаче 1-2-68.

Задача 1-2-4. Составьте программу для вычисления математического ожидания, т. е. среднего арифметического $M = \frac{1}{n}\sum_{i=1}^{n} x_i$ элементов числовой последовательности.

Задача 1-2-5. Вычислите дисперсию $D = \frac{1}{n}\sum_{i=1}^{n}(x_i - M)^2$ элементов числовой последовательности, т. е. среднее квадратичное отклонение от среднего арифметического.

Указание. Раскрыв скобки в требуемом выражении, можно заметить, что дисперсия явно выражается через сумму чисел s, сумму их квадратов s2 и общее количество элементов n следующим образом: $D = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - M^2$, а величины s, s2, n легко вычисляются по рекуррентным формулам за один просмотр последовательности.

Задача 1-2-6. Вычислите среднее геометрическое $G = \left(\prod_{i=1}^{n} x_i\right)^{1/n}$ элементов числовой последовательности с положительными членами.

Задача 1-2-7. Вычислите среднее гармоническое $A_{-1} = n / \sum_{i=1}^{n} \frac{1}{x_i}$ элементов числовой последовательности с положительными членами.

Задача 1-2-8. Подсчитайте количество положительных, отрицательных и нулевых элементов целочисленной последовательности.

Решение.

```
#include <stdio.h>
int SignCount(int *neg, int *zero, int *pos);
int main(void) {
  int n = 0, z = 0, p = 0;
  int res;

  res = SignCount(&n, &z, &p);

  if (res == 0) {
    printf("пустая последовательность\n");
    return -1;
  }
  printf("Всего элементов %d\n; из них\n", res);
  printf(" отрицательных %d\n", n);
  printf(" нулевых       %d\n", z);
  printf(" положительных %d\n", p);
  return 0;
}
int SignCount(int *neg, int *zero, int *pos) {
  double x;
  int n = 0, z = 0, p = 0, cnt = 0;
  while (scanf("%lf", &x) == 1) {
    if (x < 0) {
      n++;
    } else {
      if (x > 0) {
        p++;
      } else {
        z++;
      }
    }
    cnt++;
```

```
  }
  *neg = n;
  *zero = z;
  *pos = p;
  return cnt;
}
```

Замечание. Как отмечалось ранее, с точки зрения вычислительных процедур не стоит сравнивать вещественные числа на точное равенство/неравенство, т. е. `x == y` или `x != y` в нотации языка C. Поэтому в примере мы постарались избежать явного сравнения с нулем, отказавшись от «прозрачного» кода типа

```
if (x == 0) z++;
```

Во всех последующих задачах также предполагается, что сравнение на равенство вещественных чисел `x`, `y` является недопустимым. Корректными считаются проверки типа

```
fabs(x - y) < eps
```

или

```
fabs(x - y)/(1. + fabs(x)) < eps.
```

Таким образом, равенство понимается «с точностью `eps`», являющегося параметром программы.

Если в условии не сказано иное, то входные данные являются целочисленными и позволяют корректно вычислить ответ. Окончание ввода последовательности регламентируется одним из рассмотренных ранее способов, см. задачи 1-2-1, 1-2-2, 1-2-3, например, в зависимости от значения остатка, полученного при делении номера задачи на 3.

Задача 1-2-9. Найдите, сколько элементов последовательности кратно семи, но не кратно трем, а также вычислите их произведение.

Указание. Например,

```
#include<stdio.h>
int Calc(int *p, int *num);
int Calc(int *p, int *num){
  int n = 0, m7not3 = 0;
  int x, prod = 1;
  while (scanf("%d", &x) == 1){
    n++;
    if ( (x % 7 == 0) && (x % 3 != 0) ){
```

```
      m7not3++;
      prod *= x;
    }
  }
  *p = prod;
  *num = m7not3;
  return n;
}
```

Задача 1-2-10. Найдите количество четных и количество нечетных элементов последовательности.

Задача 1-2-11. Найдите количество четных элементов последовательности, значение которых больше десяти и не превосходит ста, а также их процент (с точностью до десятых долей) от общего числа элементов.

Задача 1-2-12. Найдите количество элементов последовательности, равных трем, количество элементов, равных пяти, и количество элементов, равных семи.

Задача 1-2-13. Найдите количество элементов последовательности, кратных трем, количество элементов, кратных пяти, и количество элементов, кратных семи.

Задача 1-2-14. Найдите количество нечетных элементов последовательности, значение которых больше нуля, а также порядковый номер последнего такого элемента.

Задача 1-2-15. Найдите количество элементов последовательности, кратных трем, значение которых меньше ста, а также порядковый номер первого такого элемента.

Задача 1-2-16. Найдите количество элементов последовательности, делящихся на пять и имеющих в записи более трех цифр, а также номер последнего такого элемента.

Задача 1-2-17. Найдите количество элементов последовательности, заканчивающихся нулем, значение которых меньше тысячи, а также номер первого такого элемента.

Задача 1-2-18. Найдите количество элементов последовательности, для которых остаток от деления на три равен остатку от деления на пять, а также значение последнего такого элемента.

Задача 1-2-19. Найдите количество четных элементов последовательности, для которых остаток от деления на пять ровно на единицу отличается от остатка от деления на семь, а также значение последнего такого элемента.

Задача 1-2-20. Найдите количество нечетных элементов последовательности, для которых остаток от деления на пять не более чем на два отличается от остатка от деления на семь, а также номер последнего такого элемента.

Задача 1-2-21. Определите, сколько раз первое считанное число x далее встречается в последовательности.

Задача 1-2-22. Считайте первое число x, а далее — элементы последовательности. Определите порядковый номер первого элемента последовательности, равного x.

Задача 1-2-23. Считайте первое число x, а далее — элементы последовательности. Определите порядковый номер последнего элемента последовательности, равного x.

Задача 1-2-24. Определите значения минимального и максимального элементов последовательности.

Задача 1-2-25. Определите порядковый номер первого числа, равного максимуму из всех элементов последовательности.

Задача 1-2-26. Определите количество чисел, равных минимуму из всех элементов последовательности.

Задача 1-2-27. Определите номер первого и последнего минимального элемента последовательности.

При решении следующих задач необходимо считывать хранящиеся в файле данные напрямую из программного кода, выполняя стандартные процедуры открытия, чтения/записи и закрытия файла.

Задача 1-2-28. В файле `input.txt` задана последовательность действительных чисел $x_1, x_2, \ldots, x_n$ неизвестной длины (возможно, пустая). Требуется найти и распечатать значение

$$s = x_1 - x_2 + \ldots + (-1)^{k+1} x_k \ldots,$$

прочитав все числа, записанные в файле. Если последовательность пуста, то надо выдать соответствующее сообщение.

Приведем полное решение этой задачи как некий образец для дальнейшего.

Решение.

```
#include <stdio.h>
/*---Прототип функции обработки файла---*/
int Process(FILE* f, double* res);
int main(void) {
  double result;
  int retcode;
```

```
  FILE* f;
  f = fopen("input.txt", "r");
  if (f == NULL) {
    printf("Ошибка открытия файла\n");
    return -1;
  }
  retcode = Process(f, &result);
  if (retcode < 0) {
    printf("В файле нет данных \n");
  } else {
    printf("Результат: %f\n", result);
  }
  fclose(f);
  return retcode;
}
int Process(FILE* f, double* res) {
  double s = 0, sign = 1, x;
  int key = -1, n;
  for (n = 0; fscanf(f, "%lf", &x) == 1; n++) {
    s += sign * x;
    sign = -sign;
    key = 0;
  }
  *res = s;
  return key;
}
```

Замечание. Переменная `f` хранит *адрес* ячейки оперативной памяти, где размещен объект типа `FILE`, являющийся структурой (описание типа `FILE` имеется в `stdio.h`). Данная структура, создаваемая и заполняемая функцией `fopen`, содержит необходимую информацию для работы функции `fscanf`, т. е. для чтения данных из файла `input.txt`. Приведенный вызов

```
f = fopen("input.txt", "r");
```

является компактной формой записи следующей конструкции:

```
const char *fname = "input.txt";
const char *fstr = "r";
f = fopen(fname, fstr);
```

Функция `fopen` принимает два указателя: на строку с именем открываемого файла и на строку с типом доступа. Это позволяет при необходимости считать имя открываемого файла с клавиатуры (см. задачу 1-3-1, а также раздел 1.6). Если функция `fopen` присвоила переменной `f` значение именованной константы `NULL`, т. е. указателя на нулевую ячейку, то требуемый доступ к файлу обеспечить не удалось.

Функция `Process()` подсчитывает количество `n` элементов последовательности (формально это значение не требуется, но указанный прием полезен в последующих задачах) и возвращает содержимое переменной `key`, значение которой равно нулю в случае успешного завершения либо коду ошибки (в данном случае -1). Искомая величина `s` вычисляется внутри функции `Process()` и становится известной в `main()`, так как записывается по адресу, хранящемуся в `res` и содержащему адрес ячейки `result`.

Задача 1-2-29. В файле `input.dat` записаны действительные числа, разделенные некоторым количеством пробелов, табуляций, по несколько символов в строке. Требуется переписать их в файл `output.dat` в виде столбца шириной 20 символов в формате 15 знаков после запятой, сохранив исходный порядок.

Решение.

```
#include <stdio.h>
int FormatFile(FILE *fOut, FILE *fIn);
int FormatFile(FILE *fOut, FILE *fIn) {
  double x;
  while (fscanf(fIn, "%lf", &x) == 1) {
    fprintf(fOut, "%20.15lf\n", x);
  }
  return 0;
}
int main(void) {
  FILE *fOut, *fIn;
  fIn = fopen("input.dat", "r");
  if (fIn == NULL) {
    return -1;
  }
  fOut = fopen("output.dat", "w");
  if (fOut == NULL) {
```

```
    return -2;
  }
  FormatFile(fOut, fIn);
  fclose(fIn);
  fclose(fOut);
  return 0;
}
```

Замечание. При запуске exe-кода автоматически инициируются три стандартных потока, т. е. три указателя типа `FILE *` с именами `stdin`, `stdout`, `stderr`. Это стандартный поток ввода, поток вывода и поток сообщений об ошибках. Как следствие, в данном случае вызовы типа

```
FormatFile (stdout,fIn);
```

либо

```
fOut = stdout; FormatFile (fOut,fIn);
```

обеспечат печать чисел на экран.

Задача 1-2-30. Распечатайте значения и порядковые номера трех элементов последовательности, имеющих наибольшие по модулю значения.

Задача 1-2-31. Среди элементов последовательности найдите пару, в которой порядковые номера элементов отличаются не менее чем на три, а сумма элементов имеет максимальное значение среди сумм подобных пар.

Указание. Будем считать, что после обработки k-го числа, $k \geqslant 4$, в переменных a1, a2, a3, a4 хранятся четыре последних введенных элемента последовательности, в переменной aMax — наибольший из первых $k-3$ элементов, в переменной sMax — искомая сумма. Например, по завершении четвертого шага алгоритма переменная aMax равна первому элементу последовательности, т. е. a1, а искомая сумма sMax равна aMax + a4. На каждом последующем шаге алгоритма переприсваиваем $a1 = a2$, $a2 = a3$, $a3 = a4$; считываем очередной элемент в a4; пересчитываем aMax, т. е. заменяем aMax на максимум из $\{aMax, a1\}$; пересчитываем sMax, т. е. заменяем sMax на максимум из величин $\{sMax, aMax + a4\}$.

Задача 1-2-32. Определите величину наибольшего отклонения элементов последовательности от их среднего арифметического.

Задача 1-2-33. Определите, все ли элементы последовательности равны между собой.

Решение. Реализуем функцию, возвращающую следующие значения:

1, если последовательность постоянна,

−1, если последовательность содержит различные элементы,

0, если в файле недостаточно данных.

```
int type(FILE* fin) {
  int x, y;
  int key = 0;
  if (fscanf(fin, "%d", &x) != 1) return key;
  key = 1;
  while (fscanf(fin, "%d", &y) == 1) {
    if (x != y) key = -1;
    x = y;
  }
  return key;
}
```

Задача 1-2-34. Определите, является ли последовательность действительных чисел строго возрастающей, строго убывающей или ни той, ни другой.

Решение. Реализуем функцию, возвращающую следующие значения:

2, если последовательность является строго возрастающей,

1, если последовательность является строго убывающей,

0, если последовательность не является ни убывающей, ни возрастающей,

−1, если в файле недостаточно данных.

```
int type(FILE* fin) {
  double x, y;
  int key = -1;
  if (fscanf(fin, "%lf%lf", &x, &y) != 2)
    return key;
  key = 0;
  if (y < x) key = 1;
```

```
  if (y > x) key = 2;
  x = y;
  while (fscanf(fin, "%lf", &y) == 1) {
    if ((y < x) && (key == 2)) key = 0;
    if ((y > x) && (key == 1)) key = 0;
    x = y;
  }
  return key;
}
```

Задача 1-2-35. Составьте программу для проверки того, является ли последовательность арифметической прогрессией.

Задача 1-2-36. Составьте программу для проверки того, является ли последовательность геометрической прогрессией.

Задача 1-2-37. Обозначим элементы последовательности через x_k, $k = 1, 2, \ldots$. Считайте первые четыре числа a, b, c, d и проверьте, удовлетворяют ли последующие элементы последовательности рекуррентному соотношению $ax_k + bx_{k+1} + cx_{k+2} = d$.

Задача 1-2-38. Определите количество чисел в последовательности, которые больше предшествующего числа.

Задача 1-2-39. Пусть последовательность является неубывающей. Определите количество различных элементов этой последовательности.

Задача 1-2-40. Пусть последовательность является неубывающей. Считайте первое число n, а далее — всю последовательность. Определите количество элементов последовательности, которые появляются более n раз.

Задача 1-2-41. Определите, сколько раз во входной последовательности встречается подпоследовательность $1, 2, 3, \ldots, 100$.

Задача 1-2-42. Определите длину наибольшего постоянного участка, т. е. максимальное количество подряд идущих элементов с одним и тем же значением.

Задача 1-2-43. Считайте первое число n, а далее — элементы последовательности. Определите количество постоянных участков последовательности, имеющих длину не меньше n.

Задача 1-2-44. Определите длину постоянного участка последовательности с наибольшей суммой элементов.

Задача 1-2-45. Определите общее количество элементов в постоянных участках целочисленной последовательности.

Задача 1-2-46. Определите, сколько элементов последовательности не содержится в постоянных участках длины 2 и более.

Задача 1-2-47. Составьте программу для вычисления математического ожидания элементов числовой последовательности, учитывая из каждого постоянного участка только один элемент.

Задача 1-2-48. Определите длину возрастающего участка последовательности с наибольшим числом элементов.

Задача 1-2-49. Определите длину возрастающего участка последовательности с наибольшей суммой.

Задача 1-2-50. Считайте первое число n, а далее — элементы последовательности. Определите количество невозрастающих участков последовательности, имеющих длину не меньше n.

Задача 1-2-51. Найдите сумму четных элементов во всех возрастающих участках целочисленной последовательности.

Задача 1-2-52. Определите, каких участков в последовательности больше: возрастающих или невозрастающих.

Элемент последовательности назовем локальным максимумом (минимумом), если он *строго* больше (меньше) своих соседей. Элемент назовем локальным экстремумом, если он является либо локальным минимумом, либо локальным максимумом. Для упрощения допустимо считать, что первый и последний элементы не могут быть локальными экстремумами.

Задача 1-2-53. Определите наибольшее расстояние между локальными максимумами (минимумами) элементов последовательности.

Задача 1-2-54. Найдите среднее арифметическое локальных экстремумов последовательности.

Задача 1-2-55. Определите и напечатайте локальные экстремумы с соседними точками. Каждая тройка печатается с новой строки.

Задача 1-2-56. Определите и напечатайте все отрезки возрастания последовательности. Каждый участок печатается с новой строки.

Задача 1-2-57. Определите и напечатайте все отрезки монотонности последовательности с явным указанием типа монотонности. Каждый отрезок печатается с новой строки.

Задача 1-2-58. Пусть последовательность чисел представляет собой коэффициенты многочлена, расположенные в порядке возрастания степеней. Считайте первое число x, а далее —

элементы последовательности (т. е. коэффициенты). Вычислите значение многочлена и его производной в точке x.

Идеи реализации. Обозначим хранящиеся в файле коэффициенты через $a_0, a_1, \ldots, a_n$. Тогда

$$P_n(x) = a_0 + a_1x^1 + \ldots + a_nx^n —$$

значение многочлена в точке x,

$$D_n(x) = a_1 + 2a_2x + \ldots + na_nx^{n-1} —$$

значение его производной. При этом переменную xk, содержащую величину x^k, удобно на очередном шаге перевычислять рекуррентно xk = xk * x.

Задача 1-2-59. Пусть последовательность чисел представляет собой коэффициенты многочлена, расположенные в порядке убывания степеней. Считайте первое число x, а далее — элементы последовательности (т. е. коэффициенты). Вычислите значение многочлена и его производной в точке x.

Идеи реализации. Воспользуйтесь схемой Горнера вычисления многочлена и преобразуйте ее в рекуррентные соотношения, связывающие значения многочлена и его производной. Если обозначить через a_k k-й элемент последовательности коэффициентов, через

$$P_k(x) = a_0x^k + a_1x^{k-1} + \ldots + a_k —$$

значение многочлена k-й степени в точке x, через

$$D_k(x) = ka_0x^{k-1} + (k-1)a_1x^{k-2} + \ldots + a_{k-1} —$$

значение производной многочлена $P_k(x)$, то данные рекуррентные соотношения будут иметь следующий вид:

$$P_0(x) = a_0, \quad P_k(x) = xP_{k-1}(x) + a_k, \quad k > 0;$$
$$D_0(x) = 0, \quad D_k(x) = P_{k-1}(x) + xD_{k-1}(x), \quad k > 0.$$

Задача 1-2-60. Для последовательности $a_1, a_2, \ldots, a_N$ определите максимальную сумму подряд идущих элементов

$$S_{[k_1,k_2]} = a_{k_1} + \ldots + a_{k_2}.$$

Указание. Реализуйте алгоритм Д. Кадана: если текущая сумма элементов подпоследовательности стала отрицательной, то выгоднее начинать новую подпоследовательность, чем продолжать старую. Также полезно разобрать следующий подход. При

наличии вспомогательного массива s: $s[0] = 0$, $s[k] = a_1 + \ldots + a_k$, $k = 1, \ldots, N$, имеем $S_{[k_1,k_2]} = s[k_2] - s[k_1 - 1]$. Для максимизации искомой суммы при текущем значении k_2 достаточно минимизировать величину $s[k_1 - 1]$. Реализовать алгоритм можно без явного создания массива s, имея только текущее значение $s[k_2]$ и минимум из величин $s[k_1 - 1]$ по всем $k_1 = 1, 2, \ldots, k_2$.

Задача 1-2-61. Найдите границы k_1 и k_2 подпоследовательности из задачи 1-2-60.

Задача 1-2-62. Для последовательности положительных целых чисел найдите наибольшее произведение двух ее элементов с разными номерами, делящееся на 17. Распечатайте найденное значение, соответствующие элементы и их порядковые номера либо сообщение, что такого числа составить нельзя.

Указание. Так как 17 является простым числом, то искомое значение может быть получено в результате произведения двух наибольших элементов последовательности (имеющих разные номера), из которых хотя бы одно делится на 17.

Задача 1-2-63. Для последовательности положительных целых чисел найдите наибольшее произведение двух элементов с разными номерами, делящееся на 51.

Указание. Так как $51 = 17 \cdot 3$, то для получения искомого произведения необходимо найти следующие различные (в смысле порядкового номера) элементы последовательности: самый большой элемент, самый большой из кратных 51, самый большой из кратных 17, самый большой из кратных 3.

Задача 1-2-64. Для последовательности положительных целых чисел найдите наибольшее число `R`, делящееся на 17, которое можно получить в результате умножения двух элементов последовательности при условии, что их порядковые номера отличаются не менее чем на пять единиц.

Указание. Создать массив `A[6]` для хранения последних шести считанных элементов последовательности и переменную `R` для записи искомого результата. Завести переменные `M17` и `M` для хранения наибольшего делящегося на 17 элемента и наибольшего (отличного от `M17`) элемента из обработанной к данному моменту части последовательности. На очередном шаге сдвинуть элементы массива на одну позицию влево, «выталкивая» `A[0]`; считать следующий элемент в `A[5]` и далее найти новые значения `M17`, `M` и `R`. Эффективность алгоритма можно несколько повысить,

«замкнув» A[] в кольцо, что позволит на каждом шаге избежать процедуры сдвига элементов массива.

Задача 1-2-65. Для последовательности различных положительных целых чисел найдите количество пар, которые в сумме дают число, делящееся на 17.

Указание. Завести массив A[17] и сохранить в его i-й ячейке количество элементов последовательности, дающих при делении на 17 остаток i.

Задача 1-2-66. Последовательность представляет собой координаты точек на плоскости, т. е. пары чисел (x_i, y_i), $i = 1, 2, \ldots$. Найдите координаты вершин прямоугольника с минимальной площадью и сторонами, параллельными осям координат, содержащего все считанные точки.

Задача 1-2-67. Последовательность представляет собой координаты точек на плоскости, т. е. пары чисел (x_i, y_i), $i = 1, 2, \ldots$. Вычислите, сколько можно образовать отрезков с концами из этого множества, таких что ни один из концов отрезка не лежит на осях координат и отрезок имеет ровно одну точку пересечения с осями.

Задача 1-2-68. Напишите генератор числовой последовательности, обладающей одним из следующих свойств: последовательность возрастающая, последовательность знакопеременная, элементы последовательности составляют арифметическую или геометрическую прогрессию, последовательность составлена из случайных чисел.

Решение. Приведем реализацию генератора, заполняющего файл целыми псевдослучайными числами из диапазона от 0 до $2^{31} - 1$. Для получения детерминированной монотонно возрастающей последовательности нужно положить key = 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
typedef unsigned int uint32_t;
/* Заполнение файла n числами  0<=x<val: */
void FillDataFile(FILE *f, uint32_t n, int key,
                  uint32_t val);
/* Псевдослучайный генератор: */
uint32_t MyRand(void);
/* Инициализация генератора: */
```

```
void MySrand(uint32_t seed);
/* Следующее значение: */
uint32_t MyNext(uint32_t i, uint32_t cur);
int main(void) {
  char fname[256];  //имя выходного файла
  uint32_t n;             //количество чисел
  int key = 0;       //тип последовательности:
  //0 - псевдослучайная; 1 - детерминированная
  uint32_t val; //верхняя граница диапазона
  FILE *fout;
  printf("Введите имя файла ");
  scanf("%s", fname);
  printf(" количество элементов ");
  scanf("%u", &n);
  printf(" тип последовательности ");
  scanf("%d", &key);
  printf(" строго верхнюю границу ");
  scanf("%u", &val);
  fout = fopen(fname, "w");
  if (fout == NULL) {
    printf("Ошибка открытия файла %s\n", fname);
    return -1;
  }
  FillDataFile(fout, n, key, val);
  fclose(fout);
  return 0;
}
void FillDataFile(FILE *f, uint32_t n, int key,
                  uint32_t val) {
  uint32_t i;
  uint32_t x = 1L;
  if (key == 0) MySrand((uint32_t)time(NULL));
  for (i = 0; i < n; i++) {
    if (key == 0) {
      x = MyRand();
    } else {
      x = MyNext(i, x);
    }
    x = x % val;
    fprintf(f, "%u\n", x);
```

```
  }
}
uint32_t MyNext(uint32_t i, uint32_t cur) {
/* детерминированная последовательность
   по явной рекуррентной формуле: */
  return i * i + cur;
}
uint32_t globalNext = 1U;
void MySrand(uint32_t seed) {
  globalNext = seed;
  return;
}
uint32_t MyRand(void) {
/* конгруэнтный линейный генератор Парка--Миллера
   для целых псевдослучайных чисел: */
  uint32_t A = 16807U;        // 7^5
// long int M = 2147483647U;  // 2^31 - 1
  uint32_t Q = 12773U;
  uint32_t R = 2836U; // M = A * Q + R
  // globalNext = (A * globalNext) % M;
  globalNext = (globalNext % Q) * A
         + (globalNext / Q) * R;
  return globalNext;
}
```

Замечание. Так как при каждом обращении функция `MyRand()` возвращает число из диапазона от 0 до `M - 1`, вычисленное по конкретной аналитической формуле, то при каждом запуске программы будет получена одна и та же (циклически повторяющаяся) последовательность чисел. Это бывает полезно, если нужно повторять расчеты с одинаковыми наборами данных. В данном случае значения `A`, `M` выбраны так, что алгоритм `next = (A * next) % M` генерирует последовательность, имеющую почти максимальный (т. е. близкий к M) период порядка $2{,}14 \cdot 10^9$. Изложенная модификация Л'Экюйе связана с возникающими в процессе его реализации переполнениями. С помощью функции `MySrand(int)` можно сменить стартовую точку последовательности, например, сделав ее зависящей от текущего астрономического времени. В данном примере функция

`time(NULL)` возвращает целое число, равное количеству секунд, прошедших с начала эры Unix (см. с. 69).

Данные функции являются некоторыми аналогами `rand()` и `srand()`; прототипы этих функций описаны в заголовочном файле `stdlib.h`, содержащем также именованную константу `RAND_MAX`, определяющую диапазон `[0, RAND_MAX]` значений `rand()`. Для генерации целочисленной последовательности из `[a,b]` можно сделать замену `a + rand() % (b - a + 1)`.

Для модельных расчетов также удобно использовать базовый генератор регистров сдвига Джорджа Марсальи (для важных расчетов — его продвинутую версию `xorshit128`):

```
uint32_t w;
uint32_t xorshift32(void){
  uint32_t x = w;
  x ^= x << 13;
  x ^= x >> 17;
  x ^= x << 5;
  return w = x;
}
```

Если необходимо получить «случайную» последовательность действительных чисел $x_k \in [0,1)$, то лучше применить генератор Фибоначчи типа $x_k = x_{k-17} - x_{k-5}$. Значения данной последовательности зависят от стартовых чисел $x_1, \ldots, x_{17}$, выбору которых следует уделить особое внимание. Если на очередном шаге получаем $x_k < 0$, то значение x_k следует заменить на $x_k + 1$. В ответственных расчетах рекомендуется применять генераторы с длинными лагами (запаздываниями), например $x_k = x_{k-97} - x_{k-33}$.

Задача 1-2-69. Напишите программу заполнения файла псевдослучайной неубывающей последовательностью, состоящей из n положительных чисел.

Правдивая история (на контрольной тестового типа). Все решают, а один бросает кубик и проставляет выпавший ответ. Все сдали свои работы, а он все бросает.
— Вы чего не сдаете?
— Да вот решил перепроверить ответы, а один все никак не сходится...

1.3. Работа с массивами

В данном разделе собраны простейшие задачи обработки, анализа и преобразования данных, организованных в одномерный массив. Прежде чем формулировать условия задач, рассмотрим базовые конструкции, используемые для создания и заполнения массива.

В качестве первого примера приведем фрагмент программы, сохраняющей в массиве сто чисел, заданных следующим рекуррентным соотношением:

$$x_0 = 1, \quad x_1 = 2, \quad x_{k+1} = 2x_k - x_{k-1}, \quad k \geqslant 1.$$

Будем считать, что эти числа представлены типом **int**.

```
int x[100];
const int NMAX = 100;
x[0] = 1;
x[1] = 2;
for (int k = 2; k < NMAX; k++) {
  x[k] = 2 * x[k - 1] - x[k - 2];
}
```

Другим вариантом может быть заполнение массива значениями, читаемыми с клавиатуры (либо из файла). Создадим массив некоторой заведомо достаточной длины, а при чтении будем контролировать реально получающееся количество элементов. Соответствующий фрагмент программы может выглядеть, например, так (рассматриваем массив вещественных чисел):

```
#define NMAX 1000
double mas[NMAX];
for (int n = 0; n < NMAX; n++) {
  if (scanf("%lf", &mas[n]) != 1) break;
}
```

По окончании цикла переменная **n** содержит фактическую длину массива.

В качестве еще одного примера работы с массивами приведем функцию **void PrnMas(double *, int)** печати на экран содержимого ячеек массива и соответствующих им адресов.

```
void PrnMas(double *mas, int ncur){
//Равносильно: void PrnMas(double mas[], int ncur)
```

```
  int n;
  for (n = 0; n < ncur; n++) {
    printf("n = <%10d>\n", n);
    printf("Val(mas[n]) = <%20.16lf>\n", mas[n]);
    printf("Adr(mas[n]) = <%20p>\n",
                                  (void*)(mas + n));
  }
  return;
}
```

При вызове

```
PrnMas(mas, n);//Равносильно: PrnMas(&mas[0],n);
```

в функцию будет передан адрес нулевого элемента массива и его длина. Это позволит не только распечатать, но и (при необходимости) изменить содержимое каждой ячейки `mas[n]`.

Отметим, что максимальный размер созданных таким образом локальных статических массивов существенно меньше размера доступной оперативной памяти, так как они размещаются в особом сегменте оперативной памяти со структурой стека заранее фиксированной глубины. Поэтому более эффективным является следующий подход. Будем считать, что первое считанное с клавиатуры число есть длина массива. В этом случае динамическое создание массива и работу с ним можно осуществить следующим образом.

```
#include <stdio.h>
#include <stdlib.h> /* malloc, free, exit */
int main(void) {
  double *mas = NULL; // адрес начала массива
  unsigned int n, nmax = 0, ncur;
  int key;
  key = scanf("%u", &nmax);
  if (key != 1 || nmax <= 0) {
    exit(1);
  }
  mas = (double *)malloc(nmax * sizeof(double));
  if (mas == NULL) {
    printf("Не удалось выделить %lu байт\n",
           nmax * sizeof(double));
```

```
16      exit(2);
16    }
17    for (n = 0; n < nmax; n++) {
18      key = scanf("%lf", mas+n);
19      if (key != 1) break;
20    }
21    ncur = n;
22    for (n = 0; n < ncur; n++) {
23      printf("%lf ", *(mas+n)); //контрольная печать
24    }
26    printf("\n");
26
27    {/*
28      рабочая часть программы
29     */}
30    free(mas);
31    return 0;
32  }
```

Разберем представленный код. В файле **stdlib.h** (от англ. standard library header) содержатся прототипы используемых далее функций **malloc** (от англ. memory allocate), **free** и **exit**. В строке 4 заводится переменная **mas** для хранения начального адреса области памяти, которая будет выделена под элементы массива. Значение инициируется именованной константой **NULL**, т. е. адресом формально запрещенной для работы нулевой ячейки. В большинстве C-компиляторов константа **NULL** определена как **#define NULL ((void*) 0)**. Если после выполнения оператора строки 7 окажется, что число **nmax** не удалось прочитать или соответствующее значение меньше либо равно нулю, то выполнение программы следует прекратить (строка 9).

В строке 11 вызывается функция **malloc**. На вход функция получает требуемое количество байт и в случае успеха возвращает адрес начальной ячейки выделенного блока памяти. Так как возвращаемое функцией значение имеет тип **void ***, то перед присвоением его рекомендуется преобразовать к требуемому типу. В данном случае за это отвечает унарный оператор **(double *)**. Если выделить необходимую память не удалось, то функция **malloc** возвращает значение **NULL**, т. е. адрес нулевой ячейки. В этом случае (строка 12) дальнейшее выполнение про-

граммы считается неразумным. Формально подобные проверки всегда следует предусматривать в программном коде, хотя на данный момент в большинстве Linux-систем процесс выделения памяти функционирует существенно сложнее: доступная память считается в некотором смысле бесконечной и `malloc` всегда возвращает ненулевое значение. Однако при попытке реально работать с выделенным массивом программа может быть аварийно завершена системой по причине нехватки памяти.

Группа операторов в строках 17—20 отвечает за считывание данных в массив. Отметим, что величина `mas + n` по определению равна адресу `n`-й ячейки массива указанного типа, вычисляется по формуле `mas + n * sizeof(*mas)` и эквивалентна `&mas[n]`. Следовательно, обращение `*(mas + n)` в строке 23 означает содержимое `n`-й ячейки, т. е. эквивалентно обращению `mas[n]`. В строке 21 в переменную `ncur` записывается число реально считанных значений, что в дальнейшем потребуется для корректной работы с массивом. Здесь и далее мы считаем, что количество вводимых элементов массива может быть как меньше, так и больше переменной `nmax`.

Замечание. Функция `malloc` перед каждым выделяемым блоком памяти автоматически записывает информацию о его размере. Это позволит затем корректно отработать функции `free` (строка 30), получающей только адрес начала соответствующего блока. Случайное либо намеренное изменение данной информации приведет к «падению» программы при вызове функции `free`, т. е., казалось бы, после «чисто» отработавшей задачи. В случае корректного вызова функция `free` помечает захваченную память как свободную, что позволит операционной системе в дальнейшем использовать ее под другие задачи. Вызов функции `free` является обязательным для успешной работы больших проектов.

Полезно провести эксперименты по динамическому выделению больших объемов памяти, сравнимых с размером оперативной памяти. До тех пор пока не исчерпаем доступные ресурсы, будем вызывать функцию `malloc(1024 * 1024)` в цикле, обнуляя захваченные элементы. Проверьте, как изменится суммарное время работы при одновременном запуске десятка подобных задач, содержащих, для обеспечения временной задержки, вызовы функции `sleep(unsigned int seconds)` (заголовочный файл `unistd.h`).

Решения задач, сформулированных ниже, следует оформить в виде отдельных функций, получающих в качестве параметров адрес начала массива (т. е. адрес его нулевого элемента), длину массива, а также другие значения, если это необходимо по условию, и выполняющих без использования дополнительных массивов необходимые действия. В функциях не разрешено ни считывать новые данные, ни печатать сообщения на экран. Все нештатные ситуации нужно обрабатывать через возвращаемые значения. Функция `main` должна заполнять массив числами из файла (либо с клавиатуры), определять его фактическую длину, вызывать конкретную функцию и распечатывать результат ее работы.

Если в задаче массив преобразовывается с удалением элементов, то ответ (преобразованный массив) должен быть размещен в начале исходного массива. При этом нужно вычислить количество элементов в новом полученном массиве. Если не сказано специально, то оставшиеся в конце и не входящие в ответ элементы исходного массива, можно «не чистить», т. е. их значения могут оставаться любыми.

В задачах, где в условии упоминается несколько массивов, предполагается, что их длины позволяют разместить только исходные данные, а дополнительные массивы создавать нельзя.

На данном этапе вычислительная сложность алгоритмов не имеет принципиального значения, поэтому методы с квадратичной оценкой трудоемкости считаются вполне допустимыми. Вопросы о существовании эффективных алгоритмов остаются в качестве самостоятельных упражнений для любознательных.

Отметим также, что часть предлагаемых задач решается на основе однопроходных алгоритмов и формально не требует предварительного сохранения данных в массив.

Задача 1-3-1. Симметричен ли массив?

Решение. Приведем полное решение для случая, когда массив читается из файла и его длина определяется первым прочитанным числом.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int Symmetry(int *mas, int n);
int *MakeVectorInt(unsigned int n,
            const char *inf1, const char *inf2);
```

```
int main(void) {
  int *mas = NULL;
  int res, n, nmax = 0;
  char fname[256];
  FILE *fin = NULL;
  printf("----Симметричность массива----\n");
  printf("Введите имя файла ->");
  scanf("%s", fname);
  fin = fopen(fname, "r");
  if (!fin) {
    printf("Не удается открыть файл %s\n", fname);
    return -1;
  }
  printf("Считываем длину последовательности\n");
  if (fscanf(fin, "%d", &nmax) != 1) {
    printf("Ошибка считывания длины \n");
    return -2;
  }
  if (nmax <= 0) {
    printf("Ошибочное значение ");
    printf("длины последовательности\n");
    return -3;
  }
  mas = MakeVectorInt((unsigned int)nmax,
                                  __FILE__, __func__);
  printf("Считываем последовательность\n");
  for (n = 0; n < nmax; n++) {
    if (fscanf(fin, "%d", &mas[n]) != 1) break;
  }
  res = Symmetry(mas, n);
  printf("%s\n", (res) ? "Да" : "Нет");
  free(mas); fclose(fin);
  return 0;
}
int Symmetry(int *mas, int n) {
  int i;
  for (i = 0; i < n / 2; i++) {
    if (mas[i] != mas[n - i - 1]) return 0;
  }
  return 1;
```

```
}
int *MakeVectorInt(unsigned int n,
            const char *inf1, const char *inf2) {
  int *tmp = (int *)malloc(n * sizeof(int));
  if (tmp == NULL) {
    printf("Ошибка в %s -> %s :\n", inf1, inf2);
    printf("не удалось выделить %lu байт!\n",
           n * sizeof(int));
    exit(1);
  }
  memset(tmp, 0, n * sizeof(int));
  return tmp;
}
```

В данном примере функция `MakeVectorInt` получает, помимо длины массива, указатели на предопределенные компилятором строковые идентификаторы `__FILE__` и `__func__`. Это позволяет отследить точку сбоя при выполнении программы. Для подобных целей также часто применяется целочисленный идентификатор `__LINE__`.

Задача 1-3-2. Переставьте элементы массива в обратном порядке.

Решение.

```
void invert(int *mas, int n) {
  int i, tmp;
  for (i = 0; i < n / 2; i++) {
    tmp = mas[i];
    mas[i] = mas[n - i - 1];
    mas[n - i - 1] = tmp;
  }
  return;
}
```

Задача 1-3-3. Циклически сдвиньте элементы массива на одну позицию вправо.

Задача 1-3-4. Циклически сдвиньте элементы массива на k позиций вправо с затратой O(n) действий (n — длина массива).

Решение. Ограничение O(n) на число действий не позволяет вызвать k раз функцию из задачи 1-3-3. Приведем решение с использованием функции из задачи 1-3-2.

```
void move(int *mas, int n, int k) {
  k %= n;
  invert(mas, n - k);
  invert(mas + n - k, k);
  invert(mas, n);
  return;
}
```

Другой способ решения состоит в постановке элемента сразу на его новую позицию. При этом образуются цепочки элементов, которые требуется циклически сдвинуть. Количество таких цепочек равно наибольшему общему делителю чисел **n** и **k**.

Задача 1-3-5. Определите значение и индекс минимального элемента массива.

Решение. Поскольку здесь ответ составляют два числа, то будем получать индекс через возвращаемое значение функции, а значение минимума — через параметр-указатель. Приведем текст функции, обрабатывающей массив.

```
int Minimum(double *min, double *mas, int n) {
  int i, m;
  *min = mas[0];
  m = 0;
  for (i = 1; i < n; i++)
    if (*min > mas[i]) {
      *min = mas[i];
      m = i;
    }
  return m;
}
```

Задача 1-3-6. Определите, какое число встречается в массиве наибольшее количество раз.

Задача 1-3-7. Введите с клавиатуры число x и определите индекс и значение элемента массива, ближайшего к числу x.

Задача 1-3-8. Введите с клавиатуры число x и определите, к какому значению ближе всего x: к минимальному в массиве, к максимальному или к среднему арифметическому.

Задача 1-3-9. Введите с клавиатуры число x и определите количество элементов массива, расстояние от которых до x в два раза меньше, чем максимальное расстояние между x и элементами массива.

Задача 1-3-10. Каждый элемент массива (кроме первого и последнего) замените на полусумму соседних элементов.

Задача 1-3-11. Замените каждый элемент массива a[i] на сумму исходных элементов от нулевого до i-го включительно.

Задача 1-3-12. Сгруппируйте отрицательные элементы массива в его начале, а положительные в конце с сохранением их порядка.

Указание. Пока удается найти очередную пару индексов i_0, j_0, такую что $i_0 < j_0$, $a[i_0]$ — самый левый положительный элемент, $a[j_0]$ — первый следующий за ним отрицательный элемент, запоминаем $tmp = a[j_0]$, сдвигаем часть массива $a[i_0], \ldots, a[j_0 - 1]$ на одну позицию вправо и вставляем $a[i_0] = tmp$. Также можно реализовать алгоритм сортировки вставками 1-4-7, либо алгоритм пузырьковой сортировки 1-4-9, учитывая при сравнении только знаки элементов массива.

Задача 1-3-13. Требуется сравнить два неупорядоченных целочисленных массива A и B как числовые множества без учета повторяющихся элементов, т. е. проверить, верно ли, что $A \subset B$, $B \subset A$, $A = B$.

Задача 1-3-14. По двум неупорядоченным целочисленным массивам построить третий, являющийся объединением как числовых множеств, т. е. с удалением повторяющихся элементов. Найти его длину.

Задача 1-3-15. По двум неупорядоченным целочисленным массивам построить третий, являющийся пересечением исходных массивов как числовых множеств. Найти его длину.

Задача 1-3-16. Удалите из массива все отрицательные элементы, а оставшиеся уплотните (т. е. сдвиньте к началу массива с сохранением порядка).

Задача 1-3-17. Введите с клавиатуры число x и удалите из массива все элементы, превосходящие x, а оставшиеся уплотните.

Задача 1-3-18. Удалите из массива все одинаковые подряд идущие элементы, оставив только один из них, а оставшиеся уплотните.

Задача 1-3-19. Удалите из массива все повторные вхождения элементов. Оставшиеся в массиве элементы уплотните.

Задача 1-3-20. Введите с клавиатуры число x и удалите из массива каждый элемент, делящийся нацело на x. Оставшиеся элементы уплотните к началу массива.

В следующих задачах считается, что исходный массив имеет достаточную для требуемого добавления элементов длину.

Задача 1-3-21. Введите с клавиатуры число x и продублируйте (т. е. вставьте рядом такой же элемент) каждый элемент массива, превосходящий x.

Задача 1-3-22. Вставьте между каждыми двумя положительными элементами исходного массива их среднее арифметическое.

Задача 1-3-23. Замените каждый отрицательный элемент исходного массива на три элемента, равных его модулю.

Задача 1-3-24. Назовем x-отрезком группу подряд идущих элементов массива, каждый из которых равен x. Для заданного числа x замените элементы каждого x-отрезка на полусумму элементов, прилегающих к этому отрезку справа и слева. Если x-отрезок расположен в начале или конце массива, то будем считать соответствующий крайний элемент равным нулю.

Задача 1-3-25. Назовем массив из N целых чисел счастливым, если существует такое $0 < k < N$, что сумма элементов с индексами от 0 до $k - 1$ совпадает с суммой элементов с индексами от k до $N - 1$. Определите, является ли данный массив счастливым.

Задача 1-3-26. Назовем массив из целых чисел плотным, если множество значений элементов массива полностью заполняет некоторый отрезок $[a, b]$ (рассматриваются только целые значения). Определите, является ли данный массив плотным.

Задача 1-3-27. Получите массив биномиальных коэффициентов для степени N, последовательно вычисляя строки треугольника Паскаля. Для реализации можно использовать только один массив длины $N + 1$ (см. задачу 1-10-1).

Задача 1-3-28. Сгруппируйте элементы с четными индексами в начале массива с сохранением их исходного порядка относительно друг друга, а элементы с нечетными индексами — в конце массива, также с сохранением их исходного порядка.

Задача 1-3-29. В массиве длины N переместите элементы с индексами $i \leqslant [(N + 1)/2]$ на позиции с четными индексами с сохранением их исходного порядка относительно друг друга, а оставшиеся элементы ($i > [(N + 1)/2]$) разместите на позициях с нечетными индексами, также с сохранением их исходного порядка. В результате начальная и конечная половины массива «перемешиваются» чередованием элементов.

Задача 1-3-30. Пусть в массиве последовательно записаны цифры некоторого длинного десятичного числа. Реализуйте для такого «числа» функции прибавления единицы и вычитания единицы. Для реализации переноса из «старшего разряда» можно заранее запасти один лишний элемент в массиве.

Задача 1-3-31. Реализуйте функции сложения и вычитания двух длинных десятичных чисел, хранящихся в массивах в виде последовательности цифр.

В языках C и C++ имеется встроенный механизм работы с многомерными массивами. По определению массивом называется одномерная совокупность его элементов. Следовательно, двумерный (в обычном понимании) массив — это одномерный массив C/C++, элементами которого являются одномерные массивы (его строки). Поэтому двумерный массив в языках C и C++ в оперативной памяти хранится именно по строкам. Аналогично определяются трехмерный, четырехмерный и т. д. массивы. При этом явных ограничений на максимальную допустимую размерность не накладывается, однако конкретная реализация компилятора такое ограничение обычно имеет.

Задача 1-3-32. В прямоугольной целочисленной таблице (матрице) из 100 строк и 10 столбцов, найти строку с максимальной суммой. Распечатать ее номер и значение суммы.

Указание.

```
int MaxSumStr(
   int matr[][10], /* адрес начала массива,
         состоящего из строк длины 10 */
   int nRow, int *sumMax);
int SumStr(int *str, int n);
int main(void) {
  int matr[100][10];
  int nRow = 100, nCol = 10;
  int nMax, sumMax;
  for (int nr = 0; nr < nRow; nr++) {
    for (int nc = 0; nc < nCol; nc++) {
      matr[nr][nc] = rand() % 100;
    }
  }
  nMax = MaxSumStr(matr,nRow,&sumMax);
  printf("Строка с наибольшей суммой:\n");
```

```
  printf("номер=%d сумма=%d\n",nMax, sumMax);
  return 0;
}
int SumStr(int *str, int n) {
  int s = 0;
  for (int i = 0; i < n; i++) s += str[i];
  return s;
}
int MaxSumStr(int matr[][10],
                    int nRow, int *sumMax) {
  int s, nCol = 10, sMax, nMax;
  nMax = 0;
  sMax = SumStr(&matr[0][0], nCol);
  for (int nr = 1; nr < nRow; nr++) {
    s = SumStr(&matr[nr][0], nCol);
    if (s > sMax) {
      sMax = s;
      nMax = nr;
    }
  }
  *sumMax = sMax;
  return nMax;
}
```

Замечание. Имя многомерного массива (как и в одномерном случае) соответствует адресу его начала и указывается при передаче в функцию в качестве параметра. Так как двумерный массив хранится по строкам, следовательно, на этапе компиляции для формирования кода предписания `&matr[nr][0]` необходимо знать длину строки — поэтому в прототипе и теле функции `MaxSumStr()` соответствующее значение требуется указать явно. Данное ограничение значительно снижает гибкость получаемого кода. Отметим, что в функцию `SumStr()` передается адрес начальной ячейки очередной строки, поэтому дальнейшая работа с этой строкой не отличается от работы с обычным одномерным массивом. О правилах использования генератора псевдослучайных чисел `rand()` см. задачу 1-2-68.

Задача 1-3-33. Для целочисленной матрицы размера 10×20 найдите сумму максимальных элементов из каждой строки. Распечатайте значение суммы.

Задача 1-3-34. Дана матрица размера 10×20. Среди строк, имеющих более двух положительных элементов, найдите строку с максимальной суммой. Распечатайте ее номер и значение суммы.

Задача 1-3-35. Дана матрица размера 10×20. Поменяйте местами (если потребуется) ее первую строку и строку, сумма модулей элементов которой максимальна. Распечатайте исходную и полученную матрицы.

Задача 1-3-36. Дана матрица размера 10×20. Поменяйте местами (если потребуется) первый столбец и столбец, сумма квадратов элементов которого максимальна. Распечатайте исходную и полученную матрицы.

Задача 1-3-37. Дана квадратная матрица размера 20×20. Найдите строку, сумма квадратов элементов которой максимальна, и поменяйте местами эту строку и столбец, имеющий такой же номер. Распечатайте исходную и полученную матрицы.

Задача 1-3-38. Дана квадратная матрица размера 20×20. Отразите ее содержимое относительно следующих прямых:

1) (главной) диагонали, проходящей из левого верхнего угла матрицы в правый нижний;
2) диагонали, проходящей из правого верхнего угла матрицы в левый нижний угол;
3) вертикальной средней линии матрицы;
4) горизонтальной средней линии матрицы.

Указание. Третье преобразование можно выполнить, используя функцию из задачи 1-3-2.

Задача 1-3-39. Дана квадратная матрица размера 17×17. Поверните ее содержимое на 90 градусов по часовой стрелке.

Указание. Из курса геометрии известно, что поворот плоскости относительно некоторой точки O на угол α эквивалентен композиции отражений относительно двух произвольных прямых, проходящих через точку O и пересекающихся под углом $\alpha/2$.

Задача 1-3-40. В двумерной матрице размера 10×20 найдите подматрицу с максимальной суммой элементов. Распечатайте исходную и найденную матрицы.

Указание. См. задачу 1-2-60.

Напомним, что создаваемый статический массив размещается в программном стеке, поэтому его максимальный допустимый размер существенно меньше размера доступной оперативной

памяти. Для повышения гибкости программного кода работа с динамическими двумерными (многомерными) массивами обычно реализуется вручную на основе одномерного массива.

```
int *CrMatr(unsigned int nRow, unsigned int nCol);
void PrnMatr(int *matr, unsigned int nRow,
                                unsigned int nCol);
int main(void) {
  int *matr;
  unsigned int nRow = 10; // число строк
  unsigned int nCol = 20; // длина строки
  matr = CrMatr(nRow, nCol);
  if (matr) PrnMatr(matr, nRow, nCol);
  if (matr) free(matr);
  return 0;
}
int *CrMatr(unsigned int nRow,
                               unsigned int nCol) {
  const int VAL = 100; // нормирующий коэффициент
  int *mas;
  mas = (int *)malloc(nRow * nCol * sizeof(int));
  if (mas == NULL) return NULL;
  // инициализация начального значения для rand()
  srand((uint32_t)time(NULL));
  for (unsigned int nr = 0; nr < nRow; nr++) {
    // перебор строк:
    for (unsigned int nc = 0; nc < nCol; nc++) {
      // перебор в строке:
        // генерация псевдослучайной
        // последовательности
      mas[nr * nCol + nc] = rand() % VAL;
    }
  }
  return mas;
}
void PrnMatr(int *mas, unsigned int nRow,
                              unsigned int nCol) {
  for (unsigned int nr = 0; nr < nRow; nr++) {
    // перебор строк:
    for (unsigned int nc = 0; nc < nCol; nc++) {
```

```
      // перебор в строке:
      fprintf(stdout, "%5d ",
                              mas[nr * nCol + nc]);
    }
    fprintf(stdout, "\n");
  }
  return;
}
```

Отметим, что в некоторых случаях при работе с матрицами также уместна ссылочная реализация (см. раздел 2.6).

Правдивая история (диалог на зачете).
— Так что же делает ваша программа?
— Работает!!
— А точнее?
— Считает!
— Что считает?
— Числа.
— Какие числа?
— Типа double, кажется...

1.4. Поиск и сортировка

Процедуры поиска и сортировки традиционно относятся к классическим задачам программирования. Всюду в этом разделе для определенности мы будем рассматривать сортировку по возрастанию, т. е. упорядочивание элементов массива a так, чтобы для любого допустимого i было выполнено $a[i] \leqslant a[i+1]$. При реализации программ поиска и сортировки можно следовать двум подходам:

1) разрабатывать частные программы, ориентированные на конкретный тип данных массивов (например, целочисленный);
2) разрабатывать универсальные программы, предназначенные для сортировки любых массивов.

В первом случае для сравнения элементов между собой используются стандартные операции <, <=, >, >=, ==. При втором подходе программа сортировки приобретает дополнительный параметр — указатель на функцию сравнения, которую можно определять отдельно для каждого требуемого типа данных.

Задача 1-4-1. Пусть элементы массива не убывают. Требуется вставить в этот массив новый элемент x с сохранением упорядоченности всего массива, считая, что реально выделенная для его хранения память больше, чем используется на данный момент. Местоположение нового элемента следует определить последовательным поиском.

Решение. Пусть для определенности массив имеет тип **double**. Построенная функция будет возвращать новую длину массива.

```
int insert(double *mas, int n, double x) {
  int i = 0, j = 0;
  for (i = 0; i < n;) {// ищем место для x
    if (x > mas[i])
      i++;
    else
      break;
  }
  for (j = n; j > i; j--) {// раздвигаем массив
    mas[j] = mas[j - 1];
  }
  mas[i] = x;    // вставляем элемент
  return n + 1;
}
```

Задача 1-4-2. Пусть элементы целочисленного массива не убывают. Требуется бинарным поиском (методом деления пополам) определить, принадлежит ли массиву заданный элемент x.

Указание. Построим функцию, возвращающую 1, если элемент x присутствует в массиве, и 0 в противном случае.

```
int search(int *mas, int n, int x) {
  int left, right, mid;
  if (x < mas[0] || x > mas[n - 1]) return 0;
  left = 0;
  right = n - 1;
  if (x == mas[0]) return 1;
  while (right - left > 1) {
    mid = (left + right) / 2;
    if (x == mas[mid]) return 1;
    if (x > mas[mid]) {
      left = mid;
```

```
    } else {
      right = mid;
    }
  }
  return (x == mas[right]) ? 1 : 0;
}
```

Задача 1-4-3. Измените решение предыдущей задачи так, чтобы функция `search` дополнительно определяла индекс найденного элемента в массиве либо позицию, в которую можно поставить искомый элемент, если он не присутствует в массиве.

Задача 1-4-4. Пусть элементы массива не убывают. Требуется вставить в этот массив новый элемент x с сохранением упорядоченности всего массива. Местоположение нового элемента определить бинарным поиском.

Задача 1-4-5. Даны два неубывающих массива. Реализуйте функцию, строящую третий неубывающий массив, который является их объединением (т. е. содержит все элементы двух исходных массивов). Оформите решение в виде функции с прототипом

```
int merge(double *c, const double *a,
                     const double *b, int na, int nb);
```

получающей необходимые адреса и длины na, nb исходных массивов и возвращающей длину полученного массива `c`, т. е. $na + nb$. Оцените сложность алгоритма (количество необходимых действий).

Указание. На первом шаге из двух значений a[0], b[0] выбираем наименьшее и помещаем в c[0]. Если оказалось, что $b[0] < a[0]$, то на втором шаге процедура повторяется для a[0] и b[1], иначе — для a[1] и b[0]. При этом найденный на втором шаге минимальный элемент укладывается в c[1]. Если на очередном шаге получили, что в одном из массивов a, b все элементы закончились, то оставшийся «хвост» второго массива просто копируется в c. Сложность алгоритма $O(na + nb)$.

В следующих задачах необходимо реализовать функцию сортировки массива вещественных чисел по возрастанию с заголовком `void sort(double *a, int n)`.

Полезно сравнить время работы рассматриваемых далее алгоритмов на массивах большого размера (в том числе при наличии частичной упорядоченности данных). Для грубого ана-

лиза в ОС Linux допустимо применять утилиту `time` в формате `time ./a.exe`. Для аккуратного подсчета можно, подключив заголовочный файл `time.h`, вызвать функцию `clock()`, возвращающую переменную типа `clock_t`, содержащую количество тактов, совершенных процессором от начала выполнения программы, а затем поделить результат на макрос `CLOCKS_PER_SEC` для пересчета в секунды. Также можно воспользоваться функцией `time(NULL)`, которая возвращает *целое* число типа `time_t`, равное количеству секунд, прошедших с 00:00:00 UTC 1 января 1970 года — так называемой даты начала эры Unix (Unix Epoch).

```
#include <time.h>
.....
clock_t cbegin, cend;
time_t tbegin, tend;
double tT, cT;
cbegin = clock();
tbegin = time(NULL);
sort(a, n);
tend = time(NULL);
cend = clock();
tT = difftime(tend, tbegin);
cT = (double)(cend - cbegin) / CLOCKS_PER_SEC;
printf("time = %.0lf clock = %lf\n", tT, cT);
```

Следует помнить, что при работе функции менее секунды найденное время `T` будет равно нулю.

Задача 1-4-6. Метод перестановки максимального элемента (selection sort). *Идея алгоритма*: за первый просмотр найдем максимальный элемент из $\{a[0], \ldots, a[n-1]\}$ и поменяем местами его и элемент $a[n-1]$; далее процедура повторяется для подмассивов $\{a[0], \ldots, a[n-2]\}, \ldots, \{a[0], a[1]\}$.

Формальное описание. Пусть дано некоторое число k, $0 < k < n$. Находим максимальный элемент среди чисел $a_0, \ldots, a_k$. Пусть этим максимумом является некоторый элемент a_j. Обмениваем значения элементов a_j и a_k. Указанную процедуру последовательно выполняем для $k = n-1, n-2, \ldots, 1$. Сложность алгоритма $O(n^2)$ действий.

Задача 1-4-7. Сортировка вставками с последовательным поиском (insertion sort). *Идея алгоритма*: массив из одного элемента $a[0]$ упорядочен; добавим к нему элемент $a[1]$,

сохранив общую упорядоченность; затем добавим к результату a[2] и т. д.; положение добавляемого элемента определяется последовательным перебором.

Формальное описание. Предположим, что первые k элементов массива (т. е. $a_0, \ldots, a_{k-1}$) уже упорядочены нужным образом. Тогда, выполнив упорядоченную вставку элемента a_k в этот массив (задача 1-4-1), мы получим упорядоченную совокупность из $k+1$ элемента. Последовательно выполняем эту процедуру для $k = 1, 2, \ldots, n-1$. Сложность алгоритма $O(n^2)$ действий.

Решение. Приведем одну из прозрачных (и достаточно эффективных) реализаций алгоритма.

```
for (i = 1; i < n; i++) {
  j = i;
  while (j > 0) {
    if(a[j] < a[j - 1]) {
      swap(&a[j], &a[j - 1]);
      j--;
    }
    else{
      break;
    }
  }
}
```

Задача 1-4-8. Сортировка вставками с бинарным поиском. Алгоритм решения аналогичен предыдущему, но место для вставки нового элемента в упорядоченную часть массива отыскивается с помощью бинарного поиска (задача 1-4-2). Если алгоритм применить к обратно упорядоченному массиву, то на каждом шаге потребуется сдвигать на одну позицию всю отсортированную часть, что в итоге повлечет $O(n^2)$ действий.

Задача 1-4-9. «Пузырьковая» сортировка (bubble sort, sinking sort). *Идея алгоритма*: за первый проход обеспечим «подъем» наибольшего элемента из $\{a[0], \ldots, a[n-1]\}$ до позиции $a[n-1]$; далее процедуру будем повторять для подмассивов $\{a[0], \ldots, a[n-2]\}, \ldots, \{a[0], a[1]\}$.

Формальное описание. Последовательно сравниваем два соседних элемента a_i и a_{i+1}; если оказывается, что $a_i > a_{i+1}$, то меняем их местами. Такое сравнение и (возможно) перестановку выполняем для $i = 0, 1, \ldots, n-2$. Данную процедуру назовем

подъемом до $(n-1)$-й позиции. Далее последовательно выполняем подъемы до позиций $n-2, n-3, \ldots, 1$.

Решение.

```
for (k = 0; k < n - 1; k++) {
  for (i = 0; i < n - 1 - k; i++) {
    if (a[i] > a[i + 1]) {
      tmp = a[i + 1];
      a[i + 1] = a[i];
      a[i] = tmp;
    }
  }
}
```

Поясните, почему верхняя граница второго цикла после каждого прохода уменьшается на единицу. Если при очередном подъеме не будет выполнено ни одной перестановки, то массив уже является упорядоченным и процесс сортировки можно прекратить. Внесите соответствующее дополнение в код программы. Итоговая сложность алгоритма $O(n^2)$ действий.

Отметим, что если в упорядоченном массиве наибольший элемент переставить в начало, то сортировка потребует $n-1$ сравнение и столько же перестановок. Если же в упорядоченном массиве наименьший элемент переставить в конец, то для сортировки потребуется выполнить $O(n^2)$ сравнений.

Задача 1-4-10. Сортировка просеиванием (шейкерная сортировка, shaker sort). *Идея алгоритма*: в пузырьковой сортировке «легкий» (самый большой) элемент «всплывает» за один проход, но «тяжелые» (малые) элементы за один проход опускаются только на одну позицию; добавим процедуры быстрого спуска «тяжелых» элементов.

Формальное описание. Метод является модификацией пузырьковой сортировки и состоит из двух этапов: подъема и спуска. При подъеме последовательно сравниваются соседние элементы a_i и a_{i+1} $(i = 0, 1, \ldots)$ до тех пор, пока не будет сделана первая перестановка. Пусть эта перестановка затронула элементы a_k и a_{k+1}. Следующим этапом является спуск. Новый элемент a_k сравнивается с a_{k-1}, и если $a_k < a_{k-1}$, то выполняется перестановка. Сравнение продолжается в нисходящем направлении (т. е. для a_{k-1} и a_{k-2}, a_{k-2} и a_{k-3} и т. д.) до тех пор, пока *выполняются* перестановки и не достигнуто начало

массива. После этого возобновляется подъем с позиции $i = k + 1$. Таким образом, сортировка состоит из сменяющих друг друга процессов подъема (до первой перестановки) и спуска (до первого отсутствия перестановки), которые происходят до тех пор, пока при подъеме не будет затронут последний элемент массива a_{n-1} (при этом спуск также должен быть выполнен). Сложность алгоритма $O(n^2)$ действий. Поясните, чем данный алгоритм отличается от модифицированного варианта пузырьковой сортировки: последовательно чередуем подъем наибольшего и спуск наименьшего элемента из неотсортированной части массива.

Задача 1-4-11. Быстрая сортировка Хоара (quicksort). *Идея реализации*: в шейкерной сортировке обмены происходят только между соседними ячейками; для ускорения модифицируем алгоритм так, что станут возможны и дальние перестановки; при этом после первого прохода массив станет частично упорядоченным: будет состоять из двух неупорядоченных частей, таких что элементы первой части строго меньше некоторого *опорного* числа, а элементы второй части больше либо равны опорному. Опорное число подбирается (по возможности) так, чтобы длины полученных массивов были близки. Далее процедуру повторим для каждой из неупорядоченных частей, пытаясь в каждом случае оптимальным образом выбрать опорный элемент.

Формальное описание. Пусть мы имеем неупорядоченный массив $a_0, \ldots, a_{n-1}$. Найдем некоторое число m, удовлетворяющее условию $\min\limits_i a_i \leqslant m \leqslant \max\limits_i a_i$. Перестроим массив так, чтобы при некотором s, $0 \leqslant s \leqslant n-1$, было выполнено $a_i < m$ при $i < s$ и $a_i \geqslant m$ при $i \geqslant s$. Для этого, последовательно сравнивая элементы массива a_j, $j = 0, 1, \ldots$, с m, найдем первое значение j, такое что $a_j \geqslant m$. Теперь, последовательно сравнивая элементы массива a_k, $k = n-1, n-2, \ldots$ (т. е. от конца к началу), с m, мы либо найдем такое значение k, что $a_k < m$, $k > j$, либо получим $k = j$. Если $j < k$, то меняем местами элементы a_j и a_k и повторяем описанную выше процедуру для элементов массива $a_{j+1}, \ldots, a_{k-1}$. Если $k = j$, то мы получили искомое разбиение массива на две требуемые части при $s = k$. Теперь рекурсивно применим описанную выше процедуру для каждой из полученных частей массива (естественно, уже с новыми значениями m). После завершения всех рекурсивных вызовов массив будет упорядочен.

Быстродействие алгоритма сильно зависит от удачного выбора числа m, поскольку именно это число определяет, где массив будет разделен на две части. Если это разделение каждый раз происходит примерно посередине, то трудоемкость сортировки составляет $O(n \log_2 n)$ операций (n — количество элементов массива), однако если разделение каждый раз «отщепляет» только один элемент, то трудоемкость составит $O(n^2)$ операций. Неплохие результаты получаются, когда в качестве m берется среднее арифметическое нескольких (двух-трех) элементов рассматриваемой части массива. Заметим также, что при рекурсии первой следует обрабатывать часть массива, имеющую меньшую длину, так как это гарантирует, что глубина стека, хранящего рекурсивные вызовы функции, не превысит $\log_2 n$. При этом не стоит рекурсивно обрабатывать массивы из одного элемента и пустые массивы.

Отметим, что самостоятельная реализация действительно эффективного программного кода на основе данного алгоритма весьма затруднительна. Существенно проще запрограммировать сортировку слиянием (см. задачу 1-4-14).

Задача 1-4-12. Сортировка массива целых чисел подсчетом (counting sort). *Идея алгоритма*: если элементы исходного массива a[n] принадлежат известному a priori диапазону от 0 до V, то можно создать вспомогательный нулевой массив b[V + 1]; за один просмотр массива a для каждого значения v из диапазона от 0 до V посчитать количество элементов массива a, принимающих это значение, и сохранить результаты в b[v]; за один просмотр массива b последовательно перезаполнить в упорядоченном виде массив a. Сложность алгоритма O(n + V). Метод допускает обобщение на случай сортировки по целочисленному ключу составных объектов (структур): элементы с равными ключами сохраняются, например, в отдельных списках.

Задача 1-4-13. Пирамидальная сортировка (сортировка двоичной кучей, heapsort). *Идея алгоритма*: будем считать, что в массиве хранится двоичное дерево (см. раздел 3.3): a_0 — родительская (корневая) вершина, a_1 и a_2 — ее потомки, для родителя a_1 потомками являются a_3 и a_4, для a_2 — a_5 и a_6 и. т. д., т. е. при всех $i = 0, \ldots, n - 1$ для родителя a_i потомками являются a_{2i+1} и a_{2i+2}. Данное дерево называется пирамидой (сортирующим деревом, двоичной кучей), если каждый родитель не меньше, чем его потомки. В вершине такой пирамиды находится наибольший элемент массива. Исходный

неотсортированный массив можно превратить в пирамиду, если, двигаясь от вершины, последовательно сравнивать родителя с потомками и при необходимости менять их местами. Если для некоторой тройки «родитель—потомки» произошел обмен, то в соответствующей вершине спуск вниз следует приостановить, а процедуру проверок провести заново, поднявшись к ее родителю (и так, возможно, далее до корня дерева). Затем продолжается спуск вниз от вершины остановки. В построенной таким образом пирамиде первый элемент a_0 меняется с последним a_{n-1}. В результате наибольший элемент попадает на свое место, но условие упорядоченности нарушается. Поэтому процедура перестройки повторяется для массива на единицу меньшей длины. И т. д.

Формальное описание. Пусть мы имеем массив a_i, $i = 0, \ldots, n-1$. Будем говорить, что i-й треугольник пирамиды выстроен, если элемент a_i массива удовлетворяет условию $a_i \geqslant a_{2i+1}$ и $a_i \geqslant a_{2i+2}$ (если одно или оба значения $2i+1$ и $2i+2$ выходят за границы массива, то соответствующее неравенство считается выполненным). Процедура выстраивания i-го треугольника состоит в проверке неравенств $a_i \geqslant a_{2i+1}$ и $a_i \geqslant a_{2i+2}$ и выполнении обмена элемента a_i с максимальным из a_{2i+1}, a_{2i+2} в том случае, когда хотя бы одно из указанных неравенств нарушено.

Шаг 1. Сборка пирамиды. Выстроим 0-й треугольник. Далее для каждого $k = 3, \ldots, n-1$ будем выстраивать s-й треугольник, где s последовательно вычисляется по рекуррентной формуле $s = [k-1]/2$ и далее $s = [s-1]/2$ до $s = 0$ ($[\,]$ — целая часть). Заметим, что если при некотором s выстраивание треугольника не приводит к перестановке, то обработку текущего значения k можно не продолжать. В результате все i-е, $i = 0, \ldots, n-1$, треугольники пирамиды будут выстроены. Заметим, что при этом элемент a_0 окажется максимальным в массиве.

Шаг 2. Разборка пирамиды. Пусть $k = n$. Поменяем местами элементы a_0 и a_{k-1}. Таким образом, максимальный элемент занял свое место в будущем упорядоченном массиве. Будем теперь считать, что длина массива равна $k-1$, и выстроим 0-й треугольник. Если при этом произошла перестановка, то выстроим тот треугольник, вершина которого участвовала в обмене (т. е. каждый раз выстраиваем тот треугольник, куда переместился бывший элемент a_0). Так поступаем, пока при выстраивании треугольников происходят перестановки. Напомним, что элемент a_{n-1} уже занял свое место и в выстраиваниях не

участвует. Далее последовательно выполняем шаг 2 для $k = n-1, n-2, \ldots, 1$. В результате на позиции $n-2, n-3, \ldots, 0$ последовательно поступают требуемые элементы.

Нетрудно видеть, что алгоритм имеет гарантированную трудоемкость $O(n \log_2 n)$ и не требует дополнительной памяти.

Следующие алгоритмы, применяющие идею слияния учитывают, что два упорядоченных массива длины n можно объединить в один упорядоченный за n сравнений (см. задачу 1-4-5).

Задача 1-4-14. Сортировка простым слиянием (merge sort). На первом шаге весь массив рассматривается как совокупность упорядоченных групп по одному элементу в каждой. Слиянием (объединением) соседних групп во вспомогательный массив мы получаем упорядоченные группы, каждая из которых содержит два элемента (кроме, может быть, последней группы, которой не нашлось парной). Далее упорядоченные группы укрупняются в исходный массив.

Данный алгоритм может быть реализован с использованием рекурсивной процедуры. Разбиваем исходный массив на две половины. Если обе части оказываются упорядоченными, то после их слияния мы получим отсортированный исходный массив. Иначе рекурсивно применяем указанную процедуру к неупорядоченным частям. По завершении всех рекурсивных вызовов мы получим упорядоченный массив. В общем случае промежуточные проверки на упорядоченность можно опустить, считая, что упорядоченными будут только подпоследовательности, состоящие из одного элемента.

Сложность алгоритма $O(n \log_2 n)$ действий.

В дальнейшем предполагается, что исходный массив не помещается в оперативную память и мы вынуждены считывать информацию по частям из некоторого файла a.dat. При необходимости мы можем разбить исходные данные на несколько частей и записать их в файлы a1.dat, a2.dat,..., ak.dat либо, реально работая с одним файлом, программно поддерживать многоканальное чтение. При анализе алгоритмов такого рода особое внимание стоит уделять количеству обращений к файлу.

Задача 1-4-15. 2k-ленточное слияние. Пусть необходимо упорядочить массив длины n. Будем считать, что мы можем поддерживать k различных каналов на чтение и столько же на запись. На первом шаге считаем, что входные последовательности состоят из упорядоченных серий единичной длины. Считываем

по одной серии из каждого входного канала, сливаем их в одну упорядоченную серию длины k и записываем в первый выходной канал. Следующую упорядоченную серию длины k записываем во второй канал. Продолжаем действовать так, циклически меняя номера выходных каналов. Процедура повторяется до тех пор, пока не исчерпаются все входные последовательности. Далее входные и выходные каналы меняются местами и алгоритм повторяется. За каждый такой шаг мы получаем, что длина упорядоченных последовательностей увеличивается в k раз. Если общее число элементов равно $n = k^p$, то за $\log_k n$ шагов мы получим упорядоченный массив. Число пересылок при этом $O(n \log_k n)$. При произвольном n входные последовательности будут исчерпываться с различной скоростью и на каждом шаге мы будем реально сливать $k_i \leqslant k$ последовательностей различной длины.

Задача 1-4-16. Естественное слияние. В случае прямого слияния мы не получаем выигрыша, если исходный массив был частично упорядочен. Естественным обобщением является алгоритм, когда мы объединяем максимальные упорядоченные серии, а не последовательности фиксированной длины.

Задача 1-4-17. Многофазная сортировка. Откажемся от идеи, что у нас есть k входных и столько же выходных последовательностей одновременно. Будем считать, что в каждый момент имеется только один выходной канал, и работать так, что если какой-то из $2k - 1$ входных массивов опустел, то он становится выходным. Данный алгоритм полезен, когда входные последовательности имеют существенно разные длины. В лучшем случае число пересылок оказывается порядка $O(n \log_{2k-1} n)$.

Задача 1-4-18. Отсортировать массив целых чисел по младшей цифре посредством библиотечной функции qsort(), реализующей алгоритм quicksort. Прототип функции содержится в заголовочном файле `stdlib.h` и имеет вид

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*)(const void *, const void *));
```

Здесь первый аргумент `base` указывает на начало массива, второй и третий содержат длину (количество элементов) массива, и размер в байтах одного элемента массива, четвертый является указателем на функцию, позволяющую сравнивать элементы массива. Функция сравнения должна возвращать 0 для равных элементов, произвольное целое отрицательное число,

если первый элемент меньше второго, и произвольное целое положительное число, если первый элемент больше второго. Если два элемента массива равны (в смысле функции сравнения), то их порядок в отсортированном массиве не определен.

Решение. Реализуем алгоритм для модельного примера.

```
#include <stdlib.h>
int cmp(const void *a, const void *b);
int main(void) {
  int mas[3] = {123, -101, 1000};
  qsort((void *)mas, 3, sizeof(int), cmp);
  return 0;
}
int cmp(const void *a, const void *b) {
  int ia = *(const int *)a;
  int ib = *(const int *)b;
  return ia % 10 - ib % 10;
}
```

В результате получим 1000, −101, 123.

Задача 1-4-19. Упорядочить массив целых чисел с помощью алгоритма сортировки вставками с бинарным поиском (см. задачу 1-4-8), используя библиотечную функцию bsearch(), прототип которой содержится в заголовочном файле `stdlib.h` и имеет вид

```
void *bsearch(const void *search,
    const void *base, size_t nmemb, size_t size,
          int (*)(const void *, const void *));
```

Функция выполняет бинарный поиск элемента `*search` в упорядоченном массиве `base` из `nmemb` элементов, занимающих `size` байт каждый. Пятый параметр является указателем на функцию, позволяющую сравнивать элементы (см. задачу 1-4-18).

Программисту выдали задание: в пустой комнате спрятать два стальных шарика. На следующий день приходят, а он свой код отлаживает. Шариков нигде нет.
— Где шарики?!
— Шарики? Понятия не имею: первый сразу где-то потерялся, а потом и второй куда-то закатился...

1.5. Символьные переменные

При обработке на ЭВМ текстовой информации необходимо помнить, что в памяти компьютера хранятся не изображения символов, а соответствующие целочисленные коды. Как именно происходит восстановление внешнего вида символа на экране по его коду, зависит от типа кодирования. Наиболее простым является однобайтовое (восьмибитное) кодирование символов на основе таблицы ASCII (American Standart Code Information Interchange). В своем первоначальном варианте ASCII-7 таблица применялась при передаче телефонограмм: рабочими были младшие 7 бит, задающие 128 различных кодов, а старший бит использовался для контроля четности. При этом коды от 0 до 31 отвечали за управление принимающей аппаратурой, а остальные, соответствующие печатным символам, пробивались на бумажной ленте в виде семизнаковых групп: единицам соответствовали отверстия, нулям — пропуски. В современной версии для кодирования используются все 8 бит. Коды с 0-го по 127-й фиксированы и совпадают с кодами ASCII-7, а интерпретация расширенной части (кодов с 128-го по 255-й) зависит от типа выбранной кодовой страницы.

В первой части таблицы (коды с 0-го по 31-й) хранятся управляющие непечатные символы. Например, 7 — звонок, 8 — шаг назад (backspace), 10 — новая строка, 13 — возврат каретки, 14 — смена ленты в аппарате, 27 — клавиша Esc. К этой части также относится символ с кодом 127, клавиша Delete, — в двоичном представлении семь единиц, что пробивалось на ленте как семь отверстий.

Вторая часть таблицы, с 32-го символа (пробел) по 126-й символ (тильда, ~), содержит знаки препинания, цифры, заглавные и строчные буквы латинского алфавита. При этом цифры '0', ..., '9' имеют коды с 48 по 57, буквы 'A', ..., 'Z' — с 65 по 90, буквы 'a', ..., 'z' — с 97 по 122. Важно, что после '0' идет '1', '2', ..., а также выполнена естественная упорядоченность для заглавных 'A', 'B', ... и строчных 'a', 'b', ... латинских букв.

Третья (расширяемая) часть таблицы, с 128-го символа по 255-й, зависит от договоренности — типа выбранной кодовой страницы. К наиболее известным кириллическим таблицам относятся cp1251 (Windows-кодировка), koi-8, koi-8r, koi-8u (Linux-кодировки) и cp866 (DOS-кодировка). При неправильно указанной кодировке файл, содержащий кириллицу, будет

отображаться на экране «кракозябрами». Отметим, что на данный момент некоторые текстовые редакторы отказываются обрабатывать файлы, содержащие символы из расширенной части ASCII-таблицы. Для решения подобных проблем, актуальных для национальных алфавитов и специализированных научных областей, был предложен стандарт Unicode, присваивающий каждому активно используемому в мире символу уникальный целочисленный код. Его ранняя компьютерная реализация UTF-16 (Unicode Transformation Format, 16-bit) основана на двухбайтовом формате хранения кодов из Unicode-таблицы. На данный момент наибольшее распространение получила реализация UTF-8, использующая для хранения кода символа по необходимости от одного до четырех байт. Какое именно количество байт кодирует текущий символ, определяется по старшим битам. Например, байты вида (0*******) соответствуют символам с кодом от 0 до 127 из первой части таблицы ASCII.

Отметим, что в текстовых процессорах типа MS Word для кодирования символов используется своя внутренняя таблица шрифтов, отличная от ASCII. По этой причине не стоит использовать их для набора С-программ.

Для хранения символьных переменных в языке С используется тип **char**, занимающий один байт. Присвоить переменной **char** `at` конкретный символ можно двумя способами: либо явно задав его в одинарных кавычках, например `at = '@'`, либо указав его целочисленный код в десятичной `at = 64`, восьмеричной `at = 077`, шестнадцатеричной `at = 0x40` формах. Для считывания/печати переменных **char** используется формат `%c`. Например:

```
char c1 = '>', c2, c3, c4, c0;
c2 = 97;  // код символа  'a';
/* также допустимо: c2 = 0141; c2 = 0x61;
c2 = '\141'; c2 = '\x61'; */
c3 = c2 + 1;  // т. е. c3 = 'b';
c0 = '\n';    // код символа "новая строка"
scanf("%c", &c4);
printf("%c;%c;%c;c4 = ", c1, c2, c3);
printf("<%c>%c", c4, c0);
```

Отметим, что переменная `c0` содержит символ `'\n'` (newline, новая строка, код 10), поэтому последующая печать начнется

с новой строки. Все Linux-редакторы, отображая на экран содержимое документа и встречая символ с кодом 10, также отрабатывают переход на новую строку. Однако в системе Windows новая строка исторически (так же, как и в DOS) кодируется парой символов `'\r'` (возврат каретки, код 13) и `'\n'` (код 10). Поэтому нужно быть готовым к тому, что созданные под Linux-системой файлы некоторыми Windows-редакторами отобразятся в виде одной длинной строки, а редакторы под Linux при работе с файлами, созданными под OS Windows, дорисуют в конце каждой строки символ `'\r'` в виде знака `'^M'`. Подобные неудобства обычно решаются правильной настройкой параметров редактора либо утилитами типа `dos2unix`.

Задача 1-5-1. Считайте с клавиатуры символ и распечатайте его десятичный, восьмеричный и шестнадцатеричный коды, заключив текст в двойные кавычки.

Решение.

```
char sym;
char newline = '\n';//код символа "новая строка"
char dquo = 34; // код символа "двойные кавычки"
scanf("%c", &sym);
printf("%cchar = <%c>:", dquo, sym);
printf(" dec code = <%d>;", (int)sym);
printf(" oct code = <%o>;", (unsigned int)sym);
printf(" hex code = <%x>%c%c",
              (unsigned int)sym, dquo, newline);
```

Переменная типа **char** при переводе в целое число может принимать значения либо от −128 до 127, либо от 0 до 255. Поэтому при работе с символами, имеющими коды от 128 до 255, лучше явно указать тип **unsigned char**. Если же однобайтовая переменная для экономии места используется как целочисленная, то лучше явно описать ее как **signed char**.

Задача 1-5-2. Считайте с клавиатуры символ, и если это заглавная латинская буква, то замените ее на соответствующую строчную букву.

Решение.

```
char sym;
scanf("%c", &sym);
printf("char = <%c>, code = <%d>\n",
```

```
                                        sym, (int)sym);
if ((sym >= 'A') && (sym <= 'Z')) {
  sym = (char)(sym - 'A' + 'a');
}
printf("new char = <%c>, new code = <%d>\n",
                                        sym, (int)sym);
```

Замечание. Вызов функции `scanf("%c", &sym)` обеспечивает посимвольный ввод не всегда корректно (с точки зрения пользователя), так как введенная буква и следом нажатая клавиша **Enter** формально обрабатываются как два последовательно введенных символа — буква и `'\n'`. Это хорошо видно при запуске следующего кода (цикл прерывается при вводе комбинации клавиш Ctrl + d):

```
while(scanf("%c", &sym) == 1){
  printf("code = <%d> sym = <%c>\n",
                                        (int)sym, sym);
}
```

Отфильтровать символ «новая строка» обычно удается повторным считыванием:

```
if (sym == '\n') scanf("%c", &sym);
```

Однако более надежным является ввод полной строки (см. раздел 1.6) с последующим посимвольным разбором.

Задача 1-5-3. Определите, как кодируются управляющие клавиши типа Esc и стрелок.

Указание. Например, можно воспользоваться рассмотренной в замечании конструкцией.

Задача 1-5-4. Считайте из файла первый символ и определите, сколько раз он встречается далее в текстовом файле.

Задача 1-5-5. Найдите, сколько раз каждый символ таблицы ASCII встречается в текстовом файле.

Решение.

```
void count(FILE *fin, int *nmb) {
  unsigned char c;
  int i;
  for (i = 0; i < 256; i++) nmb[i] = 0;
  while (fscanf(fin, "%c", &c) == 1) {
    nmb[(unsigned int)c]++;
  }
```

```
  return;
}
void prn(int *nmb) {
  int i;
  for (i = 0; i < 256; i++) {
    printf("code = <%5d> ", i);
    printf("char = <%c> ", (unsigned char)i);
    printf("val = <%10d>\n", nmb[i]);
  }
  return;
}
```

Задача 1-5-6. Для заданного текстового файла найдите длину наибольшей цепочки,

1) состоящей из одного повторяющегося символа;
2) все соседние элементы которой различны;
3) состоящей из повторяющихся групп ABCD;
4) состоящей из повторяющихся групп ABCD, при условии, что последняя группа может быть неполной.

Задача 1-5-7. Определите, сколько цепочек вида AB?C встречается в файле (вместо знака ? может стоять произвольный символ).

Задача 1-5-8. Определите, сколько цепочек вида ABC*D встречается в файле. Здесь вместо знака * может стоять произвольная (в том числе и пустая) цепочка символов.

Задача 1-5-9. Для каждого символа из заданного массива определите, сколько раз он встречается в файле. Сложность алгоритма не должна превышать O(N) действий для файла длины N.

Решение.

```
int count(FILE *fin, unsigned char *str,
                          int *nmb, int n) {
  unsigned char c;
  int i = 0;
  int buf[256];
  for (i = 0; i < 256; i++) buf[i] = 0;
  while (fscanf(fin, "%c", &c) == 1) {
    buf[(unsigned int)c]++;
  }
  for (i = 0; i < n; i++) {
    nmb[i] = buf[(unsigned char)str[i]];
```

```
  }
  return i;
}
```

Замечание. При работе с текстовым файлом удобно использовать функцию `int fgetc(FILE *)` (ее прототип находится в заголовочном файле `stdio.h`), возвращающую переменную типа `int`. В случае успеха это будет неотрицательный целочисленный код считанного символа (т. е. число от 0 до 255), иначе — код ошибки. Так, в случае ошибки считывания либо при достижении конца файла функция возвращает именованную целочисленную константу `EOF` (End Of File). Проверить, что достигнут именно конец файла можно последующим вызовом `feof(FILE*)`.

Задача 1-5-10. Напишите функцию, получающую в качестве параметра имя текстового файла и подсчитывающую количество заглавных латинских букв в файле.

Идеи реализации.

```
int c;
int num = 0;
while ((c = fgetc(fin)) != EOF) {
  if (('A' < c) && (c < 'Z')) num++;
}
if (feof(fin) == 0) {
  printf("Error\n");
} else {
  printf("End of file\n");
}
```

Задача 1-5-11. Напишите функцию, получающую в качестве параметра имя текстового файла и копирующую его содержимое в другой файл с заменой всех маленьких латинских букв на большие.

Идеи реализации.

```
int c;
while ((c = fgetc(fin)) != EOF) {
  if ('a' < c && c < 'z') c = c + 'A' - 'a';
  if (fputc(c, fout) == EOF) break;
}
```

Задача 1-5-12. Напишите функцию-фильтр, копирующую из одного файла в другой все содержимое, за исключением

символов, содержащихся в заданном текстовом массиве. Функция должна иметь заголовок

```
void copyfilter(FILE *fin, FILE *fout,
                unsigned char *badsym, int n);
```

где `fin`, `fout` — указатели на входной и выходной файлы, `badsym` — массив символов, которые не надо пропускать на выход, `n` — его длина. Сложность алгоритма не должна превышать O(n) действий для файла длины n (см. задачу 1-5-9).

Задача 1-5-13. Напишите функцию-фильтр, которая считывает из файла разделенные пробелами последовательности, состоящие из цифр и букв, а копирует в другой файл целые числа (по одному в строке), которые получаются после вычеркивания всех лишних (т. е. отличных от цифр) символов. Усложните задачу, считая, что встреченные до первой цифры символы + и − определяют итоговый знак числа, т. е. плюсы игнорируются, а два минуса дают плюс.

Задача 1-5-14. Напишите функцию-фильтр, которая при каждом обращении возвращает очередной допустимый символ из заданного тестового файла. При достижении конца файла функция возвращает значение `EOF`. Набор допустимых символов задается массивом — параметром функции. Функция должна иметь заголовок

```
int filter(FILE *fin, unsigned char *goodsym,
                                    int n);
```

где `fin` — указатель на входной файл, `goodsym` — массив символов, которые нужно пропускать на выход, `n` — его длина.

Идеи реализации. Просмотр строки `goodsym` для каждого вновь прочитанного символа приведет к значительным затратам времени на обработку большого файла. Поэтому можно завести в функции массив `static int good[256]`, в который при обращении с ненулевым указателем `goodsym` записать единицы для допустимых и нули для остальных символов. Этот вызов можно рассматривать как инициализацию. Далее, обращаясь к функции с нулевым значением параметра `goodsym`, нужно проверять элементы массива `good` и выдавать ответ. Эта часть функции фактически сводится к выполнению операторов

```
int sym;
while ((sym = fgetc(fin)) != EOF){
```

```
  if (good[sym]) break;
}
return sym;
```

В первом классе.
— Кто знает порядок букв в латинском алфавите?
— Q, W, E, R, T, Y, U, I, O, P...
— А вот и нет! Эргономичнее будет Q, W, F, P, G, J, L, U, Y...

1.6. Текстовые строки

Строкой в языке С называется последовательность элементов типа **char**, заканчивающаяся `'\0'`, т. е. символом с нулевым кодом. Такой символ не имеет изображения, его нет на клавиатуре, а поэтому он не должен встречаться в текстовом файле. Это дает возможность использовать `'\0'` в качестве управляющего символа, означающего конец символьной строки. Например:

```
/* создаем массив s1 из 10 символов: */
char s1[10];
/* положим в массив s1[] строку из двух символов: */
s1[0] = 'a';
s1[1] = 'b';
s1[2] = '\0';
/* положим в массив пустую строку: */
s1[0] = '\0';
```

Следующие присвоения допустимы только на этапе описания символьных массивов:

```
/* создаем массив s2[] из 10 элементов,
содержащий строку из трех элементов;
символ s1[3]='\0' добавляется автоматически: */
char s2[10] = "abc";
/* создаем массив s3[] из 2 элементов,
содержащий строку из одного элемента;
символ s3[1]='\0' добавляется автоматически: */
char s3[] = "a";
/* создаем указатель s4 на константную строку */
char *s4 = "a.txt";
```

Для считывания с клавиатуры слова (т. е. последовательности символов, не содержащей пробелов и знаков табуляции) может быть использована конструкция следующего типа:

```
#define Nmax 100;
char str[Nmax];
scanf("%s", str); // Либо: scanf("%s",&str[0]);
```

В данном случае в конец считанной последовательности автоматически добавляется `'\0'`. Если до нажатия клавиши **Enter** на клавиатуре было набрано несколько слов, то считается только первое, а остальные останутся в буфере ввода до очередного к нему обращения. Для ввода полной строки, набранной на клавиатуре, удобно использовать функцию `gets(str)`, которая считывает и сохраняет все элементы, включая символ перехода на новую строку, и автоматически добавляет за ними символ `'\0'`. Отметим, что при работе с функциями `scanf("%s", str)`, `gets(str)` необходимо заранее выделить место, достаточное для сохранения в памяти считанной информации, иначе будут нарушены правила работы с памятью. Для защиты от ввода недопустимо больших строк обычно используют безопасную функцию `char *fgets(char *, int, FILE *)`. Здесь второй параметр равен максимальному разрешенному количеству символов для чтения, включая завершающий нулевой символ. Далее нужную информацию выделяют из считанной строки либо посимвольным разбором, либо посредством удобной функции `sscanf(const char*, const char*,...)`.

Отметим, что группы функций

```
scanf(), printf() и
gets(), puts(), fgets(,,stdin), fputs(,,stdout)
```

не всегда совместимы, поэтому в рамках одного потока стоит пользоваться либо только первым, либо только вторым набором. Их смешение может привести к трудно контролируемым ошибкам.

Задача 1-6-1. Изучите работу функций `scanf()` и `printf()` по формату `"%s"`.

Решение.

```
#define N 64
char str[N];
while (scanf("%s", str) == 1) {
```

```
  for (i = 0; i < N; i++) {
    printf("i = %d char = <%c> code = %3d\n",
                          i, str[i], (int)str[i]);
  }
  printf("<%s>\n", str);
}
```

Проанализируйте результат для входных данных `"123"`, `" 123 "`, `" 123 45 6"`, `" "`, а также некорректного ввода типа строки из тысячи единиц.

Задача 1-6-2. Реализуйте функцию, вычисляющую длину строки, т. е. количество символов до `'\0'`.

Решение.

```
int my_strlen(cons char *s) {
  int i = 0;
  while (s[i] != '\0') i++;
  return i;
}
```

Задача 1-6-3. Реализуйте функцию, копирующую одну строку в другую.

Решение. Одно из возможных решений выглядит так:

```
char *my_strcpy(char *dst, const char *src) {
  char *s = dst;
  while (*(dst++) = *(src++))
    ;
  return s;
}
```

В данном случае компактность кода существенно выше его «прозрачности», поэтому предложите свой «наглядный» вариант.

Задача 1-6-4. Реализуйте аналоги функций, выполняющих стандартные операции с текстовыми строками: `strcat()`, `strncpy()`, `strcmp()`, `strstr()`, `strspn()\lstinline` и `strcspn()`.

Задача 1-6-5. Напишите функцию, находящую количество слов в файле.

Указание. Если длина максимального слова известна и не превосходит, например, 127, то можно использовать конструкцию следующего вида:

```
num = 0;
char str[128];
while (fscanf(f, "%s", str) == 1) num++;
```

В общем случае для корректного выделения слов произвольной длины можно использовать `fgets()` с дальнейшим посимвольным анализом.

Задача 1-6-6. Напишите функцию, находящую количество непустых строк в файле.

Указание. Для строк, содержащих не более 256 символов (включая считываемый символ «новая строка»), можно использовать конструкцию следующего вида:

```
num = 0;
char str[257];//для хранения строки с учетом '\0'
while (fgets(str, 257, f) != NULL)
  if (str[0] != '\n' && str[0] != '\r') num++;
```

Модифицируйте алгоритм на случай строк заранее не известной длины.

Замечание. При решении подобных задач для файлов большого размера рекомендуется использовать низкоуровневую функцию `fread()`, что позволит существенно ускорить считывание данных.

Задача 1-6-7. С клавиатуры вводится строка, содержащая некоторое количество целых чисел, разделенных пробелами или табуляциями. Общее количество символов не превосходит 256. Распечатайте все введенные целые числа.

Указание. Для решения подобных задач удобно использовать функцию `sscanf`:

```
char str[257];
int val, sLen = 0, len = 0;
fgets(str, 256, stdin);
while(sscanf(str + sLen, "%d%n",
                        &val, &len) == 1){
  printf("<%d>\n", val);
  sLen = sLen + len;
}
```

В данном случае с клавиатуры считывается строка, а затем из нее последовательно считываются целые числа. Спецификатор формата `%n` позволяет записать в переменную `len` количество

байтов, обработанных функцией `sscanf` за текущее обращение. Это дает возможность вычислить адрес `str + sLen` начального байта оставшейся части. Отметим, что спецификатор `%n` относится к небезопасным, а поэтому возможность его использования может быть заблокирована.

Задача 1-6-8. Назовем словом группу символов, не содержащую внутри себя символов из заданного набора символов-разделителей. Примерами разделителей для слов обычного литературного текста могут служить символы `.`, `,`, `;`, `:`, `!`, `?`, пробел, `(`, `)`, `[`, `]`, `-`, `+`, `*`, `/`, `"` и т. п. Реализуйте функцию, которая при каждом обращении к ней выделяет из указанного текстового файла очередное слово. Функция должна иметь прототип

```
int GetWord(FILE *f, char *word, char *delim);
```

где `f` — указатель на входной файл, `word` — указатель на буфер для очередного слова, `delim` — указатель на строку с символами-разделителями. Возвращаемое значение 0, если слово прочитано, и -1 в случае конца файла или какого-либо другого отказа.

Задача 1-6-9. В файле, содержащем литературный текст, возможны переносы слов между строками. Модифицируйте функцию `GetWord` из задачи 1-6-8 так, чтобы она обнаруживала и правильно выводила перенесенные слова. Можно считать, что слово разбито переносом на две части, если за допустимыми (буквенными) символами идет символ «-», за которым следуют один или несколько символов перевода строки, и далее, возможно, несколько пробелов до следующего допустимого символа.

Задача 1-6-10. Используя функцию `GetWord` из задачи 1-6-8, выполните следующую обработку текстового файла:

1) найдите количество слов в исходном файле;
2) найдите максимальную, минимальную и среднюю длину слов;
3) найдите среднее количество слов в одном предложении (предложение — это последовательность слов, оканчивающаяся одним из символов «.», «!», «?»);
4) выберите и напечатайте все слова, начинающиеся с заглавной буквы.

Задача 1-6-11. Напишите функцию сортировки массива слов длины менее 256, учитывающую при сравнении только первую букву.

Решение. Воспользуемся функцией `qsort` из стандартной библиотеки (заголовочный файл `stdlib.h`).

```
#include <stdio.h>
#include <stdlib.h>
#define PRINT
int cmp(const void *a, const void *b);
int main(void) {
  unsigned const int n = 4; // для работы
  char *mas[4];             // с четырьмя словами
  unsigned const int len = 256; // такой длины
  unsigned int i;
  for (i = 0; i < n; i++) {
    mas[i] = (char *)malloc(len+1);//с учетом '\0'
    scanf("%s", mas[i]);
  }
  printf(">сходный массив: ");
  for (i = 0; i < n; i++)
    printf("%s ", mas[i]);
  printf("\n");
  qsort((void *)mas, n, sizeof(char *), cmp);
  printf(">тоговый массив: ");
  for (i = 0; i < n; i++)
    printf("%s ", mas[i]);
  printf("\n");
  return 0;
}
int cmp(const void *a, const void *b) {
  const char *ca = *(const char *const *)a;
  const char *cb = *(const char *const *)b;
#ifdef PRINT
  printf("\n%s: <%s, %s>\n", __func__, ca, cb);
#endif
  return ca[0] - cb[0];
}
```

В результате (см. задачу 1-4-18) получим упорядоченный с учетом только первой буквы массив.

Задача 1-6-12. Модифицируйте решение задачи 1-6-11 для сортировки массива слов в лексикографическом порядке.

Задача 1-6-13. Реализуйте собственную функцию сортировки массива слов в лексикографическом порядке. Для создания и хранения массива используйте следующую конструкцию.

```
int nWords = 1000;
int lMax = 30;
char **mas;
int n;
mas = (char **)malloc(nWords * sizeof(char *));
if (mas == NULL) exit(1);
for (n = 0; n < nWords; n++) {
  mas[n] = (char *)malloc(lMax * sizeof(char));
  if (mas[n] == NULL) exit(2);
}
n = 0;
while (n < nWords) {
  if (fscant(in, "%s", mas[n]) != 1) break;
  n++;
}
```

В данном примере в случае отказа при выделении памяти вызывается функция `exit`, что является некорректным действием. До полного завершения кода необходимо высвобождать всю захваченную к данному моменту память (см. задачу 2-6-1).

Задача 1-6-14. Модифицируйте решение задачи 1-6-13 так, чтобы для хранения каждого слова выделялось в точности требуемое количество байт.

Указание. Завести буфер `char buf[lMax]` достаточной длины, в буфер считывать очередное `n`-е слово, вычислять его длину, выделять требуемое место, сохраняя указатель в `mas[n]`, а затем копировать из буфера в массив. Предусмотреть защиту от ввода слов длины больше, чем `lMax`.

Задача 1-6-15. По заданному текстовому файлу сформируйте файл-словарь, содержащий все слова из исходного файла, записанные в алфавитном порядке по одному в строке.

Идеи реализации. Считая, что максимальное количество слов нам известно, организуем хранение слов на базе массива указателей на слова, а при добавлении очередного слова реализуем алгоритм сортировки вставками с бинарным поиском на основе функции сравнения `strcmp()`.

Задача 1-6-16. Оформите решение задачи 1-6-15 так, что имена исходного файла и создаваемого файла-словаря передаются в программу из командной строки в формате `./prg.e Text.txt Dict.txt`.

Указание. Приведем пример функции, печатающей передаваемые ей из командной строки аргументы.

```
int main(int *argc, char **argv) {
  for (int i = 0; i < (*argc); i++) {
    printf("i = %d, argv[%d]: <%s>\n",
                                  i, i, argv[i]);
  }
  return 0;
}
```

Задача 1-6-17. С клавиатуры последовательно вводятся строки следующего формата:

Фамилия Имя id dq

Известно, что количество строк не превосходит тысячи. Здесь Фамилия и Имя — последовательности, содержащие не более 20 и 15 символов соответственно; id — положительное целое число, являющееся уникальным идентификатором, dq — рейтинг. Требуется считать информацию, сохранить ее в памяти ЭВМ и выдать на экран упорядоченные по алфавиту имена и соответствующие id всех пользователей, имеющих значение dq не менее 1000.

Указание. Для хранения вводимой информации создать массив структур подходящего типа, заполнить его данными с клавиатуры, произвести контрольную печать, отсортировать по алфавиту по полю Имя, распечатать искомый результат.

```
/* Описание структуры и прототипов функций */
struct dataStr {
  char fname[16];
  char lname[21];
  int id;
  int dq;
};
int ReadData(struct dataStr *data);
void PrnData(struct dataStr *data, int n);
void SortData(struct dataStr *data, int n);
void PrnUns(struct dataStr *data, int n);
```

```
/* Основной блок */
int main(void){
  unsigned int nMax = 1000;
  int n;
  struct dataStr *data;
  data = (struct dataStr *)malloc(
              nMax * sizeof(struct dataStr));
  if (data == NULL) exit(1);
  n = ReadData(data);
  PrnData(data, n); // контрольная печать
  SortData(data, n); // сортировка по полю Имя
  PrnUns(data, n); // выборочная печать
  return 0;
}
/* Рабочие функции */
int ReadData(struct dataStr *data) {
  int n = 0;
  while (fscanf(stdin, "%s %s %d %d",
      data[n].lname, data[n].fname,
      &(data[n].id), &(data[n].dq)) == 4) n++;
  return n;
}
void PrnData(struct dataStr *data, int N) {
  int n = 0;
  for (n = 0; n < N; n++) {
      fprintf(stdout, "%s %s %d %d\n",
      data[n].lname, data[n].fname,
      data[n].id, data[n].dq);
  }
  return;
}
```

Задача 1-6-18. С клавиатуры последовательно вводятся строки следующего формата:

Фамилия Имя id dq

(см. задачу 1-6-17). Требуется считать информацию, сохранить ее в памяти ЭВМ и выдать на экран фамилии и id всех пользователей, имеющих значение dq не менее 1000, упорядочив их по алфавиту. Если встречаются пользователи с одинаковой фамилией, то их необходимо упорядочить по полю id.

Указание. Отсортировать введенный массив по совокупности полей Фамилия + id, реализовав соответствующую функцию сравнения, и распечатать пользователей с допустимым dq.

— Ну и пусть говорят, что использовать в качестве пароля имя своего кота — дурной тон! `RrGgTt_FfxX17!`, кыс-кыс-кыс, пойдем, дам молочка.

1.7. Разбор чисел и битовые операции

При решении задач данного раздела можно анализировать остатки от деления целых чисел, полученные с помощью оператора `%`, либо использовать битовые операции `<<`, `>>`, `~`, `^`, `|`, `&`, обеспечивающие соответственно сдвиг влево, сдвиг вправо, отрицание (not), «исключающее или» (xor, сложение по модулю 2), побитовое «или» (or, логическое сложение), побитовое «и» (and, логическое умножение).

Задача 1-7-1. Получите массив целых чисел `int b[32]`, хранящий двоичное разложение неотрицательного целого числа

$$n = b[0] + b[1] \cdot 2 + \ldots + b[31] \cdot 2^{31}, \quad n < 2^{32}.$$

Решение.

```
int longint2binary(unsigned long int n, int *b) {
  unsigned long int mask = 1;
  int i;
  for (i = 0; mask; i++) {
    if (n & mask) {
      b[i] = 1;
    } else {
      b[i] = 0;
    }
    mask = mask << 1;
  }
  return i;
}
```

В данном случае возвращаемое значение позволит проверить размер типа `long int`. Реализуйте аналогичную функцию побитового разбора переменной типа `uint64_t`.

Задача 1-7-2. Для вычисления суммы $s = 1 + 2 + \ldots + 2^n + \ldots$ реализовали следующий код:

```
int InfSum2n (void) {
  int s = 0, sn = 0, qn = 1;
  while (1) {
    sn = s + qn;
    if (s == sn) break;
    qn *= 2;
    s = sn;
  }
  return s;
}
```

Какое значение вернет функция?

Замечание. В данном случае результат «подкрепляется» аналитическими выкладками:

$$s = \sum_{n=0}^{\infty} q^n = \frac{1}{1-q}.$$

Задача 1-7-3. Получите массив целых чисел, соответствующий реальному битовому представлению заданного символа в памяти компьютера, т. е. `b[0]` содержит старший бит, а `b[n-1]` — младший бит символа.

Решение.

```
void char2bits(char c, int *b) {
  unsigned char mask = 1 << 7;
  int i;
  for (i = 0; mask; i++) {
    if (c & mask) {
      b[i] = 1;
    } else {
      b[i] = 0;
    }
    mask >>= 1;
  }
  return;
}
```

Задача 1-7-4. Получите массив целых чисел, соответствующий реальному битовому представлению целого числа типа `long int` в памяти компьютера.

Решение. Для хранения переменных типа `long int` используется несколько байт, а последовательность их размещения в памяти зависит от архитектуры ЭВМ, аппаратных настроек, выбранной сборки ОС и обычно бывает двух типов: от старшего к младшему (big-endian) или от младшего к старшему (little-endian). Поэтому в решении задачи 1-7-3 недостаточно формально заменить `char` на `long int`. Для получения искомого ответа необходимо отдельно проанализировать каждый байт.

```
void longint2bits(long int n, int *b) {
  char *c;
  int i, Nb = sizeof(long int);
  c = (char *)(&n);
  for (i = 0; i < Nb; i++) {
    char2bits(*c, b + i * 8);
    c++;
  }
  return;
}
```

Отметим, что при работе с переменными, длина которых превышает машинное слово, возможен так называемый смешанный (mixed-endian) порядок байт.

Задача 1-7-5. В заданном байте `var` установите с учетом нумерации слева направо k-й по счету бит, $0 \leqslant k \leqslant 7$, равным нулю, единице или инвертируйте его.

Указание. Положим `mask = 1 << (7 - k);`, а затем

```
var = ~mask & var; //выставить 0,
var = mask | var; //выставить 1,
var = var ^ mask; //инвертировать.
```

Задача 1-7-6. Реализуйте функцию с прототипом

```
int Copter (unsigned char key),
```

обеспечив для нее передачу семи целочисленных параметров при условии, что первые шесть могут принимать только значения 0 или 1, а седьмой — 0, 1, 2, 3.

Указание. Будем считать, что шесть младших бит переменной `key` отвечают за значения первых шести входных параметров, а два старших бита — за седьмой параметр.

Задача 1-7-7. В массиве целых чисел установите k-й по счету бит (с учетом сквозной нумерации слева направо) равным единице, нулю.

Задача 1-7-8. Для целого числа получите массив b с цифрами его записи в k-ичной системе счисления для $k \leqslant 2^8$ (либо $k \leqslant 2^{64}$). Например, для числа 254 при $k = 16$ нужно получить $b[0] = 14$, $b[1] = 15$, $b[i] = 0$ при $i > 1$.

Задача 1-7-9. На основе задачи 1-7-8 реализуйте функции сложения и вычитания «длинных» чисел, записанных в k-ичной системе счисления.

Задача 1-7-10. С клавиатуры вводятся две последовательности цифр, представляющие собой целую и дробную части некоторого положительного десятичного числа. Требуется найти его двоичное представление, сохранив результат в отдельных массивах (см. задачу 1-7-1).

Задача 1-7-11. Дано целое число. Получите целое число, записанное теми же цифрами, идущими в обратном порядке.

Задача 1-7-12. Вычислите представление числа m/n в виде десятичной дроби (т. е. выведите цифры, составляющие начало и период этой дроби).

Идеи реализации. Можно запасти массивы необходимых размеров и реализовать алгоритм деления столбиком, запоминая получающиеся остатки. Период появится тогда, когда мы получим два одинаковых остатка. Можно также проанализировать делители чисел m и n, что даст возможность заранее вычислить длины начальной части и периода дроби. В этом случае массивы не нужны, так как получаемые цифры частного можно сразу выводить на печать.

Задача 1-7-13. Реализуйте функцию возведения числа x в заданную целую степень N не более чем за $2\log_2 N$ умножений.

Идеи реализации. Найдем разложение

$$N = n_0 2^0 + n_1 2^1 + \ldots + n_k 2^k.$$

Тогда

$$x^N = x^{n_0}(x^2)^{n_1} \ldots (x^{2^k})^{n_k}.$$

Поэтому можно последовательно вычислять значения x^1, x^2, x^4 и т. д. и перемножать только те из них, которые соответствуют единицам в двоичном представлении числа N.

Задача 1-7-14. Для натуральных взаимно простых a, b, $a > b > 0$, найдите все целочисленные решения диофантова уравнения $ax + by = 1$.

Решение. Задача имеет бесконечно много решений. Если найдено одно из них (x_0, y_0), то любое решение можно представить как $(x_0 + \widetilde{x}, y_0 + \widetilde{y})$. Подставив эту пару в исходное уравнение, находим, что $a\widetilde{x} = -b\widetilde{y}$. Следовательно, всякое решение имеет вид

$$(x_0 + kb, y_0 - ka), \quad k = 0, \pm1, \pm2, \ldots.$$

Для поиска (x_0, y_0) применим алгоритм Евклида вычисления нод(a, b) (см. задачу 1-1-42): пока `a > b`, выполняем `a = a - b;`, иначе меняем `a`, `b` местами и действие повторяем. Адаптируя идею к исходной задаче, рассмотрим систему формальных тождеств:

$$1 \cdot a + 0 \cdot b = a;$$
$$0 \cdot a + 1 \cdot b = b.$$

Будем из первого равенства вычитать второе, пока `a > b`, затем поменяем тождества местами и повторим процедуру. Если на очередном шаге правая часть стала равной единице, то решение найдено.

Для формализации данного процесса перепишем исходные тождества в виде

$$x_1 \cdot a + y_1 \cdot b = d_1; \quad \text{где } x_1 = 1,\ y_1 = 0,\ d_1 = a; \qquad (1)$$
$$x_2 \cdot a + y_2 \cdot b = d_2; \quad \text{где } x_2 = 0,\ y_2 = 1,\ d_2 = b. \qquad (2)$$

Поделим d_1 на d_2 с остатком: $d_1 = d_2 \cdot p_2 + d_3$, затем умножим равенство (2) на p_2 и результат вычтем из (1). Получим

$$x_3 \cdot a + y_3 \cdot b = d_3, \qquad (3)$$

где

$$x_3 = x_1 - x_2p_2,\ y_3 = y_1 - y_2p_2,\ d_3 = d_1 - d_2p_2.$$

Поделим d_2 на d_3 с остатком: $d_2 = d_3 \cdot p_3 + d_4$, затем умножим (3) на p_3, вычтем из (2) и получим новое равенство, и т. д. Если на очередном шаге имеем $x_N \cdot a + y_N \cdot b = d_N = 1$, то искомое решение найдено: $x_0 = x_N$, $y_0 = y_N$. Приведем итоговый код:

```
x1 = 1; y1 = 0; d1 = a;
x2 = 0; y2 = 1; d2 = b;
while (d2 > 1) {
```

```
  p2 = d1 / d2; d3 = d1 % d2;
  x3 = x1 - x2 * p2; y3 = y1 - y2 * p2;
  x1 = x2; y1 = y2; d1 = d2;
  x2 = x3; y2 = y3; d2 = d3;
}
x0 = x2; y0 = y2;
```

Суть изложенного алгоритма Евклида заключается в построении следующей цепочки соотношений:

Дано:	$b_0x_0 + b_1y_0 = 1, \quad b_0 = a, \ b_1 = b; \ b_0 > b_1,$
делим с остатком:	$b_0 = b_1p_1 + b_2$, подставляем в уравнение,
вводим замену:	$x_1 = y_0 + p_1x_0, \ y_1 = x_0,$
получаем:	$b_1x_1 + b_2y_1 = 1, \quad b_1 > b_2,$
...	...
делим с остатком:	$b_k = b_{k+1}p_{k+1} + b_{k+2}$, подставляем,
вводим замену:	$x_{k+1} = y_k + p_{k+1}x_k, \ y_{k+1} = x_k,$
получаем:	$b_{k+1}x_{k+1} + b_{k+2}y_{k+1} = 1,$
...	...
получаем:	$b_Nx_N + b_{N+1}y_N = 1,$
делим с остатком:	$b_N = b_{N+1}p_{N+1} + 1$, подставляем,
получаем:	$b_{N+1}(x_Np_{N+1} + y_N) + x_N = 1.$

Отсюда находим целочисленное решение $x_N = 1$, $y_N = -p_{N+1}$. И далее по обратным рекуррентным формулам замены вычисляем частное решение x_0, y_0 исходной задачи $b_0x_0 + b_1y_0 = 1$.

Приведем рассмотренную реализацию алгоритма, считая, что массивы x, y, p, b имеют достаточную длину.

```
int DiophEqII(int *x, int *y, int *p, int *b) {
  int n, N, k;
  scanf("%d%d", b + 1, b + 2);
  n = 2;
  while (1) {
    p[n] = b[n - 1] / b[n];
    b[n + 1] = b[n - 1] % b[n];
    if (b[n + 1] == 1) break;
    n++;
  }
  N = n;
  for (k = 2; k <= N; k++) {
    printf("k = %d p = %d b = %d\n",
                k, p[k], b[k + 1]);
  }
```

```
  x[N - 1] = 1;
  y[N - 1] = -p[N];
  for (k = N - 1; k > 1; k--) {
    x[k - 1] = y[k];
    y[k - 1] = x[k] - p[k] * y[k];
  }
  printf("(%d) * (%d) + (%d) * (%d) = 1\n",
         b[1], x[1], b[2], y[1]);
  return 0;
}
```

Замечание. При желании указанную систему формальных тождеств можно хранить в виде массивов, т. е. как двумерную систему линейных уравнений:

$$\begin{cases} 1 \cdot a + 0 \cdot b = a; \\ 0 \cdot a + 1 \cdot b = b; \end{cases} \quad \Longleftrightarrow \quad mx = d.$$

Тогда допустима следующая программная реализация.

```
int DiophEqI(void) {
  int m[2][2];
  int a, b, p, d[2], x[2];
  int iMin, iMax;
  scanf("%d%d", &a, &b);
  d[0] = a; d[1] = b;
  if (a > b){
   iMax = 0; iMin = 1;
  }
  else{
   iMax = 1; iMin = 0;
  }
  m[0][0] = 1; m[0][1] = 0;
  m[1][0] = 0; m[1][1] = 1;

  while (1) {
    if (d[iMin] == 1) {
      x[0] = m[iMin][0];
      x[1] = m[iMin][1];
      break;
    }
    p = d[iMax] / d[iMin];
```

```
    m[iMax][0] -= m[iMin][0] * p;
    m[iMax][1] -= m[iMin][1] * p;
    d[iMax] -= d[iMin] * p;
    iMax = (iMax + 1) % 2;
    iMin = (iMin + 1) % 2;
  }
  printf("(%d) * (%d) + (%d) * (%d) = 1\n",
         a, x[0], b, x[1]);
  return 0;
}
```

Задача 1-7-15. Выведите в файл все подмножества множества $\{0, \ldots, N-1\}$.

Идеи реализации. Если ограничиться значениями $N < 32$, то можно воспользоваться следующим приемом. Будем считать, что для каждого целого числа m, $m < N$, набор единиц в двоичном представлении соответствует некоторому подмножеству множества $\{0, \ldots, N-1\}$. Перебрав все числа $m = 0, \ldots, 2^N - 1$ (например, с помощью цикла), мы тем самым переберем все возможные подмножества.

Задача 1-7-16. Выведите в файл все k-элементные подмножества множества $\{0, \ldots, N-1\}$.

Идеи реализации. Можно воспользоваться решением задачи 1-7-15 и отбирать только те значения m, в двоичном представлении которых ровно k единиц (см. также задачу 1-9-25).

Задача 1-7-17. Эффективно вычислите все простые числа, не превосходящие n.

Идеи реализации. Очевидный способ решения — последовательно проверять числа $1, 2, \ldots, n$ на простоту (см. задачу 1-1-33). Для больших n лучше воспользоваться алгоритмом «решето Эратосфена»: заводим битовый массив на n элементов и заполняем его 1, a priori считая все числа простыми. Затем для $k = 2, 3, \ldots, m$, $m^2 \leqslant n$ проверяем значение k-й ячейки. Если оно равно 1, то обнуляем все ячейки, с номерами, кратными k, из диапазона от k^2 до n. Асимптотика такого алгоритма порядка $O(n \log\log n)$ действий.

Задача 1-7-18. Эффективно вычислите все простые числа, лежащие между m и n, воспользовавшись алгоритмом «двойное решето Эратосфена».

Идеи реализации. Находим и сохраняем в отдельном массиве все простые числа, не превосходящие $\sqrt{n}$, а далее с их помощью вычеркиваем все «лишние» числа из требуемого промежутка. Отметим, что если в задаче 1-7-17 значение n велико, то последовательное применение данного подхода при правильно подобранных n и m позволяет ускорить работу алгоритма за счет автоматического кэширования при работе с массивами.

Задача 1-7-19. С учетом равенства $30 = 2 \cdot 3 \cdot 5$ вычислите все простые числа, лежащие между m и n.

Идеи реализации. Представим число 30 в виде $30 = 2 \cdot 3 \cdot 5$ и выпишем все простые числа до 30: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. Отсюда следует, что каждое простое число, большее 30, можно представить в одной из восьми форм: $30k + 1$, $30k + 7$, $30k + 11$, $30k + 13$, $30k + 17$, $30k + 19$, $30k + 23$, $30k + 29$. Это позволяет процесс вычеркивания (и поиска) проводить только в указанных восьми цепочках.

Для эффективного поиска простых чисел также можно использовать решето Аткина—Бернштейна.

Задача 1-7-20. В текстовом файле, хранящемся в формате UTF-8, подсчитайте количество закодированных символов.

Указание. Для хранения кода одного символа в реализации UTF-8 выделяется от одного до четырех байт. Какое именно количество байт кодирует текущий символ, можно определить, анализируя старшие биты. Если байт имеет вид (0*******), то он кодирует символ с кодом от 0 до 127 из первой части таблицы ASCII. Для кодирования символа в формате двух, трех и четырех байт используются соответственно последовательности вида

(110*****)(10******)

(1110****)(10******)(10******)

(11110***)(10******)(10******)(10******)

При этом битовое представление кода символа из таблицы Unicode последовательно вписывается справа налево на допустимые позиции, отмеченные символом *. Невостребованные левые биты заполняются нулями.

Для идентификации каждого сетевого интерфейса в пределах одного сегмента сети используется шестибайтовый MAC-адрес, обычно записываемый в виде шести пар шестнадцатеричных чисел. Первые n бит MAC-адреса кодируют номер предприятия,

а остальные — серийный номер устройства. Узнать MAC-адрес на Windows-ЭВМ можно командой `getmac`, набранной в командной строке. При этом на экран шестнадцатеричные пары будут выданы разделенными символом тире, например `A4-CE-D8-F4-AB-FB`, (в Linux для этого используется символ двоеточие).

Задача 1-7-21. Реализуйте функцию, получающую MAC-адрес в виде текстовой строки, целое значение n и возвращающую номер предприятия и серийный номер устройства в виде двух строк с искомыми цифрами.

В версии протокола IPv4 для идентификации компьютеров в разных подсетях используется четырехбайтовый IP-адрес, для удобства записываемый в виде четырех десятичных чисел, разделенных точкой, например 192.168.0.1. Таблицу соответствий между MAC-адресами и IP-адресами можно получить, набрав в командной строке команду `arp -a`. Первые n бит IP-адреса кодируют адрес сети, а остальные — адрес узла в сети.

Задача 1-7-22. Реализуйте функцию, получающую IP-адрес в виде текстовой строки, целое значение n и возвращающую два четырехбайтовых массива с адресом сети и адресом узла соответственно. Распечатайте найденные адрес сети и адрес узла в виде четырех десятичных чисел, разделенных точкой. Модифицируйте функцию так, чтобы она допускала ввод IP-адреса и величины n в форме одной строки, содержащей так называемую краткую слэш-нотацию: 192.168.161.1/24.

В терминологии сетей TCP/IPv4 маской сети называют четырехбайтовый адрес, в котором первые n бит — единицы, а остальные — нули. Побитно «перемножив» IP-адрес и маску, получим адрес сети; побитно «перемножив» IP-адрес и инвертированную маску, найдем адрес узла в сети.

Задача 1-7-23. Реализуйте функцию, получающую IP-адрес и маску сети в виде двух четырехбайтовых массивов и возвращающую четырехбайтовые массивы с адресом сети и адресом узла.

Отметим, что для адресации компьютеров в подсети не используются два адреса: адрес сети, т. е. адрес, в котором все биты, отсекаемые маской, равны 0, и так называемый широковещательный адрес, в котором все биты, отсекаемые маской, равны 1.

Задача 1-7-24. Включите в реализацию функции из задачи 1-7-23 проверку на корректность полученного адреса узла.

— Бился Иван-царевич с одноглавым Чудищем и срубил ему голову, а у того две выросло, срубил две — четыре выросло, срубил четыре — восемь выросло, а как восемь срубил — так Чудищу и конец пришел.
— А почему шестнадцать не выросло?
— Видимо, Чудище имело четырехбитную архитектуру.

1.8. Обработка множества точек

В следующих задачах предполагается, что в файле записано несколько пар чисел, которые можно рассматривать как координаты множества точек на плоскости или как координаты множества концов отрезков на прямой. Требуется составить программы, которые читают исходные данные и отвечают на поставленные вопросы или выдают значения параметров требуемых геометрических объектов. Следует стремиться к построению минимальных по сложности алгоритмов (сложность оценивается порядком числа операций в зависимости от числа точек или отрезков).

Задача 1-8-1. Множество точек определяет ломаную. Имеет ли она самопересечения?

Задача 1-8-2. Множество точек определяет многоугольник. Является ли он выпуклым?

Задача 1-8-3. Множество точек определяет многоугольник. Определите его диаметр (т. е. длину наибольшей диагонали). Решение должно иметь сложность O(N), где N — число вершин многоугольника.

Задача 1-8-4. Дано множество отрезков. Покрывает ли их объединение заданный отрезок [a, b]?

Задача 1-8-5. Дано множество отрезков. Найдите точку, которая принадлежит наибольшему количеству отрезков, определите это количество.

Задача 1-8-6. Дано множество точек. Найдите центр и радиус минимального круга, который содержит все эти точки.

Задача 1-8-7. Дано множество отрезков. Выберите из него и распечатайте связное подмножество отрезков, объединение которых дает отрезок наибольшей длины.

Задача 1-8-8. Дано множество точек. Из этих точек как из центров начинают одновременно и с одинаковой скоростью «расти» круги (т. е. точки являются центрами, а радиусы увеличиваются с одинаковой скоростью). Если два круга сталкиваются, то их рост прекращается. Определить радиусы всех получившихся кругов, после того как их рост прекратится.

Задача 1-8-9. Окружность разделена на m равных дуг, и внутри каждой дуги на окружности отмечено некоторое количество точек (возможно, нулевое). Требуется передвинуть эти точки на окружности так, чтобы были выполнены следующие условия:

1) каждая точка должна по возможности наименее удаляться от своей исходной позиции;
2) ни одна точка не должна покидать свою первоначальную дугу;
3) точки должны сохранять исходный порядок, а угловое расстояние между любыми двумя соседними точками не должно быть меньше заданной величины δ (для разрешимости данной задачи можно считать, что $\delta < 2\pi/mn$, где n — наибольшее количество точек, приходящихся на одну дугу).

Задача 1-8-10. Обобщением задачи 1-8-9 является задача о размещении названий на географической карте. Пусть заданы координаты нескольких точек на плоскости (населенные пункты) и с каждой точкой связан некоторый прямоугольник, задаваемый своей длиной и шириной (он ограничивает текст наименования населенного пункта). Требуется разместить каждое наименование рядом с соответствующей ему точкой так, чтобы наименования (прямоугольники) не накладывались друг на друга.

Задача 1-8-11. Множество точек определяет многоугольник (возможно, невыпуклый, но без самопересечений). Определите, находится заданная точка внутри или вне этого многоугольника.

Задача 1-8-12. Пусть плоские выпуклые многоугольники представляются следующим типом данных:

```
struct Polygon {
  int n;
  double *x, *y;
};
```

где n — число вершин многоугольника, x, y — указатели на массивы координат вершин. Реализуйте следующий набор функций, обеспечивающих работу с многоугольниками как с объектами.

```
/* периметр: */
double Perimeter(struct Polygon p);
/* площадь: */
double Area(struct Polygon p);
/* пересечение: */
struct Polygon* Clip(struct Polygon a,
                     struct Polygon b);
/* равны?: */
int Equal(struct Polygon a, struct Polygon b);
/* выпуклый?: */
int Convex(struct Polygon a);
```

Считаем, что пересечение непересекающихся многоугольников пусто. Функция `Clip` создает новый многоугольник, т. е. выделяет для него память и заполняет требуемыми значениями.

Задача 1-8-13. Пусть в трехмерном пространстве определена плоскость: задан вектор нормали и трехмерные координаты одной точки. Реализуйте функцию, которая вычисляет координаты ортогональной проекции точки пространства на эту плоскость.

Задача 1-8-14. Пусть в трехмерном пространстве дана плоскость: два базисных вектора и трехмерные координаты одной точки. Реализуйте функцию, которая вычисляет координаты ортогональной проекции точки пространства на эту плоскость.

Задача 1-8-15. Пусть в трехмерном пространстве дана плоскость с локальной системой координат: два базисных вектора и трехмерные координаты точки на плоскости, являющейся началом координат. Реализуйте функцию, которая вычисляет локальные координаты ортогональной проекции точки пространства на эту плоскость.

— Буратино, предположим, что вам дали два яблока, а некто взял у вас одно яблоко. Сколько яблок у вас осталось?

— Понятия не имею! Может, у меня своих был мешок?!

Мораль: всегда обнуляйте переменные!

1.9. Рекурсия

При построении алгоритмов для задач данного раздела полезно сравнивать два подхода: рекурсивный (отложенные вычисления) и итеративный (последовательные вычисления). В первом случае искомый результат получается в процессе обратного хода рекурсии (что требует сохранения всех промежуточных вычислений). Во втором случае ответ полностью найден на последнем шаге, и поэтому цепочка рекурсивных выходов носит формальный характер.

Задача 1-9-1. Как отработают вызовы `f(0);`, `f(5);`, `f(10);`.

```
void f(int n) {
  if (n == 5) return;
  printf(">%d", n);
  f(n + 1);
  printf("<%d", n);
  return;
}
```

Задача 1-9-2. Напишите рекурсивную реализацию вычисления суммы $S(n) = 1 + 2 + \ldots + n$ для $n \geqslant 1$ с учетом представления $S(n) = S(n-1) + n$.

Решение.

```
int S(int n) {
  if (n <= 0)
    exit(1);
  if (n == 1)
    return 1;
  return S(n - 1) + n;
}
```

Задача 1-9-3. Напишите рекурсивную реализацию вычисления суммы целых неотрицательных чисел $S(a, b) = a + b$, считая, что из арифметических действий допустимы только увеличение на 1 и уменьшение на 1.

Указание. Например, $S(a, b) = S(a + 1, b - 1)$.

Задача 1-9-4. С клавиатуры вводятся целые числа. Ввод завершается либо комбинацией клавиш Ctrl + d, либо набором произвольных нечисловых данных. Напишите рекурсивную реализацию вычисления суммы $S(n) = x_1 + \ldots + x_n$ для $n \geqslant 1$.

Решение. Поясните, в чем идейное отличие следующих решений:

```
int Sback(void){
 int cur;
 if(scanf("%d",
            &cur) == 1){
   return cur + Sback();
 }
 else{
   return 0;
 }
}
```

```
int Sforth(int sCur){
  int cur;
  if(scanf("%d",
           &cur) == 1){
    return Sforth(cur
                  + sCur);
  }
  else{
    return sCur;
  }
}
```

Задача 1-9-5. Напишите рекурсивную реализацию вычисления функции n!, сравните ее эффективность с эффективностью последовательного вычисления (см. задачу 1-1-17).

Задача 1-9-6. Сравните эффективность рекурсивной реализации нахождения n-го числа Фибоначчи $f_n = f_{n-1} + f_{n-2}$, $f_1 = f_2 = 1$, с эффективностью вычислений по рекуррентной и явной формулам (см. задачу 1-1-36).

Указание.

```
unsigned long int FBRecursion(int N) {
  unsigned long int xn;
  if (N == 1 || N == 2) return 1;
  xn = FBRecursion(N - 2) + FBRecursion(N - 1);
  return xn;
}
```

Задача 1-9-7. Для заданного целого числа напечатайте на экране его вид в k-ичной системе счисления.

Задача 1-9-8. Напишите рекурсивную функцию перевода целого числа $d = (a_n...a_0)_k$, записанного в k-ичной системе счисления при $k < 10$, в десятичную систему счисления.

Указание. Воспользуйтесь схемой Горнера:

$$d = a_0 + k(a_1 + \ldots + k(a_{n-1} + ka_n)\ldots).$$

Задача 1-9-9. Напишите рекурсивную и последовательную реализации печати символьной строки `char *s` в формате

```
printf("<|%c|...|%c|>", s[0], ..., s[n-1]);
```

Задача 1-9-10. Реализуйте функцию, печатающую на экране инструкции для игры «Ханойская башня».

Решение.

```
void Hanoi(int n,  // число колец
           int fr, // номер башни с кольцами
           int to  // номер конечной башни
           ){
    int tmp = 6 - fr - to;
    if (n == 1){
        printf("%d -> %d\n", fr, to);
        return;
    }
    Hanoi(n-1, fr, tmp);
    printf("%d -> %d\n", fr, to);
    Hanoi(n-1, tmp, to);
    return;
}
```

Также можно воспользоваться «мнемоническими правилами»:

для четного n циклически повторять $(1,2)$; $(1,3)$; $(2,3)$;

для нечетного n циклически повторять $(1,3)$; $(1,2)$; $(2,3)$.

Здесь запись (i,j) означает, что нужно выполнить перекладывание одного верхнего кольца между стержнями i и j.

Задача 1-9-11. Оцените размер сегмента оперативной памяти в вашей системе, предназначенного для хранения стека вызовов.

Решение. В стеке вызовов для каждой вызванной функции хранятся значения локальных переменных класса **auto** и адрес оператора в точке возврата (также возможны пустые байты для выравнивания адресов). Поэтому для функции `f` каждый рекурсивный вызов уменьшит размер стека на **sizeof**(size_t) + **sizeof**(void *) байт.

```
void f(size_t n) {
  printf("n = %zu\n", n);
  f(n + 1);
  return;
}
```

Задача 1-9-12. Распечатайте все последовательности длины 5 вида $a_0a_1a_2a_3a_4$, где $a_i \in \{1,3,7\}$. Найдите количество таких цепочек.

Указание.

```
int main(void){
  int a[5]; // текущая цепочка
  int val = 0; // итоговое количество 3^5
  Step(a, 0, &val);
  printf("Val = %d\n", val);
  return 0;
}
void Step(int *a, int k, int *pval) {
  if (k > 5) return;
  if (k == 5){
    (*pval)++; // построена новая цепочка
    Prn(a,k); // печать цепочки
    return;
  }
  a[k] = 1; Step(a, k+1, pval); // рекурсивный
  a[k] = 3; Step(a, k+1, pval); // спуск
  a[k] = 7; Step(a, k+1, pval); // по вариантам
  return;
}
```

Задача 1-9-13. Распечатайте все последовательности длины 5 вида $a_0a_1a_2a_3a_4$, где $a_i \in \{1, 2, \ldots, 9\}$ и при всех $k \geqslant 0$ сумма $a_0 + \ldots + a_k$ кратна a_{k+1}. Найдите количество таких цепочек.

Задача 1-9-14. Для заданных положительных целых чисел a, b, $a < b$, найдите и распечатайте все представления вида $b = a[][]\ldots$, где вместо каждой пары скобок [] подставлено одно из следующих выражений: $+2$, $+3$, $*5$. Считаем, что в полученной формуле все действия выполняются последовательно слева направо независимо от приоритета операций.

Задача 1-9-15. Для заданных положительных целых чисел a, b и N_0 найдите и распечатайте все представления $b = a[][]\ldots$ длины не более N_0, где вместо каждой пары скобок [] подставлено одно из следующих выражений: $[+2]$, $[-3]$, $[*5]$. В формуле действия выполняются последовательно слева направо независимо от приоритета операций.

Задача 1-9-16. Пусть заданы натуральное число a, множество из двух натуральных чисел $B = \{b_1, b_2\}$, множество из двух бинарных арифметических действий $Q = \{-, /\}$. Найдите и распечатайте все возможные представления единицы в виде

$1 = a * [] * [] * \ldots * []$, где вместо каждой $*$ подставлено произвольное действие из Q, а вместо $[]$ — произвольное число из B. В формуле действия выполняются последовательно слева направо независимо от приоритета операций.

Задача 1-9-17. Пусть заданы целое число a, множество $B = \{b_1, b_2, \ldots, b_n\}$ из n различных целых чисел, множество из четырех бинарных операций $Q = \{-, /, +, \cdot\}$ и целое число N_0. Найдите и распечатайте все возможные представления единицы в виде $1 = a * [] * [] * \ldots * []$ (см. задачу 1-9-16), содержащие не более N_0 арифметических действий. В формуле действия выполняются последовательно слева направо независимо от приоритета операций.

Задача 1-9-18. Золотым сечением Φ называется положительный корень уравнения $x^2 - x - 1 = 0$, т. е. число $\Phi = \dfrac{1+\sqrt{5}}{2}$. Его можно определить рекурсивно как $x = 1 + \dfrac{1}{x}$, т. е. с помощью цепной дроби $\Phi = 1 + \dfrac{1}{1 + \dfrac{1}{1+\ldots}}$. Найдите экспериментально количество слагаемых цепной дроби, обеспечивающих точность 10^{-10}.

Указание. Покажите, что рекуррентный процесс $x_{n+1} = 1 + 1/x_n$ сходится к Φ при $n \to \infty$ для произвольного $x_0 > 0$. Для этого можно использовать так называемую паутинную диаграмму Ламерея — изображение графиков функций $\varphi(x) = 1 + 1/x$ и $f(x) = x$ с отмеченной на них траекторией $x_0, \varphi(x_0), \varphi^2(x_0), \ldots$. При этом координаты траектории задаются точками $\{(x_i, y_i),\ i = 0, 1, \ldots\}$, где x_0 — выбранное стартовое приближение, $y_0 = \varphi(x_0)$, x_1: $f(x_1) = y_0$, $y_1 = \varphi(x_1)$, x_2: $f(x_2) = y_1, \ldots$.

Задача 1-9-19. Для приближенного вычисления числа e можно использовать следующие соотношения:

$$1)\ e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots, \quad 2)\ e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{2}{3 + \cfrac{3}{4 + \ldots}}}},$$

$$3)\ \frac{e-1}{2} = [0; 1, 6, 10, 18, \ldots], \quad 4)\ e = [2; 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots],$$

где $[a_0; a_1, a_2, a_3, \ldots] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ldots}}}$.

Напишите рекурсивные реализации формул (что, конечно, менее эффективно, чем рекуррентный пересчет в цикле). Сравните точность формул, взяв одинаковое число слагаемых.

Идеи реализации. Например, для второй формулы имеем $e = 2 + 1/\varphi(1)$, где $\varphi(n) = n + n/\varphi(n+1)$, $\varphi(N) = N$ при выбранном значении N.

Задача 1-9-20. Найдите приближенное значение числа e, используя соотношение $(e+1)/(e-1) = \varphi(2)$, где $\varphi(n) = n + 1/\varphi(n+4)$ и $\varphi(N) = N$ при достаточно большом N. Сравните точность данного метода и методов из задачи 1-9-19 при одинаковом числе слагаемых.

Задача 1-9-21. Напишите рекурсивную реализацию вычисления дисперсии $D = \frac{1}{n}\sum_{i=1}^{n}(x_i - M)^2$ элементов числовой последовательности. Сравните эффективность данного подхода с эффективностью однопроходного алгоритма задачи 1-2-5.

Указание. См. задачу 1-9-4.

Задача 1-9-22. Напишите рекурсивную реализацию алгоритма сортировки простым слиянием (см. задачу 1-4-14).

Задача 1-9-23. Пусть задан двумерный целочисленный массив A размерности M на N, заполненный нулями и двойками. Двойки соответствуют стенкам, а нули — пустотам. Требуется смоделировать процесс заливки пустот из заданной (i,j)-й ячейки, т. е. заполнения единицами тех нулевых элементов массива, до которых последовательно можно добраться.

Задача 1-9-24. Пусть задан двумерный целочисленный массив A размерности M на N, заполненный нулями и двойками. Двойки соответствуют стенкам, а нули — проходам. Требуется найти и распечатать все возможные пути из заданной (i,j)-й ячейки до ячейки с номером $(0,0)$, содержащие не более k шагов.

Задача 1-9-25. Требуется вывести в файл все k-элементные подмножества множества $\{0,\ldots,N-1\}$.

Идеи реализации. Создадим массив из N элементов-признаков, соответствующих элементам исходного подмножества и указывающих, использован или нет данный элемент в подмножестве. Формирование подмножества выполняется рекурсивной процедурой, которая последовательно помечает один свободный элемент массива как использованный в подмножестве и обращается сама к себе. Глубину рекурсивных вызовов нужно ограничить значением k. Также см. задачу 1-7-16.

Задача 1-9-26. Составьте программу, которая выводит значение введенного натурального числа словами. Например, для числа 427 выводится строка «четыреста двадцать семь».

Задача 1-9-27. Пусть имеется некоторое количество однозначных (состоящих из одной цифры) чисел. Построим арифметическое выражение, используя все эти числа и объединяя их операциями сложения, умножения, вычитания, деления, возведения в степень. Составьте программу, которая по заданному массиву однозначных чисел вычисляет значения всех таких выражений и печатает их вместе с видом соответствующего выражения. При этом порядок чисел в выражении всегда остается одним и тем же, а действия выполняются слева направо без учета приоритета операций.

Задача 1-9-28. Решите задачу 1-9-27 со следующими усложнениями:

1) можно использовать скобки для группировки операций;
2) можно изменять порядок следования чисел в выражении;
3) можно «склеивать» несколько подряд идущих чисел для образования многозначного числа.

Задача 1-9-29. Реализуйте аналог игры «Пятнашки» на поле 3 на 3.

Указание. Будем считать, что массив 3×3 заполнен числами от 0 до 8, причем 0 соответствует пустому полю. Требуется найти самую короткую последовательность шагов, приводящую заданный набор к упорядоченному (если читать по строкам) виду $1, 2, \ldots, 8, 0$.

— А что такое рекурсия?
— Да посмотри в Википедии.
— Уже смотрел... Написано: «См. Рекурсия».

1.10. Динамическое программирование

Ключевая идея динамического программирования напоминает метод математической индукции: явно решаем исходную задачу в некоторых тривиальных случаях и на этой основе строим общее решение для произвольных входных данных. При этом реализации соответствующих алгоритмов по форме часто

совпадают с реализацией рекуррентных процедур на основе вспомогательных массивов.

Задача 1-10-1. Найдите $C_N^K = \frac{N!}{(N-K)!K!}$ для целых N, K, $N \geqslant K \geqslant 0$.

Решение. Воспользуемся следующей рекуррентной формулой:

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k, \quad C_n^0 = C_n^n = 1.$$

В данном случае исходная задача вычисления C_n^k сведена к двум аналогичным задачам «предыдущего уровня». Это позволяет последовательно для $n = 0, 1, \ldots, N$ определить все наборы $\{C_n^k,\ k = 0, \ldots, n\}$.

```
#define N 10
unsigned long int C[N + 1];
C[0] = 1;
for (k = 1; k <= N; k++)
  C[k] = 0;
for (n = 1; n <= N; n++)
  for (k = n; k > 0; k--) C[k] = C[k] + C[k - 1];
```

Сравните данный подход, рекурсивную реализацию и вычисления по явной формуле; для каждого из методов экспериментально найдите такое N, что величина $C_N^{N/2}$ правильно и за разумное время вычисляется на основе типа **unsigned int**.

Задача 1-10-2. В куче лежит N камней. Два игрока по очереди делают ход: берут некоторое количество камней, от 1 до M. Проигрывает тот, кто не сможет сделать свой ход (т. е. перед ним «пустая куча»). Реализуйте алгоритм, позволяющий для заданных положительных N и M определить победителя при правильной игровой стратегии.

Решение. Будем в n-ю ячейку массива `int a[N+1]` записывать 1, если для кучи из n камней у первого игрока имеется выигрышная стратегия, и 2 в противном случае (т. е. при наличии выигрышной стратегии у второго игрока при любой игре первого):

```
int a[N + 1];
a[0] = 2;
for (n = 1; n <= N; n++) {
  a[n] = 2;
  for (m = 1; m <= M; m++)
    if (a[n - m] == 2) {
```

```
        a[n] = 1;
        break;
      }
  }
```

Отметим, что проверка `if (m > n) break;` не требуется, так как $a[0] = 2$. В данном случае $a[k(M+1)] = 2$, $k = 0, 1, \ldots$, т. е. у первого игрока нет выигрышной стратегии, если число камней кратно $M+1$. Усложните правила, считая, что если число камней в куче кратно трем, то у игрока есть возможность забрать треть камней.

Задача 1-10-3. Решите задачу 1-10-2 при условии, что на каждом шаге игрок может взять из кучи 1, 3, 5 камней либо половину (если число камней в куче на данный момент четно).

Задача 1-10-4. Пусть в условиях задачи 1-10-3 каждый игрок всегда реализует оптимальную стратегию: выигрывая, он выбирает вариант с наименьшим числом ходов, а проигрывая — с наибольшим. Для заданного положительного N найдите, через сколько ходов закончится игра в этом случае.

Указание. Будем в n-ю ячейку массива `int s[N+1]` записывать искомый ответ: если $a[n] = 1$, то $s[n]$ равно 1 плюс минимум из $s[k_j]$ по всем доступным на текущем ходе выигрышным вариантам $a[k_j] = 2$; при $a[n] = 2$ значение $s[n]$ выбирается как 1 прюс максимум из $s[k_j]$ по всем доступным проигрышным вариантам $a[k_j] = 1$. При этом $s[0] = 0$, что формально соответствует выигрышу второго игрока через нуль шагов.

Задача 1-10-5. Реализуйте решение задачи 1-10-4 на основе одного знакопеременного массива `int s[N+1]`. Можно ли получить ответ, сохраняя в `s[n]` положительные числа (т. е. количество шагов до окончания игры)?

Задача 1-10-6. В куче лежит N камней. Два игрока по очереди делают ход: берут некоторое число камней, от 1 до M, но не больше, чем взял предыдущий игрок. Проигрывает тот, кто не сможет сделать свой ход. Реализуйте алгоритм, позволяющий для заданных положительных N и M определить победителя при правильной стратегии.

Решение. В ячейку с индексом n, m массива `int a[N+1][M+1]` запишем 1, если для кучи из n камней при условии, что на предыдущем ходе второй игрок взял m камней, у первого игрока имеется выигрышная стратегия; иначе запишем 2.

```
int a[N + 1][M + 1];
int n, m, m1;
for (n = 0; n <= N; n++)
  for (m = 0; m <= M; m++)
    a[n][m] = 2;
for (n = 1; n <= N; n++)
  for (m = 1; m <= M; m++)
    for (m1 = 1; m1 <= m; m1++)
      if (a[n - m1][m1] == 2) {
        a[n][m] = 1;
        break;
      }
```

Если в n-й строке полученной таким образом матрицы имеется хотя бы одна единица, то игрок, начинающий игру с кучей из n камней, имеет выигрышную стратегию. В этом случае можно формально записать единицу в ячейку `a[n][0]`:

```
for (n = 0; n <= N; n++)
  for (m = 1; m <= M; m++)
    if (a[n][m] == 1) {
      a[n][0] = 1;
      break;
    }
```

Задача 1-10-7. Имеются две кучи с N_1 и N_2 камнями. Два игрока по очереди делают ход: игрок может взять из произвольной кучи 1, 3, 5 камней либо половину (если число камней в куче на данный момент четно). Проигрывает тот, кто не сможет сделать свой ход. Реализуйте алгоритм, позволяющий для заданных положительных N_1, N_2 определить победителя при правильной стратегии.

Задача 1-10-8. Для заданного N найдите количество M_N различных цепочек длины N, состоящих только из нулей и единиц и не содержащих двух подряд идущих единиц, а также распечатайте все такие цепочки.

Указание. Рекуррентная формула для M_N совпадает с формулой для чисел Фибоначчи, т. е. имеет вид

$$M_n = M_{n-1} + M_{n-2}, \quad M_1 = 2, \quad M_2 = 3.$$

Действительно, пусть $M_{n-1} = M^0_{n-1} + M^1_{n-1}$, где M^0_{n-1} и M^1_{n-1} равны количеству последовательностей длины $n-1$,

заканчивающихся на 0 и 1 соответственно. Тогда ноль допустимо приписать к каждой из M_{n-1} последовательностей, а единицу только к каждой из M^0_{n-1} последовательностей, т. е. $M_n = M_{n-1} + M^0_{n-1}$. При этом $M^0_{n-1} = M_{n-2}$, так как ноль мы могли приписать к любой из M_{n-2} последовательностей.

Для построения требуемых цепочек можно создать массив s[M][N], $M = M_N$, и последовательно для $n = 1, \ldots, N$ заполнять его так, что на n-м шаге в него будут записываться все правильные цепочки длины n. Действительно, для $n = 1$ имеем две цепочки: $s[0][0] = 0$, $s[1][0] = 1$. Переход от найденного набора всех последовательностей длины n к последовательностям длины $n + 1$ очевиден: если текущая цепочка заканчивается единицей, то ее продолжаем нулем; иначе (если цепочка заканчивается нулем) цепочку дублируем и, добавив к оригиналу нуль, а к копии единицу, получаем две новые цепочки. Эффективную по памяти программу можно построить, реализовав работу с массивом s[M][N] как с одномерным битовым массивом подходящей длины (см. задачу 1-7-5).

Так как в данном случае требуется распечатать (а не сохранить) получающиеся цепочки, то можно завести одномерный массив `char bstr[N]` и реализовать решение в виде рекурсивной функции типа

```
void PrnBitsStr(char *bstr, int n, int N),
```

считая, что n содержит длину верно заполненной головной части последовательности. Если n становится равным N, то печатается содержимое массива `bstr` и рекурсивные вызовы завершаются.

Задача 1-10-9. Для заданного n распечатайте все правильные скобочные последовательности, состоящие из n открывающих и n закрывающих скобок, а также явно найдите их количество C_n. Например, для $n = 1$ имеем одну последовательность, (); для $n = 2$ — две последовательности, (()), ()(); для $n = 3$ — пять последовательностей, ((())), (()()), (())(), ()(()), ()()(), и т. д.

Указание. Каждую правильную скобочную последовательность X можно записать в виде X = (Y)Z, где Y, Z — правильные (возможно, пустые) скобочные последовательности. При этом Y будет состоять из $n - 1$ пар скобок при пустой Z, из $n - 2$ пар скобок при односкобочной Z и т. д. до пустой Y и состоящей из $n - 1$ пар скобок Z. Отсюда следует рекуррентное соотношение для количества C_n правильных скобочных последовательностей из n пар скобок:

$C_n = C_0C_{n-1} + C_1C_{n-2} + \ldots + C_{n-2}C_1 + C_{n-1}C_0,$
$C_0 = C_1 = 1.$

Здесь исключительно для симметрии формулы добавлен множитель $C_0 = 1$, т. е. формально доопределено, что имеется одна пустая скобочная последовательность. Алгоритм генерации всех возможных последовательностей строится аналогичным образом на основе перебора всех возможных представлений $X = (Y)Z$. Для контроля правильности полученного результата полезно использовать явную формулу $C_n = \frac{(2n)!}{n!(n+1)!}$.

Замечание. На данный момент известно более шестидесяти математических конструкций, приводящих к числам C_n, названным в честь Эжена Шарля Каталана. Свойства данной последовательности на полвека ранее изучал Леонард Эйлер при подсчете количества различных способов разрезания правильного n-угольника на треугольники по непересекающимся диагоналям (также см. задачу 1-10-18).

Задача 1-10-10. Пусть взвешенный ориентированный граф без циклов задан матрицей смежности `int s[n][n]`. Пусть его вершины занумерованы так, что при размещении их на прямой в порядке возрастания номеров все ребра направлены слева направо (такая топологическая упорядоченность возможна для произвольного ациклического графа). Найдите кратчайшие расстояния d[i], $i = 0, \ldots, n-1$, от нулевой вершины графа до всех остальных.

Идеи реализации. Будем считать, что все элементы массива `int d[n]` формально инициализированы недосягаемо большой положительной константой Inf (либо явно, либо косвенно через вспомогательный массив) и определена функция Sdist(j, i), возвращающая либо расстояние s[j][i] от вершины j до вершины i, либо Inf, если ребро из j в i отсутствует. Будем считать, что реализована функция MinDist(s, d, i), которая для заданного i перебирает множество E обработанных к данному моменту вершин (т. е. вершин с номерами j, $0 \leqslant j < i$) и возвращает $\min\limits_{E}(d[j] + Sdist(j, i))$. Тогда решение имеет вид

```
d[0] = 0;
for (i = 1; i < n; i++) d[i] = Inf;
for (i = 1; i < n; i++) d[i] = MinDist(s, d, i);
```

Задача 1-10-11. Из массива a[n] целых чисел выделите строго возрастающую подпоследовательность наибольшей длины.

Указание. Поставьте в соответствие последовательности a[n] ориентированный ациклический граф «всех соседей справа», формально считая, что из вершины с номером i имеется ребро единичной длины в вершину с номером j, если только $a[i] \leqslant a[j]$. В этом случае исходная задача сводится (см. задачу 1-10-10) к вычислению элементов массива d[i], $i = 0, \ldots, n$, содержащих для каждого i наибольшее расстояние от нулевой вершины графа до i-й, и последующему выбору $\max\limits_{0 \leqslant i \leqslant n} d[i]$.

Задача 1-10-12. Определите, сколько палиндромов содержится в заданной символьной строке `char s[n]`.

Указание. Если реализовать функцию подсчета всех палиндромов с центром в `s[i]` и ее модификацию для работы с палиндромами четной длины, то окончательная сложность составит $O(n^2)$ действий. Для решения задачи с линейной асимптотикой можно применить алгоритм Манакера.

Задача 1-10-13. Определите, можно ли набрать сумму S, используя только монеты достоинством $a_0, \ldots, a_{n-1}$ при условии, что имеется ровно одна монета каждого вида.

Указание. По сути, требуется выполнить перебор всех вариантов (см. задачу 1-7-15). Для этого, например, можно создать нулевой массив `v[S+1]`, а затем положить `v[s]=1`, если сумму `s` набрать возможно. Имея пустое множество монет, можно составить только нулевую сумму, т. е. $v[0] = 1$. Если дана единственная монета a_0, то дополнительно положим $v[0 + a_0] = 1$. Пусть задача решена для набора $a_0, \ldots, a_{k-1}$. Тогда при добавлении монеты a_k следует выбрать из массива все $v[s] = 1$ и для каждого такого s положить $v[s + a_k] = 1$. Повторяя данную процедуру для $k = 1, 2, \ldots, n$, мы получим решение исходной задачи.

Задача 1-10-14. Определите, можно ли набрать сумму S, используя только монеты номиналом $a_0, \ldots, a_{n-1}$ при условии, что имеется неограниченное количество монет каждого вида.

Указание. Создадим нулевой (например, битовый) массив `v[S+1]`. Если сумму s, $0 \leqslant s \leqslant S$, набрать возможно, положим s-й элемент равным единице. Массив заполняется последовательно для $s = 0, 1, \ldots, S$. При $s = 0$ имеем $v[0] = 1$. Пусть задача решена для некоторого $s \geqslant 0$. Для вычисления значения $v[s + 1]$ рассмот-

рим все допустимые величины $v[s+1-a_i]$, $i=0,\ldots,n-1$. Если среди них найдется хотя бы одна единица, задаем $v[s+1]=1$.

Задача 1-10-15. Пусть имеется неограниченное количество монет номиналом $a_0,\ldots,a_{n-1}$. Какое минимальное количество монет M потребуется взять, чтобы набрать сумму S?

Указание. Жадный алгоритм (на каждом шаге берем наибольшее количество монет максимального возможного номинала) в общем случае дает ошибочный ответ. Правильное решение можно реализовать на основе динамического алгоритма (см. задачи 1-10-13 и 1-10-14). Создадим массив `M[S+1]` и инициализируем все его элементы значением -1, которое будет соответствовать недосягаемо большой величине. Тогда $M[s]=$ $=\min\{M[s-a_0],\ldots,M[s-a_{n-1}]\}+1$, $M[0]=0$. Здесь необходимо контролировать выход за нижнюю границу массива и правильно обрабатывать значение -1.

Задача 1-10-16. Пусть задана целочисленная матрица a[M][N]. Считая, что из ячейки a[i][j] за один шаг можно переместиться либо в `a[i+1][j]`, либо в `a[i][j+1]`, вычислите количество различных путей из `a[0][0]` в `a[M-1][N-1]`; найдите значение максимальной возможной суммы $\sum_{k=1}^{N+M-1} a[i_k][j_k]$ вдоль такого пути и распечатайте соответствующий путь.

Решение. Последовательно заполним вспомогательный массив **int** `s[M][N]` так, что в `s[i][j]` будет храниться количество путей из ячейки `a[i][j]` в ячейку `a[M-1][N-1]`. Если `s[i+1][j]` и `s[i][j+1]` найдены, то `s[i][j]` $=$ $=$ `s[i+1][j]` $+$ `s[i][j+1]`. Следовательно, можно обработать последнюю строку `s[M-1][j]` для j от N $-$ 1 до 0 и последний столбец `s[i][N-1]` для i от M $-$ 1 до 0; далее заполнить предпоследнюю строку `s[M-2][j]` и продолжать до строки `s[0][j]`. По окончании работы в `s[0][0]` будет записано искомое число.

```
for (j = N - 1; j >= 0; j--) s[M - 1][j] = 1;
for (i = M - 1; i >= 0; i--) s[i][N - 1] = 1;
for (i = M - 2; i >= 0; i--)
  for (j = N - 2; j >= 0; j--)
    s[i][j] = s[i + 1][j] + s[i][j + 1];
```

Отметим, что для решения данной задачи достаточно одномерного массива (см. задачу 1-10-1). Более того, можно заметить, что элементы матриц `s[i][j]` преобразуются по правилу

построения треугольника Паскаля, и поэтому искомый ответ вычисляется явно: $s[0][0] = C_{N+M-2}^{N-1}$.

Для решения второй части задачи можно применить аналогичный подход. Будем в ячейке массива s[i][j] хранить значение максимальной возможной суммы, которую можно получить, перемещаясь из ячейки (i, j) в ячейку (M, N). Тогда

```
s[M - 1][N - 1] = a[M - 1][N - 1];
for (j = N - 2; j >= 0; j--)
  s[M - 1][j] = s[M - 1][j + 1] + a[M - 1][j];
for (i = M - 1; i >= 0; i--)
  s[i][N - 1] = s[i + 1][N - 1] + a[i][N - 1];

for (i = M - 2; i >= 0; i--)
  for (j = N - 2; j >= 0; j--)
    s[i][j] = max(s[i + 1][j],
                  s[i][j + 1]) + a[i][j];
```

По окончании работы искомый путь восстанавливается по заполненной матрице s[i][j].

Задача 1-10-17. Пусть задана целочисленная матрица a[M][N]. Пусть из ячейки a[i][j] за один шаг можно переместиться только в одну из ячеек a[i + 1][j], a[i][j + 1], a[i + 1][j + 1] и только при условии, что содержимое соответствующей ячейки отлично от нуля (т. е. проход по ячейке с нулевым значением запрещен). Вычислите количество различных допустимых путей из a[0][0] в a[M − 1][N − 1], найдите значение максимальной возможной суммы $\sum$ a[i][j] вдоль такого пути, распечатайте соответствующий путь.

Указание. См. решение задачи 1-10-16.

Задача 1-10-18. Пусть задана целочисленная матрица a[M][N]. Пусть из ячейки a[i][j] за один шаг можно переместиться только в одну из ячеек a[i + 1][j], a[i][j + 1] и только при условии, что соответствующая ячейка находится не выше диагонали. Вычислите количество различных допустимых путей из a[0][0] в a[M − 1][N − 1] и распечатайте все такие пути.

Указание. Установите взаимно однозначное соответствие между множеством допустимых путей указанного вида и правильными скобочными последовательностями (см. решение задачи 1-10-7).

Задача 1-10-19. По заданному натуральному W и набору пар натуральных элементов $\{(p_i, w_i),\ i = 1, \ldots, n)\}$ постройте такое подмножество пар $\{(p_{i_j}, w_{i_j}),\ j = 1, \ldots, k\}$, возможно с повторениями, что величина $S_p = \sum_{j=1}^{k} p_{i_j}$ достигает наибольшего значения при условии $S_w = \sum_{j=1}^{k} w_{i_j} \leqslant W$. Таким образом, требуется решить задачу максимизации суммы S_p, обычно называемой стоимостью рюкзака, при ограничении $S_w \leqslant W$ на сумму S_w, называемую весом рюкзака, при условии, что каждый элемент множества можно использовать произвольное количество раз.

Указание. Последовательно найдите решение исходной задачи при ограничениях $S_w \leqslant 0$, $S_w \leqslant 1, \ldots$, $S_w \leqslant W$. Для этого создайте массив `int Sp[W+1]` и заполните его так, что `Sp[v]` содержит максимальную возможную стоимость при допустимом весе $S_w \leqslant v$:

```
Sp[0] = 0;
for (v = 1; v <= W; v++)
  Sp[v] = MaxPrice(Sp, w, p, n, v);
```

Функция `MaxPrice()` возвращает $\max\limits_{w[i] \leqslant v} \{Sp[v-w[i]] + p[i]\}$.

Задача 1-10-20. Решите задачу максимизации стоимости рюкзака (см. задачу 1-10-19) при условии, что каждый элемент множества можно использовать не более одного раза.

Указание. Создайте и заполните массив `int Sp[W+1][n+1]` так, что `Sp[v][j]` содержит максимальную стоимость при допустимом весе $S_w \leqslant v$ на наборе $\{(p_i, w_i),\ i = 1, \ldots, j)\}$:

```
for (j = 0; j <= n; j++)
  Sp[0][j] = 0;
for (v = 0; v <= W; v++)
  Sp[v][0] = 0;
for (j = 1; j <= n; j++)
  for (v = 1; v <= W; v++)
    if (w[j] > v) {
      Sp[v, j] = Sp[v, j - 1];
    } else {
      Sp[v, j] = Sp[v - w[j], j - 1] + p[j];
      Sp[v, j] = max(Sp[v, j - 1], Sp[v, j]);
    }
```

Искомое значение будет получено в Sp[W][n].

Реализуйте аналогичный алгоритм, последовательно заполняя матрицу `Sv[p][j]`, содержащую минимально возможный вес рюкзак со стоимостью p, сформированного на наборе $\{(p_i, w_i),\ i = 1, \ldots, j)\}$.

Задача 1-10-21. Для взвешенного ориентированного графа $g^{(N)}$, имеющего N вершин и заданного, например, матрицей смежности, найдите длину кратчайшего пути $s(g^{(N)})$, начинающегося из нулевой вершины и проходящего через все остальные ровно один раз.

Указание. На первом шаге для каждого из подграфов $\{g_k^{(2)}\}_{k=1}^{N-1}$, состоящих из двух вершин и содержащих начальную (нулевую) вершину, решите задачу о поиске длины кратчайшего пути $s(g_k^{(2)}, j)$, начинающегося в нулевой вершине и заканчивающегося в j-й, $j = 1, \ldots, N-1$. Имея множество таких решений $\{s(g_k^{(2)}, j)\}$, несложно решить аналогичную задачу следующего уровня: для каждого из подграфов $\{g_k^{(3)}\}$, состоящих из трех вершин и содержащих нулевую вершину, найти длину кратчайшего пути $s(g_k^{(3)}, j)$, начинающегося в нулевой вершине и заканчивающегося в j-й, $j = 1, \ldots, N-1$. Действительно, соответствующее семейство $\{s(g_k^{(3)}, j)\}$ может быть получено прямым перебором: $s(g_k^{(3)}, j) = \min\limits_k (s(g_k^{(2)}, i) + d[i][j])$. Аналогично строится семейство решений $\{s(g_k^{(4)}, j)\}$, $j = 1, \ldots, N$, и т. д. Данная «задача коммивояжера» относится к числу NP-полных задач и может быть решена предложенным алгоритмом за $O(N^2 2^N)$ действий. Это существенно лучше полного перебора множества из $(N-1)!$ допустимых путей, но все же недопустимо много уже при N порядка сотни даже для современных суперЭВМ.

Три программиста едут в автобусе без билетов. Заходит кондуктор и к первому:
— Ваш билетик!
Тот кивает: «Сзади». Кондуктор ко второму:
— Ваш билетик!
Тот кивает: «Сзади». Кондуктор к третьему:
— Ваш билетик!
Тот оборачивается — сзади стенка.
— А я с ними.

Мораль: изучайте динамическое программирование!

Глава 2

ЧИСЛЕННЫЕ АЛГОРИТМЫ

Реализация вычислительных алгоритмов представляет собой важный, но достаточно специфический раздел программирования. Дело в том, что только конечное множество чисел $F \subset \mathbf{R}$ представляется в ЭВМ точно, а остальные числа округляются до некоторого числа из F. Для большинства современных ЭВМ каждое число x, принадлежащее F, имеет вид

$$x = \pm \left(\frac{d_0}{2^0} + \frac{d_1}{2^1} + \frac{d_2}{2^2} + \ldots + \frac{d_t}{2^t} \right) 2^q, \quad d_i = 0, 1.$$

При этом число в скобках называют *мантиссой* (дробной частью) числа x, d_i — *разрядами мантиссы*, t — *длиной мантиссы*, q — *порядком* (*экспонентой*) *числа*. Такая структура чисел множества F приводит к тому, что на прямой $\mathbf{R}$ они расположены неравномерно: сгущаются к нулю и встречаются все реже при удалении от нуля. При этом относительная погрешность округления всегда остается ограниченной некоторой постоянной величиной, называемой машинной точностью.

Диапазон изменения величин t и q ограничен и зависит от способа представления данных для конкретной ЭВМ. На распространенных на данный момент моделях компьютеров число типа **double** занимает 8 байт: 1 бит отвечает за знака, 11 бит отводятся для показателя, 52 — для мантиссы (старший бит d_0 мантиссы обычно равен 1 и в памяти не запоминается). В результате имеем следующий диапазон для ненулевых значений |x|: $[2^{-2^{10}}, 2^{2^{10}}]$, или приблизительно $[10^{-308}, 10^{308}]$, а также формальные сложности представления нуля (так как всегда $d_0 = 1$), который к тому же бывает двух видов, ± 0. Кроме нуля, имеются еще два выделенных «специальных числа»: Inf (infinity — бесконечность) и NaN (Not a Number — не число). Первое может получиться, например, в результате деления ненулевого числа на ноль, второе — в результате извлечения квадратного корня из отрицательного числа. Отметим, что реально используемый в ЭВМ стандарт IEEE 754-2008 имеет множество дополнительных тонких моментов, останавливаться на которых мы не будем.

Так как при сложении/вычитании чисел сначала производится выравнивание порядков и только потом выполняется арифметическое действие, то найдется такое наибольшее $\varepsilon \in F$, называемое машинной точностью, что $\varepsilon > 0$, но в машинной арифметике выполняется равенство $1 + \varepsilon = 1$. При этом ε ограничивает

относительную погрешность, с которой производятся округления действительных чисел в множество F. Для машин с указанной архитектурой имеем $\varepsilon \approx 10^{-15}$, что существенно больше, чем минимальное положительное машинное число 10^{-308}. Величину ε можно оценить следующим образом:

```
double eps = 1.; while(1. + eps > 1.) eps /= 2.;
```

Однако представление чисел в регистрах может иметь большее число разрядов, чем в оперативной памяти, и точность арифметики на регистрах может оказаться выше.

Все сказанное выше означает, что между математически точными вычислениями и вычислениями на ЭВМ имеется принципиальное отличие, поэтому алгоритмы, традиционно применяемые в точной арифметике, могут некорректно работать при расчетах на ЭВМ. Как следствие, к методам и постановкам задач вычислительной математики предъявляют дополнительные требования.

1. Решаемая численно задача должна быть устойчива, т. е. малое изменение входных параметров не должно принципиально менять результат.
2. Должен быть численно устойчив выбранный алгоритм, т. е. ошибки округления в промежуточных вычислениях не должны искажать окончательный ответ.
3. Имеющиеся вычислительные ресурсы (память, быстродействие, программное обеспечение) должны позволить реализовать алгоритм и получить ответ за требуемое время.

2.1. Вычислительная погрешность

Задача 2-1-1. Напишите программу, определяющую машинную точность представления вещественных чисел в форматах **float** и **double**, а также соответствующие данным типам наименьшие и наибольшие по модулю значения.

Ответ. Для **float**: приблизительный диапазон ненулевых значений $|x| — [3{,}4 \cdot 10^{-38}; 3{,}4 \cdot 10^{38}]$, мантисса — 7 знаков (удобно использовать форматы печати `"%12.7f"`, `"%12.7e"`); для **double**: приблизительный диапазон ненулевых значений $|x| — [1{,}7 \cdot 10^{-308}; 1{,}7 \cdot 10^{308}]$, мантисса — 15 знаков (удобно использовать форматы печати `"%.15lf"`, `"%25.15le"`). На данный момент тип **long double** обычно совпадает с типом **double**.

Задача 2-1-2. Можно ли непосредственными вычислениями проверить, что ряд $\sum\limits_{j=1}^{\infty} \frac{1}{j}$ расходится?

Ответ. Нет. Более того, нестрого можно сказать, что при расчетах на ЭВМ любой ряд с убывающими по модулю элементами ведет себя как сходящийся.

Задача 2-1-3. Для вычисления суммы

$$S(10^3) = \sum_{n=2}^{10^3} \frac{1}{(n+1)(n-1)}$$

можно использовать формулы

$$S_1 = 0, \quad S_n = S_{n-1} + \frac{1}{(n-1)(n+1)}, \quad n = 2, \ldots, 10^3,$$

$$R_{10^3} = 0, \quad R_{n-1} = R_n + \frac{1}{(n-1)(n+1)}, \quad n = 10^3, \ldots, 2.$$

Найдите точное значение $S(10^3)$ с учетом

$$\frac{1}{(n-1)(n+1)} = \frac{1}{2}\left(\frac{1}{n-1} - \frac{1}{n+1}\right)$$

и сравните величины S_{10^3}, R_1, $S(10^3)$. Объясните полученный результат.

Задача 2-1-4. Существует единственное число Эйлера $e \approx$ $\approx 2{,}71828182845904523\ldots$, для которого выполняется $(e^x)' = e^x$ и имеет место разложение

$$e^x = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^k}{k!} + \ldots = S_\infty(x).$$

Формально вычислите суммы $S_k(x)$ для достаточно большого k при $x = 1, -15, -20$, т. е. найдите приближенные значения e^1, e^{-15}, e^{-20}. Сравните результаты со значениями функции `exp(x)` из математической библиотеки (подключите заголовочный файл `math.h` и используйте ключ `-lm` при сборке исполняемого exe-файла) и объясните, почему относительная погрешность для e^1 существенно меньше, чем для e^{-15}, e^{-20}. Улучшите алгоритм для отрицательных показателей с учетом формулы $e^{-x} = 1/e^x$.

Задача 2-1-5. Для произвольной достаточно гладкой функции $f(x)$ и всякой точки x_0 при малых h выполняется оценка

$$\left|\frac{f(x_0+h) - f(x_0)}{h} - f'(x_0)\right| = \varphi(h) \leqslant C \cdot |h|,$$

где константа C зависит только от f, x_0. Например, для $f(x) = e^x$ и $x_0 = 1$ при $|h| \leqslant 1$ можно положить $C = 4$. Постройте таблицу значений для функции $\varphi(h)$ при $f(x) = e^x$, $f'(x) = e^x$, $x_0 = 1$, $h = 10^{-k}$, $k = 1, \ldots, 14$. Объясните, почему с некоторого k значения $\varphi(h)$ начинают расти, что формально противоречит указанной оценке.

Задача 2-1-6. Пусть ищется наименьший корень x_1 уравнения

$$y(x) = (x - 10^{-3})(x - 10^3) = 0,$$

т. е. уравнения

$$x^2 - 1000{,}001\,x + 1 = 0.$$

Какая из двух формул,

$$x_1^{(1)} = a - \sqrt{a^2 - 1}, \quad x_1^{(2)} = \frac{1}{a + \sqrt{a^2 - 1}}, \quad \text{где} \quad a = \frac{1000{,}001}{2},$$

дает более точный результат?

Решение. Вторая формула представляет собой результат избавления от иррациональности в числителе первой формулы, во втором случае точность результата значительно выше. В первом случае приходится вычитать большие по модулю и близкие друг к другу числа, что приводит к *эффекту пропадания значащих цифр*, часто существенно искажающему конечный результат вычислений. Абсолютная погрешность также увеличивается, если выполняется *деление на малое* (*умножение на большое*) число. Еще одна опасность — *выход за диапазон допустимых значений* в промежуточных вычислениях (что произойдет, например, после умножения исходного уравнения на достаточно большое число).

Задача 2-1-7. Составьте функцию для вычисления корней квадратного уравнения $ax^2 + bx + c = 0$ с прототипом

```
int SolveQuadEq(double a, double b, double c,
                double *x1, double *x2);
```

Здесь `a`, `b`, `c` — коэффициенты уравнения, `x1`, `x2` — указатели на переменные для размещения результата. Возвращаемые значения: 1 — уравнение имеет два действительных корня, 0 — уравнение имеет один корень, −1 — корни уравнения комплексные, −2 — корней нет. В случае комплексных корней пусть `*x1` содержит действительную часть, а `*x2` — мнимую. Функция должна корректно обрабатывать всевозможные значения коэффициентов

a, b, c. Например, случай $a = b = 0$, $c \neq 0$ — это случай отсутствия корней.

Идеи реализации. При использовании традиционных формул нахождения корней квадратного уравнения может возникнуть существенная потеря точности. Следует преобразовать формулы так, чтобы не вычислялась разность двух близких больших чисел (это можно сделать, умножив числитель и знаменатель одной из формул на множитель, сопряженный к числителю). Другая опасность — получить переполнение в результатах промежуточных операций. Для устранения этой опасности следует предварительно определить сравнительные порядки коэффициентов и нормировать их заменой $a_1 = ka$, $b_1 = kb$, $c_1 = kc$ или масштабировать заменой $\tilde{x} = kx$ с подходящим коэффициентом k, чтобы по возможности сделать коэффициенты уравнения близкими по порядку величинами.

Протестируйте работу функции для следующих значений:

$$\begin{aligned} &a = 1, && x_1 = -10^5, && x_2 = 10^{-5};\\ &a = 10^{300}, && x_1 = 1, && x_2 = 2;\\ &a = 1, && x_1 = 1{,}99999999, && x_2 = 2{,}00000001. \end{aligned}$$

Задача 2-1-8. Наименьшее по модулю собственное значение матрицы Уилкинсона

$$A = \begin{pmatrix} 20 & 20 & 0 & 0 & \dots & 0 & 0 \\ 0 & 19 & 20 & 0 & \dots & 0 & 0 \\ 0 & 0 & 18 & 20 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 2 & 20 \\ \varepsilon & 0 & 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

при $\varepsilon = 0$ равно 1. Как оно изменится при $\varepsilon = 20^{-19} \cdot 20! \approx \approx 5 \cdot 10^{-7}$?

Решение. Характеристическое уравнение имеет вид

$$\det(A - \lambda I) = (20 - \lambda)(19 - \lambda) \cdots (1 - \lambda) - 20^{19} \cdot \varepsilon = 0.$$

Свободный член в этом уравнении равен 0, следовательно, наименьшее собственное значение также равно 0. Таким образом, задачи вычисления собственных чисел и определителя для данной матрицы являются численно неустойчивыми: незначительная погрешность в элементах матрицы может существенно исказить ответ. В данном случае изменение одного

элемента матрицы 20×20 на величину порядка $5 \cdot 10^{-7}$ привело к изменению величины определителя с $20! \approx 2{,}4 \cdot 10^{18}$ до нуля. Для сравнения: $5 \cdot 10^{-7}$ *км* $= 0{,}5$ *мм*, в приблизительно 10^{18} *км* оценивается диаметр нашей галактики Млечный Путь.

Задача 2-1-9. Метод Крамера решения систем линейных алгебраических уравнений с невырожденной матрицей размера $n \times n$ позволяет найти точное решение, вычислив $n + 1$ определитель матриц размера $n \times n$. Оцените число арифметических действий $N(n)$ и время $T(n)$ работы такого алгоритма для $n = 20$, $n = 100$ и сравните с числом атомов процессора и возрастом Вселенной соответственно.

Указание. Нахождение определителя является неустойчивой задачей (см. задачу 2-1-8), и уже поэтому метод применять не стоит. Для метода Крамера требуется вычислить $n + 1$ определитель; если вычислять определитель по определению (т. е. методом перестановок), то потребуется порядка $n \cdot n!$ арифметических действий.

— Ну и чему тебя на мехмате учат?
— На уроках математики, что $1/3 + 1/3 = 2/3$, *а на занятиях по программированию, что* $1/3 + 1/3 = 0$.
— Они что, между собой не смогли договориться?

2.2. Суммирование рядов и вычисление элементарных функций

Суммирование ряда — процесс весьма чувствительный к накоплению вычислительной погрешности. Чем медленнее сходится числовой ряд, тем более вероятно получить ответ, не имеющий ничего общего с истинной суммой. Следовательно, для вычисления суммы, возможно, придется преобразовать ряд к другому виду, более подходящему для машинного суммирования.

Задача 2-2-1. Напишите программу для вычисления частичной суммы $\sum_{1}^{n} a_k$ заданного числового ряда в прямом и обратном направлении (т. е. для $k = 1, \ldots, n$ и $k = n, \ldots, 1$). На ее основе реализуйте автоматический поиск такого n, что результаты суммирования в разных направлениях будут значительно отли-

чаться. Протестируйте работу программы для следующих рядов:

$$\sum_k \frac{1}{\sqrt{k}}, \quad \sum_k \frac{\ln k}{k}, \quad \sum_k \frac{1}{k}, \quad \sum_k \frac{1}{k^2}, \quad \sum_k \frac{1}{k!}.$$

Как изменится результат, если каждое слагаемое умножить на $(-1)^k$?

Задача 2-2-2. Напишите программу, вычисляющую сумму

$$S(x) = \sum_{k=1}^{\infty} \frac{1}{k(k+x)}, \quad x \geqslant 0,$$

с точностью 10^{-8}.

Идеи реализации. Определим, сколько слагаемых данного ряда обеспечивает требуемую точность. Из известной оценки для остаточной суммы ряда

$$\sum_{k=n}^{\infty} \frac{1}{k(k+x)} \leqslant \frac{1}{(n+1)(n+1+x)} + \int_{n+1}^{\infty} \frac{1}{t(t+x)}\,dt$$

и из условия точности следует, что потребуется взять порядка 10^8 слагаемых. Для ускорения сходимости исходного ряда представим его в виде

$$S(x) = \bigl(S(x) - S(1)\bigr) + S(1) = \overline{S}(x) + S(1),$$

где

$$\overline{S}(x) = (1-x)\sum_{k=1}^{\infty} \frac{1}{k(k+x)(k+1)}, \quad S(1) = \sum_{k=1}^{\infty}\left(\frac{1}{k} - \frac{1}{k+1}\right) = 1.$$

Легко проверить, что скорость сходимости нового ряда $\overline{S}(x)$ существенно выше и для достижения требуемой точности достаточно взять порядка 10^4 слагаемых. Попробуйте еще уменьшить полученное число.

Задача 2-2-3. Напишите программу для вычисления суммы

$$S(x) = \sum_{1}^{\infty} \frac{1}{k^2+1}$$

с точностью 10^{-8}, предварительно оценив время работы программы.

Идеи реализации. Используйте алгоритм решения задачи 2-2-2 и следующие соотношения:

$$\sum_{1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}, \quad \sum_{1}^{\infty} \frac{1}{k^4} = \frac{\pi^4}{90}.$$

Задача 2-2-4. Напишите программу, вычисляющую для сходящихся рядов задачи 2-2-1 сумму $\sum_{1}^{\infty} a_k$ с точностью 10^{-8}. Обоснуйте полученные результаты.

Задача 2-2-5. Напишите программу для вычисления произведения

$$S = \prod_{k=2}^{\infty} \left(1 - \frac{1}{k^2}\right).$$

Сравните полученный результат с точным ответом $S = 0{,}5$.

Задача 2-2-6. Напишите программу для вычисления

$$S = \prod_{k=1}^{\infty} \frac{4k^2}{(2k-1)(2k+1)}.$$

Сравните полученный результат с точным ответом $S = 0{,}5\pi$.

Задача 2-2-7. Реализуйте функции

```
double sin_e (double x, double eps),
double cos_e (double x, double eps),
double exp_e (double x, double eps),
double ln_e (double x, double eps),
```

которые вычисляют значения указанных элементарных функций в точке x с точностью ε методом суммирования соответствующего ряда Тейлора. Можно считать, что точность достигнута, если первое из отброшенных слагаемых a_{k+1} изменяет итоговую сумму менее чем на ε.

Идеи реализации. Хотя ряды Тейлора для функций sin, cos, exp сходятся для любого x, их непосредственное суммирование для больших x приводит к неверным результатом из-за накопления вычислительной погрешности. Опишем приемы снижения погрешности суммирования.

1. Для функций $\sin x$ и $\cos x$ сведем данное значение x к значению $x' \in [0, \pi/2]$, используя периодичность и формулы приведения. При совместной реализации этих двух функций формулы приведения позволяют уменьшить отрезок до $[0, \pi/4]$.

Вычисление очередных слагаемых рядов Тейлора следует проводить по рекуррентным соотношениям

$$a_1 = x,\ \ a_{k+1} = -a_k \frac{x^2}{2k(2k+1)} \quad \text{для} \quad \sin x,$$

$$a_1 = 1,\ \ a_{k+1} = -a_k \frac{x^2}{2k(2k-1)} \quad \text{для} \quad \cos x.$$

2. Для функции e^x представим аргумент $x > 0$ в виде суммы целой и дробной части $x = [x] + \{x\}$ и будем вычислять $e^x = e^{[x]} \cdot e^{\{x\}}$. Для вычисления $e^{[x]}$ запасем константу $2{,}718281828459045235 \approx e$ и применим алгоритм быстрого возведения в целую степень (задача 1-7-13). Для $e^{\{x\}}$ можно суммировать ряд Тейлора с учетом рекуррентного соотношения для слагаемых $a_{k+1} = a_k \cdot \{x\}/k$. При $x < 0$ используем равенство $e^{-x} = 1/e^x$.

3. Для натурального логарифма ряд Тейлора имеет вид

$$\ln(1+y) = y - \frac{y^2}{2} + \frac{y^3}{3} - \frac{y^4}{4} + \ldots$$

и сходится для $|y| < 1$. Таким образом, следует выразить $\ln x$ через $\ln(1+y)$ с возможно меньшим y. Для этого запасем константу $1{,}6487212707001282 \approx \sqrt{e}$ и подберем число n так, чтобы представить $x > 1$ в виде $x = (\sqrt{e})^n \cdot (1+y)$, $|y| < 1$. Тогда искомое значение вычисляется как $\ln x = n/2 + \ln(1+y)$. Логарифм в правой части этого равенства вычисляется по ряду Тейлора. Для $0 < x < 1$ используем равенство $\ln 1/x = -\ln x$.

Задача 2-2-8. Для функции $\operatorname{tg} x$ имеет место представление в виде цепной дроби

$$\operatorname{tg} x = \cfrac{1}{\cfrac{1}{x} - \cfrac{1}{\cfrac{3}{x} - \cfrac{1}{\cfrac{5}{x} - \cfrac{1}{\cfrac{7}{x} - \cfrac{1}{\ldots}}}}}$$

Реализуйте функцию, вычисляющую $\operatorname{tg} x$ по этому представлению, и проверьте, насколько быстро сходится процесс вычислений, т. е. сколько членов дроби надо взять для получения результата с заданной точностью при различных значениях x.

Программист помогает сестренке делать математику.
— Сколько будет 2^4, 2^6, 2^9, 2^{16}?
Он мгновенно:
— 16, 64, 512, 65536.
— Какой ты умный! А сколько 5^2?
Он через некоторое время:
— Кажется, что-то дробное должно получиться...

2.3. Полиномиальная интерполяция

Задача приближения функций обычно ставится следующим образом: заданы набор аргументов x_i и значения функции $y_i = f(x_i)$, $i = 0, \ldots, n-1$. Требуется вычислить приближенное значение функции в некоторой точке x.

Задача 2-3-1. Найдите коэффициенты интерполяционного полинома

$$P_{n-1}(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$$

из условий $P_{n-1}(x_i) = y_i$, $i = 0, \ldots, n-1$, и положите $f(x) \approx P_{n-1}(x)$. Проверьте работоспособность алгоритма для $f(x) = x^{n-1} + 1$, $e^{-(0{,}75-x)^2}$, $x \in [0, 1]$.

Указание. Коэффициенты могут быть найдены из решения следующей системы линейных алгебраических уравнений относительно n неизвестных $a_0, a_1, \ldots, a_{n-1}$:

$$\begin{cases} a_0 + a_1 x_0 + \ldots + a_{n-1} x_0^{n-1} = y_0, \\ a_0 + a_1 x_1 + \ldots + a_{n-1} x_1^{n-1} = y_1, \\ \quad \ldots \\ a_0 + a_1 x_{n-1} \ldots + a_{n-1} x_{n-1}^{n-1} = y_{n-1}. \end{cases}$$

Определителем этой системы является определитель Вандермонда, равный $\prod\limits_{i \neq j} (x_i - x_j)$ и отличный от нуля при $x_i \neq x_j$, $i \neq j$. Однако при больших n данная система близка к вырожденной, так как функции $\{1, x, \ldots, x^{n-2}, x^{n-1}\}$ «почти» линейно зависимы. Как следствие, задача становится некорректной, а вычислительная погрешность значительно искажает ответ, что приводит к ошибочному решению. Поэтому при численном построении интерполяционного полинома обычно не вычисляют

явным образом коэффициенты a_i, а используют эквивалентную запись интерполяционного многочлена в форме Лагранжа:

$$P_{n-1}(x) \equiv L_n(x) = \sum_{i=0}^{n-1} y_i \prod_{\substack{j=0 \\ j\neq i}}^{n-1} \frac{x - x_j}{x_i - x_j}.$$

Отметим, что при больших n стоит уделить внимание последовательности вычислений по данной формуле.

Задача 2-3-2. Реализуйте функцию

```
double Lagrange (double x, double *xi,
                 double *yi, int n)
```

для вычисления интерполяционного многочлена Лагранжа. Здесь `x` — точка, в которой вычисляется значение, `xi`, `yi` — массивы значений аргументов и функции, `n` — количество элементов в массивах `xi`, `yi`, возвращаемое значение — результат вычисления многочлена.

Задача 2-3-3. Численно покажите, что если для функции Рунге

$$f(x) = \frac{1}{25x^2 + 1}$$

на отрезке $[-1, 1]$ интерполяционный многочлен строить на равномерной сетке $x_i = x_{i-1} + h$, то

$$\max_{0{,}726\ldots \leqslant |x| < 1} |f(x) - L_n(x)| \to \infty \quad \text{при} \quad n \to \infty.$$

Проверьте, как изменится качество приближения, если взять так называемые узлы Чебышева:

$$x_i = \cos\left(\frac{2(n-i)-1}{2n}\pi\right), \quad i = 0, \ldots, n-1.$$

Указание. Постройте многочлен указанным в задаче 2-3-1 методом и сравните его значения со значениями функции $f(x)$ не только в узлах (где они совпадают при отсутствии вычислительной погрешности), но и в некоторых точках между узлами. Для чебышевских узлов максимальное отклонение по всем точкам отрезка стремится к нулю при увеличении n, если отсутствует вычислительная погрешность.

Задача 2-3-4. По заданным массивам значений x_i, y_i, $i = 0, \ldots, n-1$, вычислите приближенное значение функции в заданной точке x с помощью

1) кусочно-линейной интерполяции,

2) кусочно-квадратичной интерполяции.

Идеи реализации. Сначала определяется отрезок $[x_k, x_{k+1}]$, которому принадлежит заданная точка x. Затем на данном отрезке функция приближается многочленом первой степени (по значениям x_k, x_{k+1}, y_k, y_{k+1}) либо многочленом второй степени (по значениям x_k, x_{k+1}, x_{k+2}, y_k, y_{k+1}, y_{k+2} или x_{k-1}, x_k, x_{k+1}, y_{k-1}, y_k, y_{k+1}).

Задача 2-3-5. По заданным массивам значений аргумента x_i, функции y_i и производной функции $dy_i = f'(y_i)$, $i = 0, \ldots, n-1$, вычислите приближенное значение функции в заданной точке x с помощью эрмитовой интерполяции.

Идеи реализации. Сначала определяется отрезок $[x_k, x_{k+1}]$, которому принадлежит заданная точка x. Затем на данном отрезке функция приближается многочленом третьей степени так, чтобы на краях отрезка значения многочлена и его производной совпадали со значениями функции и ее производной. Для этого удобно представить искомый многочлен в виде

$$a(x - x_k)^3 + b(x - x_k)^2 + c(x - x_k) + d,$$

что приводит к системе уравнений для определения неизвестных коэффициентов a, b, c, d

$$\begin{cases} d = y_k, \\ c = dy_k, \\ ah_k^3 + bh_k^2 + ch_k + d = y_{k+1}, \\ 3ah_k^2 + 2bh_k + c = dy_{k+1}, \quad \text{где} \quad h_k = x_{k+1} - x_k. \end{cases}$$

Задача 2-3-6. По заданным массивам значений аргумента x_i и функции y_i, $i = 0, \ldots, n-1$, вычислите коэффициенты многочлена — наилучшего среднеквадратичного приближения заданной степени k для функции $y = f(x)$.

Идеи реализации. Искомый многочлен $P_k(x) = a_k x^k + \cdots + a_0$ доставляет минимум выражению

$$\Phi(P_k) = \sum_{i=0}^{n-1} \bigl(y_i - P_k(x_i)\bigr)^2.$$

Приравнивая к нулю производные,

$$\frac{\partial \Phi(P_k)}{\partial a_s} = 0, \quad s = 0, \ldots, k,$$

получаем систему линейных уравнений для определения неизвестных коэффициентов $a_0, \ldots, a_k$. Например, для приближения линейной функцией $y = ax + b$ получается система

$$\begin{cases} \sum\limits_{i=0}^{n-1} (y_i - ax_i - b)x_i = 0, \\ \sum\limits_{i=0}^{n-1} (y_i - ax_i - b) = 0, \end{cases}$$

или, после проведения простейших преобразований,

$$\begin{cases} \alpha_{11}a + \alpha_{12}b = \beta_1, \quad \alpha_{11} = \sum\limits_{i=0}^{n-1} x_i^2, \ \alpha_{12} = \sum\limits_{i=0}^{n-1} x_i, \ \beta_1 = \sum\limits_{i=0}^{n-1} x_i y_i; \\ \alpha_{21}a + \alpha_{22}b = \beta_2, \quad \alpha_{21} = \sum\limits_{i=0}^{n-1} x_i, \ \alpha_{22} = n, \ \beta_2 = \sum\limits_{i=0}^{n-1} y_i. \end{cases}$$

Теперь значения a и b можно легко определить, выписав решение этой системы в явном виде.

Задача 2-3-7. Выпишите систему уравнений для определения коэффициентов многочлена наилучшего приближения $P_{n-1}^{(0)}(x)$ для функции $f(x)$ в пространстве $L_2(0,1)$. Формально найдите ее решение одним из методов раздела 2.6. Оцените качество приближения (в том числе графически) на примере функции $f(x) = x^{n-1} + 1$, $x \in [0,1]$, при $n = 5, 10, 30, 50$.

Решение. Наилучшее приближение ищется в виде $P_{n-1}^{(0)}(x) = \sum\limits_{j=0}^{n-1} a_j x^j$ с неизвестными коэффициентами a_j, которые определяются из условия минимума функционала

$$\int_0^1 \left(f(x) - \sum_{j=0}^{n-1} a_j x^j \right)^2 dx.$$

Дифференцируя функционал по a_i и приравнивания производные к нулю, получаем уравнения

$$\int_0^1 \left(f(x) - \sum_{j=0}^{n-1} a_j x^j \right) x^i \, dx = 0, \quad i = 0, 1, \ldots, n-1,$$

или

$$\sum_{j=0}^{n-1} \frac{a_j}{i+j+1} = \int_0^1 f(x)x^i\,dx, \quad i = 0, 1, \ldots, n-1.$$

Приходим к системе уравнений с *матрицей Гильберта*:

$$H_{ij} = \frac{1}{i+j+1}, \quad 0 \leqslant i, j \leqslant n-1, \quad \|H^{-1}\|_\infty \sim \frac{1}{\sqrt{n}}(1+\sqrt{2})^{4n}.$$

Оценка показывает, что при увеличении n матрица H быстро стремится к вырожденной, и задача становится численно некорректной.

— Тебя почему не взяли в первый класс?
— Спросили, есть ли самое большое целое число. Я ответил, что есть и равно 2147483647. *— «А что получится, если к нему прибавить единицу?» — «Самое маленькое отрицательное, равное* −2147483648».
— Эх ты. А вдруг это был unsigned int?

2.4. Численное интегрирование

Для приближенного вычисления определенных интегралов обычно применяется метод квадратурных формул:

$$I^{[a,b]}(f) = \int_a^b f(x)\,dx \approx S_n^{[a,b]}(f) = \sum_{i=1}^{n} c_i f(x_i).$$

При этом узлы $\{x_i\}$ и коэффициенты $\{c_i\}$ выбираются специальным образом так, что для погрешности $R_n(f) = |I^{[a,b]}(f) - S_n^{[a,b]}(f)|$ верна оценка вида $R_n(f) \leqslant C(b-a)^k$. К наиболее известным квадратурам относятся следующие:

формула прямоугольников по левой точке

$$(b-a)f(a), \quad R_n(f) \| f' \| \frac{(b-a)^2}{2},$$

формула прямоугольников по центральной точке

$$(b-a)f\left(\frac{a+b}{2}\right), \quad R_n(f) \leqslant \|f''\| \frac{(b-a)^3}{24},$$

формула трапеций

$$\frac{b-a}{2}\big(f(a)+f(b)\big), \quad R_n(f) \leqslant \|f''\| \frac{(b-a)^3}{12},$$

формула Симпсона

$$\frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right), \quad R_n(f) \leqslant \|f^{(4)}\| \frac{(b-a)^5}{2880},$$

формула Гаусса по трем узлам

$$\frac{b-a}{18}\bigl(5f(x_-)+8f(x_0)+5f(x_+)\bigr),\quad R_n(f)\leqslant\|f^{(6)}\|\frac{(b-a)^7}{2016000},$$

$$\text{где}\quad x_0=\frac{a+b}{2},\quad x_\pm=\frac{a+b}{2}\pm\frac{b-a}{2}\sqrt{\frac{3}{5}}.$$

В данном случае $\|g\|=\max\limits_{x\in[a,b]}|g(x)|$.

Задача 2-4-1. Реализуйте каждый из описанных выше методов интегрирования в виде функции с прототипом

```
double Integral(double a, double b,
                double (*f)(double));
```

где f — указатель на подынтегральную функцию. Проверьте выполнение указанных оценок погрешности для явно интегрируемых f, взяв, например, $f(x)=x^n$, $n=0,1,\ldots,5,10$, $f(x)=e^x$, $f(x)=\sin(x)$.

Если значение погрешности $R_n(f)$ для конкретной задачи получается недопустимо большим, то обычно используют следующий прием. Область интегрирования $[a,b]$ разбивается на N подотрезков $[a,b]=\bigcup\limits_{k=1}^{N}[a_k,b_k]$, и на каждом подотрезке $[a_k,b_k]$ значение интеграла $I^{[a_k,b_k]}(f)$ заменяется на значение квадратуры $S_n^{[a_k,b_k]}(f)$. В результате для вычисления интеграла

$$I^{[a,b]}(f)=\sum_{k=1}^{N}I^{[a_k,b_k]}(f)$$

получается так называемая составная квадратурная формула

$$S_{N,n}^{[a,b]}(f)=\sum_{k=1}^{N}S_n^{[a_k,b_k]}(f)$$

с оценкой погрешности

$$R_{N,n}^{[a,b]}(f)\leqslant\sum_{k=1}^{N}R_n^{[a_k,b_k]}(f).$$

Задача 2-4-2. Для каждого из описанных выше методов интегрирования реализуйте составную квадратуру в виде функции

```
double Integral(double a, double b,
                double (*f)(double), int N),
```

где N — число разбиений отрезка интегрирования [a, b] на равные подотрезки. Рассмотрите несколько примеров, интегрируемых в элементарных функциях, и сравните приближенные значения интегралов, полученные при разных N, с точными значениями. Например:

$$\int_0^{100\pi} \cos 1000x\,dx = 0, \qquad \int_0^{100} \exp^{-1000x}\,dx \approx 10^{-3}, \qquad \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} = \pi.$$

Задача 2-4-3. Аналитически и численно проверьте, что

$$\int_{-\pi}^{\pi} \sin mx \cos nx\,dx = 0,$$

$$\int_{-\pi}^{\pi} \sin mx \sin nx\,dx = \begin{cases} 0, & m \neq n, \\ \pi, & m = n \neq 0, \\ 0, & m = n = 0, \end{cases}$$

$$\int_{-\pi}^{\pi} \cos mx \cos nx\,dx = \begin{cases} 0, & m \neq n, \\ \pi, & m = n \neq 0, \\ 2\pi, & m = n = 0. \end{cases}$$

Здесь m, n — неотрицательные целые числа. Такое свойство тригонометрических функций означает ортогональность семейства $\{1, \sin(mx), \cos(mx),\ m = 1, 2, \ldots\}$ на отрезке $[-\pi, \pi]$ относительно скалярного произведения $(u, v) = \int_{-\pi}^{\pi} u(x)v(x)\,dx$.

Задача 2-4-4. Для составной квадратурной формулы Гаусса по трем узлам численно найдите константы C и p в оценке погрешности

$$R(N) = \left|I^{[-\pi,\pi]}(f) - S_{N,3}^{[-\pi,\pi]}(f)\right| \sim C/N^p$$

при $N \gg 1$ и $f(x) = \sin^2 mx$, $m = 1, 10^3$.

Указание. Для достаточно больших значений N исследуйте график

$$F(\ln N) = \ln R^{-1}(N) \sim \ln C^{-1} + p \ln N.$$

Задача 2-4-5. Реализуйте функцию с прототипом

```
double IntegralEps(double a, double b,
                   double (*f)(double), double eps);
```

для вычисления значения интеграла по выбранной составной квадратурной формуле с переменным шагом и локальным ε-контролем точности.

Идеи реализации. Пусть мы имеем шаг h и найденное значение квадратуры $S^{[a,\tilde{a}]}(f)$ на отрезке $[a,\tilde{a}]$, $a \leqslant \tilde{a} < b$. Вычислим

$$S_1 = S^{[\tilde{a},\tilde{a}+h]}(f), \quad S_2 = S^{[\tilde{a},\tilde{a}+h/2]}(f) + S^{[\tilde{a}+h/2,\tilde{a}+h]}(f).$$

Если $|S_1 - S_2| \leqslant \varepsilon h$, то считаем, что требуемая точность на шаге достигнута, и полагаем

$$I^{[a,\tilde{a}+h]} \approx S^{[a,\tilde{a}+h]} = S^{[a,\tilde{a}]}(f) + S_2.$$

Если $|S_1 - S_2| > \varepsilon h$, то уменьшим шаг h в два раза, т. е. положим $h = h/2$, и повторим вычисление S_1 и S_2, начав с точки $\tilde{a}$. Будем уменьшать шаг до тех пор, пока не добьемся выполнения неравенства $|S_1 - S_2| \leqslant \varepsilon h$, что позволит найти $S^{[a,\tilde{a}+h]}$.

Если при этом будет верна оценка $\delta h < |S_1 - S_2|$, где $\delta \ll \varepsilon$, то следующий шаг интегрирования от точки $\tilde{a} + h$ начнем выполнять с тем же значением h. Если же верна оценка $|S_1 - S_2| \leqslant \delta h$, то текущий шаг h обеспечивает «слишком высокую» точность, и, с целью сокращения вычислительных затрат, для вычисления интеграла по следующему частичному отрезку этот шаг можно увеличить в два раза, т. е. положить $h = 2h$. На практике величину δ нужно выбирать в пределах от $0{,}1\varepsilon$ до $0{,}01\varepsilon$ и нужно установить минимальное и максимальное значение для величины шага.

Детский садик на прогулке. От одного ребенка вся группа бегает.
— Что, в салочки играют?
— Да нет, он у нас сегодня Помножитель на Ноль, вот все и боятся, что их обнулят.
— Почему же вон та девочка не убегает?
— А ей все равно, она у нас Not a Number.

2.5. Ряд Фурье

При изучении физических процессов, связанных, например, с анализом звуковых и тепловых процессов, эффективные

алгоритмы численного анализа удается построить на основе разложения искомой функции f(x) в ряд Фурье:

$$f(x) \equiv S_\infty(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx).$$

В задачах необходимо реализовать функцию

```
void f2c(double * a, double * b,
         double (*)(double), int N, double eps),
```

определяющую с точностью ε первые N коэффициентов ряда Фурье, и функцию

```
double c2f(double x, double *a, double *b, int N),
```

вычисляющую при заданных массивах коэффициентов a, b, значение соответствующей суммы Фурье в заданной точке x. Используя эти функции, нужно проверить (например, построив графики) сходимость полученного ряда Фурье к исходной функции при увеличении N.

Задача 2-5-1. Пусть на отрезке $[-\pi, \pi]$ задана гладкая функция f(x). Выпишите систему уравнений для определения коэффициентов тригонометрического многочлена

$$S_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N} a_n \cos(nx) + \sum_{n=1}^{N} b_n \sin(nx),$$

наилучшего в смысле приближения к f(x) в пространстве $L_2(-\pi, \pi)$ (см. задачу 2-3-7). Для функций $(\pi^2 - x^2)^2 + 1$, x, $\pi - x$, $|x|$ исследуйте графически близость f(x) и соответствующего ряда при $N = 10, 100, 1000$, численно определив значения коэффициентов с некоторой локальной точностью ε (см. задачу 2-4-5).

Указание. Система уравнений для определения оптимальных коэффициентов $\{a_n, b_n\}$ может быть найдена из условия минимума соответствующего квадратичного функционала (см. задачу 2-3-7). В данном случае (с учетом ортогональности тригонометрического базиса на отрезке $[-\pi, \pi]$, см. задачу 2-4-3) матрица имеет диагональный вид, откуда находим: $a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)\,dx$,

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(\pi n x)\,dx, \quad b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(\pi n x)\,dx.$$

Задача 2-5-2. Пусть на отрезке $[0, \pi]$ задана гладкая функция $f(x) = x(\pi - x)2^x$, равная нулю на границе. Выпишите систему уравнений для определения коэффициентов тригонометрического многочлена

$$S_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N} a_n \cos(nx) + \sum_{n=1}^{N} b_n \sin(nx),$$

оптимального для $f(x)$ в смысле наилучшего приближения в пространстве $L_2(0, \pi)$. Численно исследуйте сходимость ряда при увеличении N для различных функций $f(x)$.

Указание. По формуле $f(-x) = -f(x)$, т. е. кососимметрично, продолжите $f(x)$ на отрезок $[-\pi, \pi]$ и воспользуйтесь решением задачи 2-5-1. С учетом кососимметрии продолженной функции получите, что

$$a_n = 0, \quad b_n = \frac{2}{\pi} \int_0^{\pi} f(x) \sin(nx)\, dx.$$

Задача 2-5-3. Пусть на отрезке $[0, \pi]$ задана гладкая функция $f(x) = e^{x^2(\pi - x)^2}$, $f'(0) = f'(\pi) = 0$. Найдите систему уравнений для определения коэффициентов тригонометрического многочлена

$$S_N(x) = \frac{a_0}{2} + \sum_{n=1}^{N} a_n \cos(nx) + \sum_{n=1}^{N} b_n \sin(nx),$$

оптимального в смысле наилучшего приближения в пространстве $L_2(0, \pi)$ (см. задачу 2-3-7). Исследуйте точность полученного приближения для различных N.

Указание. Симметрично продолжите $f(x)$ на отрезок $[-1, 1]$ и воспользуйтесь решением задачи 2-5-1. С учетом симметрии получите

$$b_n = 0, \quad a_n = \frac{2}{\pi} \int_0^{\pi} f(x) \cos(nx)\, dx.$$

Как изменится точность приближения, если $f(x)$ продолжить кососимметрично?

Для прикладных задач особое значение имеет дискретное преобразование Фурье, основанное на ортогональности так называемых дискретных тригонометрических функций — это свойство позволяет выбрать их в качестве базиса в соответствующем многомерном пространстве.

Задача 2-5-4. Пусть в пространстве $\mathbf{R}^{N+1}$ задан набор векторов $\{\psi^{(m)},\ m = 1,\ldots,N-1\}$ вида

$$\psi^{(m)} = \left(\psi_0^{(m)},\ldots,\psi_k^{(m)},\ldots,\psi_N^{(m)}\right)^T,$$

где координаты m-го вектора вычисляются по формуле

$$\psi_k^{(m)} = \sin(mkh),\quad k = 0,1,\ldots,N,\quad m = 1,\ldots,N-1,\quad h = \frac{\pi}{N}.$$

При этом $\psi_0^{(m)} = \psi_N^{(m)} = 0$ для всех $m = 1,\ldots,N-1$. Аналитически и численно проверьте, что система функций $\{\psi^{(m)}\}$ ортогональна относительно скалярного произведения $(u,v)_0 = \sum_{k=1}^{N-1} u_k v_k h$, т. е. $\left(\psi^{(m)},\psi^{(n)}\right)_0 = 0$, $m \neq n$, $\left(\psi^{(m)},\psi^{(m)}\right)_0 = \pi/2$. Проведите аналогию с задачей 2-4-3.

Задача 2-5-5. Разложите вектор $f = [f_0,f_1,\ldots,f_{N-1},f_N]^T$ с нулевыми крайними значениями $f_0 = f_N = 0$ по базису $\{\psi^{(m)}\}$, т. е. найдите коэффициенты c_m в представлении $f = \sum_{m=1}^{N-1} c_m\psi^{(m)}$.

Указание. Реализуйте функции

```
int f2c(double *c, const double *f, const int N),
int c2f(double *f, const double *c, const int N).
```

Первая функция получает массив f[i], $i = 0,1,\ldots,N$, f[0] = f[N] = 0, и вычисляет с учетом ортогональности векторов $\{\psi^{(m)}\}$ массив соответствующих коэффициентов c[m], $m = 1,\ldots,N-1$, по формуле $c_m = \left(u,\psi^{(m)}\right)_0/(\pi/2)$. Вторая функция по набору коэффициентов c[m], $m = 1,\ldots,N-1$, восстанавливает массив значений соответствующей функции f[k], $k = 0,1,\ldots,N$, f[0] = f[N] = 0. Например,

```
int c2f(double *f, const double *c, const int N){
  int k, m;
  double xk, h = M_PI / (double)N;
  for (k = 0; k <= N; k++){
    f[k]=0.;
    xk = k * h;
    for (m = 1; m <= N - 1; m++){
      f[k] += c[m] * sin(m * xk);
    }
  }
  return 0;
}
```

Проведите аналогию с задачей 2-5-2 и протестируйте работу функций на примере $u_k = x_k(\pi - x_k)2^{x_k}$, $x_k = kh$, $k = 0, \ldots, N$, $h = \pi/N$.

Задача 2-5-6. Пусть в пространстве $\mathbf{R}^{N+1}$ задан набор векторов $\{\varphi^{(m)},\ m = 0, \ldots, N\}$ вида $\varphi^{(m)} = \left(\varphi_0^{(m)}, \ldots, \varphi_k^{(m)}, \ldots, \varphi_N^{(m)}\right)^T$, где координаты m-го вектора вычисляются по формуле

$$\varphi_k^{(m)} = \cos(mkh), \quad k = 0, 1, \ldots, N, \quad m = 0, \ldots, N, \quad h = \frac{\pi}{N}.$$

Аналитически и численно проверьте, что система векторов $\varphi^{(m)}$ ортогональна относительно скалярного произведения $(u, v)_1 = \sum_{k=1}^{N-1} u_k v_k h + u_0 v_0 \frac{h}{2} + u_N v_N \frac{h}{2}$, т. е. $\left(\varphi^{(m)}, \varphi^{(n)}\right)_1 = 0$, $m \neq n$, $\left(\varphi^{(m)}, \varphi^{(m)}\right)_1 = \pi$ для $m = 0, N$, $\left(\varphi^{(m)}, \varphi^{(m)}\right)_1 = \pi/2$ для $m \neq 0, N$. Проведите аналогию с задачей 2-4-3.

Задача 2-5-7. Представьте произвольный вектор $u = [u_0, u_1, \ldots, u_{N-1}, u_N]^T$ в виде $u = \sum_{m=0}^{N} c_m \varphi^{(m)}$, т. е. найдите коэффициенты c_m.

Указание. С учетом ортогональности векторов $\{\varphi^{(m)}\}$ имеем $c_m = \left(u, \varphi^{(m)}\right)_1 / \left(\varphi^{(m)}, \varphi^{(m)}\right)_1$. Реализуйте функции

```
int f2c(double *c, const double *f, const int N),
int c2f(double *f, const double *c, const int N)
```

(см. задачу 2-5-5) и протестируйте их работу на примере $u_k = e^{x_k^2(\pi - x_k)^2}$, $x_k = kh$, $k = 0, \ldots, N$, $h = \pi/N$. Проведите аналогию с задачей 2-5-3.

— Вычислим следующий предел: $\lim\limits_{x \to 8} \dfrac{1}{x - 8} = \infty$. *Все понятно?*
— Ну, да...
— Отлично! Найдите предел $\lim\limits_{x \to 5} \dfrac{1}{x - 5}$.
— Может, так: ∞ *?*
— Нет, подумайте.
— Ну конечно же: ∞ *!*

2.6. Матрицы, линейная алгебра

Работа с матрицами имеет свою специфику в каждом алгоритмическом языке. В первую очередь эта специфика связана с необ-

ходимостью передавать матрицу в качестве параметра в различные процедуры и функции. В языке C при работе с динамически создаваемыми матрицами обычно используют два подхода.

В первом случае элементы матрицы $A = (a_{i,j})$ из m строк и n столбцов (т. е. с $i = 0, \ldots, m-1$, $j = 0, \ldots, n-1$) «укладываются» по строкам в одномерный массив mas длины $m \cdot n$. Если обозначить элементы этого массива через mas[k], то имеют место соответствия $k = n \cdot i + j$, $i = [k/n]$, $j = k \bmod n$. Параметрами, определяющими местоположение элементов матрицы, здесь являются указатель на начало массива mas и длина строки n. Количество строк m необходимо только для контроля границы массива.

Во втором случае для каждой строки матрицы выделяется отдельный массив длины n и указатели на начала этих массивов-строк собираются в дополнительном массиве a[m] так, что a[i] является указателем на i-ю строку матрицы a. Тогда a[i][j] соответствует $a_{i,j}$. Местоположение элементов матрицы определяет указатель на начало массива a, величины m, n необходимы для контроля границ. Отметим, что с точки зрения эффективности вычислений этот метод существенно хуже предыдущего, поскольку требует двух обращений к памяти для извлечения элемента матрицы, а также использует дополнительный массив для хранения указателей на массивы строк. Более того, строки матрицы разбросаны по памяти, что затрудняет кэширование и приводит к значительному увеличению времени доступа к элементам.

Задача 2-6-1. Реализуйте функцию выделения памяти для матрицы размера $m \times n$ и функцию освобождения этой памяти при хранении матрицы первым и вторым способами. Функции должны иметь прототипы

```
double *CreateMatrix1(int m, int n);
double *FreeMatrix1(double *mas);
double **CreateMatrix2(int m, int n);
double *FreeMatrix2(double **a, int m);
```

При этом функция `CreateMatrix2` возвращает указатель на массив указателей на строки матрицы, либо NULL в случае отказа в выделении памяти. Заметим, что отказ может возникнуть уже после выделения памяти для нескольких строк матрицы. В этом случае функция `CreateMatrix2` должна корректно обработать эту ситуацию и освободить уже захваченную для строк память.

Решение.

```
double *CreateMatrix1(unsigned int m,
                                  unsigned int n) {
  unsigned int i;
  double *mas;
  mas = (double *)malloc(n * m * sizeof(double));
  if (mas == NULL) return NULL;
  for (i = 0; i < m * n; i++) {
    mas[i] = 0.;
  }
  return mas;
}
double *FreeMatrix1(double *a) {
  if (a != NULL) free(a);
  return NULL;
}
double **CreateMatrix2(unsigned int m,
                                  unsigned int n) {
  unsigned int i, j;
  double **a;
  a = (double **)malloc(m * sizeof(double *));
  if (a == NULL) return NULL;
  for (i = 0; i < m; i++) {
    a[i] = (double *)malloc(n * sizeof(double));
    if (a[i] == NULL) {
      FreeMatrix2(a, i);
      return NULL;
    }
  }
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      a[i][j] = 0.;
    }
  }
  return a;
}
double **FreeMatrix2(double **a, unsigned int m) {
  unsigned int i;
  if (a == NULL) return NULL;
```

```
  for (i = 0; i < m; i++) {
    if (a[i] != NULL) free(a[i]);
  }
  free(a);
  return NULL;
}
```

Замечание. При желании эффективность первого подхода можно объединить с наглядностью второго следующим способом: выделим один блок памяти для хранения всех строк матрицы, один блок для указателей на эти строки и установим между ними соответствие.

```
double **CreateMatrix3(unsigned int m,
                                  unsigned int n) {
  double *mas;
  double **a;
  mas = (double *)malloc(m * n * sizeof(double));
  if (mas == NULL) return NULL;
  a = (double **)malloc(m * sizeof(double *));
  if (a == NULL) {
    free(mas);
    return NULL;
  }
  for (unsigned int i = 0; i < m; i++)
    a[i] = mas + n * i;
  return a;
}
```

В результате ячейка a[i][j] эквивалентна ячейке mas[i * n + j]. С точки зрения эффективности еще лучше обойтись единым блоком памяти для хранения указателей и элементов:

```
double **CreateMatrix4(unsigned int m,
                                  unsigned int n) {
  double **a;
  // выделяем место для хранения
  // указателей на строки и элементы матрицы:
  a = (double **)malloc(m * sizeof(double *) +
                        n * m * sizeof(double));
  if (a == NULL) return NULL;
  //задаем указатель на начальную строку:
```

```
  a[0] = (double *)(a + m);
  for (unsigned int i = 1; i < m; i++)
    a[i] = a[i - 1] + n; //на последующие строки
  return a;
}
```

Задача 2-6-2. Реализуйте функции для выполнения операций суммирования и умножения двух прямоугольных матриц, умножения матрицы на число, умножения матрицы на вектор при хранении матриц первым и вторым способом.

Задача 2-6-3. Для описанных выше подходов постройте функции для заполнения квадратной матрицы Гильберта H_n с элементами вида $(H_n)_{i,j} = 1/(1+i+j)$, $i,j = 0,\ldots,n-1$.

Решение.

```
void Hilbert1(double *a, int n) {
#define a_(i, j) a[(i) * n + (j)]
  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a_(i, j) = 1.0 / (double)(i + j + 1);
#undef a_
}
void Hilbert2(double **a, int n) {
  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[i][j] = 1.0 / (double)(i + j + 1);
}
```

Формально при всех $n > 0$ определитель матрицы H_n отличен от нуля. Однако можно показать, что для обратной матрицы верна оценка

$$\|H_n^{-1}\|_\infty \sim \frac{1}{(2\pi)^{3/2}2^{7/4}\sqrt{n}}\left(\sqrt{2}+1\right)^{4n},$$

т. е. H_n стремится к вырожденной при увеличении n. Именно поэтому матрица Гильберта часто используется для тестирования надежности численных алгоритмов линейной алгебры.

Методом Гаусса обычно называют процедуру приведения матрицы A к треугольному виду за счет линейных комбинаций ее строк. Алгоритм имеет кубическую сложность, т. е. реализуется

в виде трех вложенных циклов. На данный момент имеются различные варианты метода, мы схематически изложим идею одного из них — метода с выбором главного (ведущего) элемента по столбцу.

Пусть $A = (a_{i,j})$ — матрица, $\mathbf{a}_i = \{a_{i,0}, a_{i,1}, \ldots, a_{i,n-1}\}$ — i-я строка матрицы. Для $k = 0, 1, \ldots, n-1$ последовательно выполняются следующие действия.

1. Определяется такое s, что $|a_{s,k}| = \max\limits_{k \leqslant i \leqslant n-1} |a_{i,k}|$.
2. Строки $\mathbf{a}_k$ и $\mathbf{a}_s$ меняются местами.
3. Строка $\mathbf{a}_k$ делится на элемент $a_{k,k}$, в результате диагональный элемент становится равным единице.
4. Для каждого $i = k+1, \ldots, n-1$ выполняется модификация i-й строки: $\mathbf{a}_i = \mathbf{a}_i - a_{i,k}\,\mathbf{a}_k$. В результате элементы $a_{i,k}$ становятся равными нулю.

Естественно, на очередном шаге не нужно обрабатывать нулевые элементы матрицы, т. е. $a_{i,j}$ при $j < k$.

Задача 2-6-4. Реализуйте функцию приведения невырожденной квадратной матрицы к верхнетреугольному виду методом Гаусса с выбором главного элемента по столбцу.

Задача 2-6-5. Реализуйте функцию вычисления ранга (т. е. максимального числа линейно независимых строк/столбцов) прямоугольной вещественной матрицы методом Гаусса с выбором главного элемента по подматрице. У функции должны быть следующие параметры: матрица, количество ее строк и количество ее столбцов. Возвращаемое значение — искомый ранг. Проведите тестирование для матриц с известным ответом:

$$A_1 = \begin{pmatrix} 5 & 1 & 1 & 2 \\ 5 & 1 & 3 & 4 \\ 10 & 2 & 4 & 6 \\ 15 & 3 & 5 & 8 \end{pmatrix},\ A_2 = \begin{pmatrix} 1 & 0{,}99 & 0{,}999 \\ 1 & 1 & 0{,}99 \\ 1 & 1 & 1 \end{pmatrix},\ A_3 = H_{15};$$

$$(\operatorname{rank} A_1 = 3),\ (\operatorname{rank} A_2 = 3),\ (\operatorname{rank} A_3 = 15).$$

Идеи реализации. Модифицируем первые два шага метода Гаусса следующим образом.

1. Определим такие s, r, что $|a_{s,r}| = \max\limits_{\substack{k \leqslant i \leqslant n-1 \\ k \leqslant j \leqslant n-1}} |a_{i,j}|$.
2. Поменяем местами строки $\mathbf{a}_k$, $\mathbf{a}_s$; поменяем местами столбцы $\mathbf{a}^k$, $\mathbf{a}^r$.

   Шаги 3, 4 остаются без изменений.

Применив модифицированный метод Гаусса, мы получим в итоге матрицу с нулями ниже главной диагонали. При отсутствии ошибок округления ранг матрицы равен количеству ненулевых элементов на главной диагонали. Однако при практической реализации из-за вычислительной погрешности может оказаться, что элементы диагонали, которые должны были стать нулевыми, имеют некоторое малое значение. Следует ввести порог и считать элементы матрицы ниже его нулями. С другой стороны, «почти вырожденные» матрицы могут иметь точные ненулевые диагональные элементы ниже установленного порога, и для них матриц будет получен неверный результат. Аккуратный выбор порога представляет собой непростую задачу. Для указанных матриц можно попробовать взять порог от 10^{-12} до 10^{-15}.

Задача 2-6-6. Реализуйте функцию вычисления определителя квадратной вещественной матрицы (напомним, что данная задача численно неустойчива). Параметрами функции должны быть матрица и ее размерность. Возвращаемое значение — искомый определитель. Протестируйте работу этой функции на следующих $(n \times n)$-матрицах:

$$A = \begin{pmatrix} 5 & -4 & 1 & & & & \\ -4 & 6 & -4 & 1 & & \text{0} & \\ 1 & -4 & 6 & -4 & 1 & & \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ & & 1 & -4 & 6 & -4 & 1 \\ & \text{0} & & 1 & -4 & 6 & -4 \\ & & & & 1 & -4 & 5 \end{pmatrix} \quad (\det A = (n+1)^2),$$

$$A = \begin{pmatrix} x+y & x & x & x & x \\ x & x+y & x & x & x \\ x & x & x+y & x & x \\ \dots & \dots & \dots & \dots & \dots \\ x & x & x & x+y & x \\ x & x & x & x & x+y \end{pmatrix} \quad (\det A = y^{n-1}(y+nx)).$$

Идеи реализации. Не стоит решать данную задачу, используя явную формулу (метод перестановок) для подсчета определителя. Даже для матриц размера 20×20 потребуется слишком много времени, а полученное в результате число вряд ли будет близким к правильному из-за потери точности. Поэтому приведем исходную матрицу к верхнетреугольному виду, применяя алгоритм метода Гаусса. Перестановка строк матрицы может изменить знак определителя, умножение строки на число

умножает определитель на то же число. При приведении матрицы к треугольному виду следует принимать во внимание множитель изменения определителя и знак, появившиеся в результате каждой перестановки и линейной комбинации. Для полученной треугольной матрицы определитель вычисляется перемножением диагональных элементов, а потом корректируется в соответствии с накопленным множителем.

Еще раз напомним, что задача вычисления определителя действительной матрицы в общем случае является сильно неустойчивой.

Задача 2-6-7. Реализуйте функцию решения системы линейных уравнений $\mathbf{Ax} = \mathbf{b}$ методом Гаусса. Параметрами функции должны быть матрица, ее размерность, вектор правой части системы, вектор решения. Возвращаемое значение для невырожденной матрицы — длина вектора невязки $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$. Для вычисления величины $\|\mathbf{r}\|$ потребуется предварительно сделать копию матрицы и вектора правой части. В случае вырожденной матрицы верните -1. Протестируйте работу функции на системе с матрицей Гильберта (см. задачу 2-6-3) при $\mathbf{x} = (1, \ldots, 1)^{\mathrm{T}}$ для различных n.

Идеи реализации. С компонентами вектора правой части системы проводятся те же преобразования, которые выполняются при приведении матрицы к треугольному виду. После получения эквивалентной системы с треугольной матрицей решение вычисляется «снизу вверх» обратной подстановкой. Для контроля вырожденности матрицы следует использовать пороговое значение для нулевых элементов (см. замечание к задаче 2-6-5).

Задача 2-6-8. Реализуйте функцию вычисления обратной матрицы A^{-1} для данной квадратной вещественной матрицы A. Параметрами функции должны быть исходная матрица, обратная матрица, их размерность. Возвращаемое значение для невырожденной матрицы A — норма матрицы $\mathbf{I} - \mathrm{A}^{-1}\mathrm{A}$, например корень квадратный из суммы квадратов ее элементов; в случае вырожденной матрицы верните -1.

Идеи реализации. В области памяти, отведенной для хранения обратной матрицы, размещается единичная матрица. Исходная матрица A приводится к треугольному виду с единицами на диагонали, затем (аналогичными преобразованиями) к диагональному виду, т. е. к единичной матрице. Все указанные преобразования синхронно проводятся с единичной матрицей. В ре-

зультате она преобразуется в обратную к A. Сравните точность решения системы уравнений методом Гаусса и методом умножения вектора правой части на полученную обратную матрицу.

Задача 2-6-9. Реализуйте метод прогонки — экономичную версию алгоритма Гаусса для трехдиагональных матриц.

Решение. Пусть требуется найти решение системы уравнений $\mathbf{Ay} = \mathbf{f}$, где $\mathbf{y} = (y_0, y_1, \ldots, y_N)^T$ — вектор неизвестных, $\mathbf{f} = (f_0, f_1, \ldots, f_N)^T$ — заданный вектор правых частей, A — квадратная $((N+1) \times (N+1))$-матрица

$$\begin{pmatrix} c_0 & -b_0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 \\ -a_1 & c_1 & -b_1 & 0 & \ldots & 0 & 0 & 0 & 0 \\ 0 & -a_2 & c_2 & -b_2 & \ldots & 0 & 0 & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & 0 & \ldots & -a_{N-2} & c_{N-2} & -b_{N-2} & 0 \\ 0 & 0 & 0 & 0 & \ldots & 0 & -a_{N-1} & c_{N-1} & -b_{N-1} \\ 0 & 0 & 0 & 0 & \ldots & 0 & 0 & -a_N & c_N \end{pmatrix}.$$

Будем искать решение в виде

$$y_k = \alpha_{k+1} y_{k+1} + \beta_{k+1}, \quad k = 0, \ldots, N-1,$$

а значения α_k, β_k и y_k вычислим по коэффициентам исходной системы и правой части. Из первого уравнения следует, что

$$y_0 = \alpha_1 y_1 + \beta_1, \quad \alpha_1 = \frac{b_0}{c_0}, \quad \beta_1 = \frac{f_0}{c_0}.$$

Далее последовательно находим

$$\alpha_{k+1} = \frac{b_k}{c_k - a_k \alpha_k}, \quad \beta_{k+1} = \frac{f_k + a_k \beta_k}{P(c_k - a_k \alpha_k)}, \quad k = 0, \ldots, N-1,$$

а из системы

$$\begin{aligned} y_{N-1} &= \alpha_N y_N + \beta_N, \\ -a_N y_{N-1} &+ c_N y_N = f_N \end{aligned}$$

определим

$$y_N = \frac{f_N + a_N \beta_N}{c_N - a_N \alpha_N}.$$

Это позволяет рекуррентно вычислить остальные компоненты вектора неизвестных:

$$y_k = \alpha_{k+1} y_{k+1} + \beta_{k+1}, \quad k = N-1, N-2, \ldots, 0.$$

Полученные соотношения называют формулами *правой прогонки*.

Для разрешимости и устойчивости алгоритма достаточно выполнения следующих условий: все коэффициенты действительны; c_0, c_N, a_k, c_k, b_k при $k = 1, 2, \ldots, N-1$ отличны от нуля; $|c_k| \geqslant |a_k| + |b_k|$, $k = 1, 2, \ldots, N-1$, $|c_0| \geqslant |b_0|$, $|c_N| \geqslant |a_N|$ и хотя бы одно из неравенств является строгим.

При реализации исходную матрицу A обычно хранят в виде трех отдельных массивов длины $N+1$ и для найденного решения вычисляют норму вектора невязки. Если в процессе расчетов необходимо оптимизировать работу с оперативной памятью и разрешается «затереть» элементы матрицы, то прогоночные коэффициенты α_i, β_i размещают на месте исходных элементов.

Задача 2-6-10. Пусть в пространстве $\mathbf{R}^{N-1}$ задана трехдиагональная матрица

$$A = \begin{pmatrix} \frac{2}{h^2}+p & -\frac{1}{h^2} & 0 & \ldots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+p & -\frac{1}{h^2} & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & 0 & -\frac{1}{h^2} & \frac{2}{h^2}+p & -\frac{1}{h^2} \\ 0 & 0 & \ldots & 0 & -\frac{1}{h^2} & \frac{2}{h^2}+p \end{pmatrix},$$

где $p > 0$, $h = \pi/N$, а также набор чисел

$$\lambda^{(m)} = \frac{4}{h^2}\sin^2\left(\frac{mh}{2}\right) + p, \quad m = 1, \ldots, N-1,$$

и векторов $\psi_k^{(m)} = \sin(mkh)$, $k = 1, \ldots, N-1$. Аналитически и численно проверьте, что векторы $\psi^{(m)}$ являются *собственными векторами* для A, т. е. $A\psi^{(m)} = \lambda^{(m)}\psi^{(m)}$, $m = 1, \ldots, N-1$.

Указание. Выбираем очередное целочисленное m из указанного диапазона, по явным формулам инициализируем действительную переменную lam и действительный массив psi с индексами от 0 до N включительно, для удобства положив psi[0] = psi[N] = 0. Далее вычисляем нормированную длину вектора погрешности $\frac{1}{\lambda_m}A\psi^{(m)} - \psi^{(m)}$. Здесь деление на λ_m обеспечивает переход к относительной погрешности. Приведем реализацию центральной части алгоритма.

```
nev = 0.; d = 2./h/h + p; r = -1./h/h;
k = 1;
tmp = 1./lam*(d*psi[k] + r*psi[k+1]) - psi[k];
```

```
nev += tmp*tmp*h;
for(k = 2; k <= N - 2; k++){
  tmp = 1./lam*(r*psi[k-1] + d*psi[k]
                           + r*psi[k+1]) - psi[k];
  nev += tmp*tmp*h;
}
k = N - 1;
tmp = 1./lam*(d*psi[k] + r*psi[k+1]) - psi[k];
nev += tmp*tmp*h;
nev = sqrt(nev);
```

В данном случае множитель h в скалярном произведении отвечает за усреднение по размерности задачи N, после такой нормировки длина $\psi^{(m)}$ становится равной $\sqrt{\pi/2}$.

Изложенную процедуру повторяем для всех m.

Задача 2-6-11. Пусть в пространстве $\mathbf{R}^{N+1}$ задана трехдиагональная матрица

$$A = \begin{pmatrix} \frac{2}{h^2}+p & -\frac{2}{h^2} & 0 & \dots & 0 & 0 \\ -\frac{1}{h^2} & \frac{2}{h^2}+p & -\frac{1}{h^2} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & -\frac{1}{h^2} & \frac{2}{h^2}+p & -\frac{1}{h^2} \\ 0 & 0 & \dots & 0 & -\frac{2}{h^2} & \frac{2}{h^2}+p \end{pmatrix},$$

где $p \geqslant 0$, $h = \pi/N$, а также набор чисел

$$\lambda^{(m)} = \frac{4}{h^2}\sin^2\left(\frac{mh}{2}\right) + p, \quad m = 0, 1, \dots, N,$$

и векторов $\varphi_k^{(m)} = \cos(mkh)$, $k = 0, \dots, N$. Аналитически и численно проверьте, что векторы $\varphi^{(m)}$ являются *собственными векторами* для A, т. е. $A\varphi^{(m)} = \lambda^{(m)}\varphi^{(m)}$, $m = 0, \dots, N$.

Задача 2-6-12. Для решения системы линейных уравнений

$$-\frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + py_k = f_k, \quad k = 1, \dots, N-1,$$
$$y_0 = y_N = 0, \quad h = \frac{\pi}{N}, \quad p \geqslant 0,$$

реализуйте метод Фурье (т. е. метод разложения по собственным векторам).

Решение. Перепишем задачу в матричном виде относительно вектора $\mathbf{y} = [y_1, \ldots, y_{N-1}]^T$: $A\mathbf{y} = \mathbf{f}$, где $\mathbf{y}, \mathbf{f} \in \mathbf{R}^{N-1}$. Для собственных чисел и собственных векторов данной матрицы известны аналитические формулы (см. задачу 2-6-10). При этом (см. задачу 2-5-4) собственные векторы ортогональны относительно скалярного произведения $(u, v)_0$, т. е. образуют базис в пространстве $\mathbf{R}^{N-1}$. Следовательно, формально существует разложение $\mathbf{y} = \sum\limits_{n=1}^{N-1} c_n \psi^{(n)}$. Подставив соотношение в исходную систему, получим

$$A\left(\sum_{n=1}^{N-1} c_n \psi^{(n)}\right) = \mathbf{f}, \text{ т. е. } \sum_{n=1}^{N-1} c_n \lambda_n \psi^{(n)} = \mathbf{f}.$$

Умножим равенство скалярно на $\psi^{(m)}$, $m = 1, \ldots, N-1$, и с учетом ортогональности базиса найдем:

$$\left(\sum_{n=1}^{N-1} \lambda_n c_n \psi^{(n)}, \psi^{(m)}\right)_0 = \left(\mathbf{f}, \psi^{(m)}\right)_0, \text{ т. е. } c_m = \frac{\left(\mathbf{f}, \psi^{(m)}\right)_0}{\lambda_m \left(\psi^{(m)}, \psi^{(m)}\right)_0}.$$

Таким образом, $c_m = \dfrac{d_m}{\lambda_m}$, где величины $d_m = \dfrac{\left(\mathbf{f}, \psi^{(m)}\right)_0}{\left(\psi^{(m)}, \psi^{(m)}\right)_0}$ являются коэффициентами в разложении вектора $\mathbf{f} = \sum\limits_{m=1}^{N-1} d_m \psi^{(m)}$. Определив набор коэффициентов $\{c_m\}$, далее вычисляем координаты искомого вектора $\mathbf{y}_k = \sum\limits_{m=1}^{N-1} c_m \psi_k^{(m)}$, $k = 0, \ldots, N$. Отметим, что $\left(\psi^{(m)}, \psi^{(m)}\right)_0 = \dfrac{\pi}{2}$ при всех $m = 1, \ldots, N-1$.

Задача 2-6-13. Для решения системы линейных уравнений

$$-\frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + p y_k = f_k, \quad k = 1, \ldots, N-1, \quad p \geqslant 0,$$

$$-\frac{2}{h^2}(y_1 - y_0) + p y_0 = f_0, \quad \frac{2}{h^2}(y_N - y_{N-1}) + p y_N = f_N, \quad h = \frac{\pi}{N},$$

реализуйте метод Фурье.

Указание. Перепишите исходную задачу в матричном виде $A\mathbf{y} = \mathbf{f}$, где $\mathbf{y}, \mathbf{f} \in \mathbf{R}^{N+1}$, относительно вектора $\mathbf{y} = [y_0, \ldots, y_N]^T$. Далее воспользуйтесь решением задачи 2-6-12, принимая во внимание задачи 2-6-11 и 2-5-6. Если $p = 0$, то $\lambda_0 = 0$. Следовательно, при $p = 0$ необходимым и достаточным условием существования искомого решения будет выполнение равенства

$d_0 = 0$, т. е. ортогональность правой части $\mathbf{f}$ младшей собственной функции: $d_0 = \left(\mathbf{f}, \psi^{(0)}\right)_1 = 0$.

Отметим, что для рассмотренных примеров нахождение коэффициентов d_m и восстановление решения y_k можно существенно ускорить за счет арифметических свойств собственных функций при помощи так называемого быстрого преобразования Фурье.

Алгоритмы Гаусса, Фурье и метод прогонки относятся к классу *точных* методов, т. е. при отсутствии ошибок округления позволяют найти точное решение системы за конечное число арифметических действий. Однако высокая вычислительная сложность реализации $O(n^3)$ и жесткие требования к структуре матрицы ограничивают область их применения. Рассматриваемые далее *итерационные* алгоритмы за конечное число арифметических действий обычно позволяют только приблизиться с некоторой точностью к искомому решению. Однако эти алгоритмы могут применяться для решения различных прикладных задач большой размерности. Для построения алгоритмов такого типа преобразуем исходную систему $A\mathbf{x} = \mathbf{b}$ к эквивалентному виду $\frac{\mathbf{x} - \mathbf{x}}{\tau} + A\mathbf{x} = \mathbf{b}$, где τ — действительное число (итерационный параметр). Выберем некоторое начальное приближение $\mathbf{x}^0$, например положим $\mathbf{x}^0 \equiv 0$, и будем определять последующие приближения $\mathbf{x}^1, \mathbf{x}^2, \ldots$ по формуле

$$\frac{\mathbf{x}^{k+1} - \mathbf{x}^k}{\tau} + A\mathbf{x}^k = \mathbf{b},$$

эквивалентной

$$\mathbf{x}^{k+1} = (I - \tau A)\mathbf{x}^k + \tau\mathbf{b}.$$

Такой процесс называется двухслойным итерационным методом с постоянным шагом. Можно доказать, что если взять достаточно малое $\tau > 0$, то для некоторого класса матриц (например, положительно определенных матриц A простой структуры) алгоритм будет сходиться с произвольного начального приближения $\mathbf{x}^0$ со скоростью геометрической прогрессии с показателем $q < 1$. Однако отметим, что при $\tau \to 0$ имеем $q \approx 1 - c\tau$, т. е. сходимость замедляется.

Если $A = A^T > 0$ и для произвольного вектора $\mathbf{y}$ выполняется оценка

$$0 < m \leqslant \frac{(A\mathbf{y}, \mathbf{y})}{(\mathbf{y}, \mathbf{y})} \leqslant M,$$

т. е. $\lambda(A) \in [m, M]$, то в некотором смысле наивысшую скорость сходимости обеспечивает выбор $\tau_0 = \dfrac{2}{m+M}$. В этом случае

$$\|\mathbf{x} - \mathbf{x}^k\|_2 \leqslant q^k \|\mathbf{x} - \mathbf{x}^0\|_2, \quad q = \frac{M-m}{M+m}.$$

Таким образом, данный алгоритм, часто называемый в литературе методом Ричардсона, требует для реализации априорной информации о границах спектра: $m \leqslant \lambda(A) \leqslant M$. Необходимые для проведения расчетов конкретные значения m и M, равные для $A = A^T > 0$ наибольшему и наименьшему собственным числам соответственно, можно оценить по теореме Гершгорина: все собственные значения λ произвольной матрицы A принадлежат объединению кругов

$$O_i = \left\{ |z - a_{ii}| \leqslant \sum_{j \neq i} |a_{ij}| \right\}, \quad i = 0, 1, \ldots, n-1,$$

в комплексной плоскости; если же указанное объединение кругов распадается на несколько связных частей, то каждая такая часть содержит столько собственных значений, сколько кругов ее составляют.

Для действительных матриц из теоремы Гершгорина находим:

$$M \leqslant \max_i \sum_j |a_{ij}|, \quad m \geqslant \min_i \left(a_{ii} - \sum_{j \neq i} |a_{ij}| \right).$$

Отметим, что для матриц $A = A^T > 0$ всегда можно грубо положить $m \approx 0$, т. е. $\tau = 2/M$, хотя формально это дает завышенную оценку $q = 1$.

Задача 2-6-14. Реализуйте метод Ричардсона в виде функции

```
double Richardson(double *x, const double *a,
                  const double *b, double tau, int n,
                  double eps, int mIter),
```

возвращающей найденное приближение $\mathbf{x}$ и норму вектора невязки $\|\mathbf{b} - A\mathbf{x}\|_2$. Проведите тестирование программы на матрицах из задач 2-6-3, 2-6-10, 2-6-11, сравнив теоретическую и реальную скорости сходимости. Для этого в задачах $A\mathbf{x} = \mathbf{b}$ с известным точным ответом $\mathbf{x}^*$ на каждом k-м шаге, $k = 0, 1, \ldots$, итерационного процесса сохраняйте в некотором файле очередную тройку

$$k \qquad \|\mathbf{x}^* - \mathbf{x}^k\| \qquad q^k \|\mathbf{x}^* - \mathbf{x}^0\|.$$

По завершении работы программы постройте в gnuplot графики полученных дискретных функций.

Еще одним классическим двухслойным итерационным алгоритмом является метод наискорейшего градиентного спуска:

$$\frac{\mathbf{x}^{k+1}-\mathbf{x}^k}{\tau_k}+A\mathbf{x}^k=\mathbf{b},\quad \tau_k=\frac{(\mathbf{r}^k,\mathbf{r}^k)}{(A\mathbf{r}^k,\mathbf{r}^k)},\quad k=0,1,\ldots .$$

Если $A=A^T>0$, то метод сходится с произвольного начального приближения с оценкой $\|\mathbf{x}-\mathbf{x}^k\|_A\leqslant q^k\|\mathbf{x}-\mathbf{x}^0\|_A$, где $q=\dfrac{M-m}{M+m}$, $\|\mathbf{x}\|_A=(A\mathbf{x},\mathbf{x})^{1/2}$. В данном случае для реализации алгоритма априорная информация о границах спектра не нужна. По сравнению с методом Ричардсона на каждом шаге дополнительно требуется одно умножение на матрицу A и вычисление двух скалярных произведений.

Задача 2-6-15. Реализуйте метод наискорейшего градиентного спуска в виде функции с прототипом

```
double GradientDescen(double *x, const double *A,
                      const double *b, int n,
                      double eps, int mIter);,
```

возвращающей найденное решение $\mathbf{x}$ и норму $\|\mathbf{r}\|_A=(A\mathbf{r},\mathbf{r})^{1/2}$ вектора невязки $\mathbf{r}=\mathbf{b}-A\mathbf{x}$. Проведите тестирование программы на матрицах из задач 2-6-3, 2-6-10, 2-6-11, сравнив теоретическую и реальную скорость сходимости. Для этого для задач $A\mathbf{x}=\mathbf{b}$ с известным точным ответом $\mathbf{x}$ постройте график дискретной функции $\{(k,\ln(\|\mathbf{x}-\mathbf{x}^k\|_A),\ k=0,1,\ldots\}$ и проверьте гипотезу, что $\ln(\|\mathbf{x}-\mathbf{x}^k\|_A)\sim k\ln q+\ln(\|\mathbf{x}-\mathbf{x}^0\|_A)$.

Если матрица A имеет большой разброс собственных чисел и величина $\xi=m/M$ мала, то для рассмотренных алгоритмов скорость сходимости $q\approx 1-2\xi$ близка к единице. В этом случае эффективные алгоритмы удается построить в виде

$$B\frac{\mathbf{x}^{k+1}-\mathbf{x}^k}{\tau}+A\mathbf{x}^k=\mathbf{b},$$

т. е.

$$B\mathbf{y}^{k+1}=\mathbf{b}-A\mathbf{x}^k,\quad \mathbf{x}^{k+1}=\mathbf{x}^k+\tau\mathbf{y}^{k+1}.$$

При этом матрицу B стараются выбрать максимально близкой к матрице A при условии, что система уравнений $B\mathbf{y}^{k+1}=\mathbf{r}^k$ решается быстро. Пусть $A=D+L+R$, где D — диагональная матрица, L и R — соответственно левая нижняя и правая верхняя треугольные матрицы с нулевыми диагоналями (строго нижняя и

строго верхняя треугольные матрицы). Тогда для расчетов можно применять следующие алгоритмы:

метод Якоби ($B = D$, $\tau = 1$):

$$D(\mathbf{x}^{k+1} - \mathbf{x}^k) + A\mathbf{x}^k = \mathbf{b}, \text{ т. е. } D\mathbf{x}^{k+1} + (L + R)\mathbf{x}^k = \mathbf{b};$$

метод Гаусса—Зейделя ($B = D + L$, $\tau = 1$):

$$(D + L)(\mathbf{x}^{k+1} - \mathbf{x}^k) + A\mathbf{x}^k = \mathbf{b}, \text{ т. е. } (D + L)\mathbf{x}^{k+1} + R\mathbf{x}^k = \mathbf{b}.$$

Пусть матрица A обладает свойством диагонального преобладания, т. е. справедливо

$$\sum_{i \neq j} |a_{ij}| \leqslant q|a_{ii}|, \quad i = 0, 1, \ldots, n-1, \quad q < 1.$$

Тогда методы Якоби и Гаусса—Зейделя сходятся с любого начального приближения и верна оценка $\|\mathbf{x} - \mathbf{x}^k\|_\infty \leqslant q^k \|\mathbf{x} - \mathbf{x}^0\|_\infty$, где $\|\mathbf{z}\|_\infty = \max\limits_i |z_i|$.

Задача 2-6-16. Реализуйте метод Якоби в виде функции с прототипом

```
double Jacobi(double *x, const double *A,
        const double *b, int n, double eps, int mIter);,
```

возвращающей найденное приближенное решение $\mathbf{x}$ и норму $\|\mathbf{r}\|_\infty = \max\limits_i |r_i|$ вектора невязки $\mathbf{r} = \mathbf{b} - A\mathbf{x}$. Сравните теоретическую и практическую скорости сходимости (см. задачи 2-6-14, 2-6-15) в указанной норме.

Задача 2-6-17. Реализуйте метод Гаусса—Зейделя в виде функции с прототипом

```
double GaussSeidel(double *x, const double *A,
        const double *b, int n, double eps, int mIter);,
```

возвращающей найденное приближенное решение $\mathbf{x}$ и норму $\|\mathbf{r}\|_\infty = \max\limits_i |r_i|$ вектора невязки $\mathbf{r} = \mathbf{b} - A\mathbf{x}$. Сравните теоретическую и практическую скорости сходимости (см. задачи 2-6-14, 2-6-15) в указанной норме.

Методом верхней релаксации называется итерационный процесс с матрицей $B = D + \omega L$:

$$(D + \omega L)\frac{\mathbf{x}^{k+1} - \mathbf{x}^k}{\tau} + A\mathbf{x}^k = \mathbf{b}.$$

Здесь ω называется *параметром релаксации*, обычно $\omega = \tau$. Методы Якоби ($\omega = 0$, $\tau = 1$) и Гаусса—Зейделя ($\omega = \tau = 1$) также являются методами релаксации.

Пусть $A = A^T > 0$. Тогда при $0 < \omega = \tau < 2$ метод верхней релаксации (в частности, метод Гаусса—Зейделя) сходится с любого начального приближения.

Задача 2-6-18. Реализуйте метод верхней релаксации в виде функции с прототипом

```
double SOR(double *x, const double *A, const double *b,
           int n, double eps, int mIter);,
```

возвращающей найденное приближение `x` и норму вектора невязки $\|\mathbf{r}\|_\infty$. Сравните теоретическую и практическую скорости сходимости (см. задачи 2-6-14, 2-6-15) в указанной норме.

Дама с малышом пришли оформляться в детский сад. Требуется взвесить малыша, но шкала весов от 30 кг.
Дама: «Не получится».
Воспитательница: «Очень просто — взвешиваем маму с малышом, одну маму и находим разность».
Дама: «Действительно, очень просто. Но только все равно не получится — я не мама, я тетя!»

Мораль: учитесь обобщать алгоритмы!

2.7. Нелинейные уравнения

В данном разделе рассматриваются простейшие методы поиска решения z для одномерных уравнений вида $f(x) = 0$, а также систем уравнений $F(\mathbf{x}) = \mathbf{0}$, где F — вектор функций $(f_1, \ldots, f_m)^T$, зависящих от векторного аргумента $\mathbf{x} = (x^1, \ldots, x^m)^T$.

Задача 2-7-1. Реализуйте функцию для решения уравнения $f(x) = 0$ с точностью ε *методом деления пополам* (методом бисекции) со следующим прототипом:

```
int Bisection(double *x, double a, double b,
              double (*f)(double), double eps);
```

Здесь `x` — указатель на полученный корень; `a`, `b` — границы отрезка, содержащего корень; `f` — указатель на функцию, задающую уравнение; `eps` — точность. Возвращаемое значение 0, если точность достигнута, -1, если уравнение решить не удалось.

Идеи реализации. Метод деления пополам позволяет найти корень z непрерывной функции f(x) на отрезке [a, b] при условии, что f(a)f(b) ⩽ 0. В этом случае на каждом шаге метода вычисляется знак f(c) в точке c = (a + b)/2, и если f(a)f(c) ⩽ 0, то мы переходим к отрезку [a, c], иначе — к отрезку [c, b]. Процедура повторяется до тех пор, пока не получим отрезок длины меньше ε, на концах которого значения функции имеют разные знаки. Метод бисекции строит систему стягивающихся отрезков, содержащих корень. В непрерывном случае подобная система отрезков имеет пределом единственную точку. Однако при программной реализации необходимо помнить, что расстояние между представимыми в ЭВМ числами фиксировано, т. е. не может стать сколь угодно малым. Это означает, что если при делении отрезка [a, b] его границы стали двумя соседними представимыми числами, то последующая операция вычисления середины c = (a + b)/2 уже не даст нового числа, так как результат округлится либо в a, либо в b и отрезок не изменится. Это может привести к зацикливанию алгоритма, если выбранная точность поиска корня меньше расстояния между соседними представимыми числами в окрестности этого корня. Корректная программа должна аккуратно отслеживать и разрешать эту ситуацию.

Отметим, что для многих задач точку $\overline{x}$ можно считать приближением с точностью ε к корню уравнения f(x) = 0, если выполняется неравенство $|f(\overline{x})/f'(\overline{x})| < \varepsilon$.

Задача 2-7-2. Реализуйте функцию для решения уравнения f(x) = 0 с точностью ε *методом секущих* со следующим прототипом:

```
int Secant (double *x1, double *x2,
            double (*f)(double), double eps);
```

Здесь `x1` — указатель на первое начальное приближение; `x2` — указатель на второе начальное приближение и получающийся в результате корень; `f` — указатель на функцию, задающую уравнение; `eps` — точность. Возвращаемое значение 0, если точность достигнута, −1, если уравнение решить не удалось.

Идеи реализации. Выберем два различных начальных приближения x_0, x_1 и заменим исходную задачу f(x) = 0 на линейный аналог

$$f(x_0)\frac{x - x_1}{x_0 - x_1} + f(x_1)\frac{x - x_0}{x_1 - x_0} = 0,$$

откуда найдем x_2. Таким образом, вычисления ведутся по формуле

$$x_{n+1} = x_n - f(x_n)\,\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \quad \text{при заданных } x_0, x_1.$$

Задача 2-7-3. Реализуйте функцию для решения уравнения $f(x) = 0$ с точностью ε *методом Ньютона* (Ньютона—Рапсона) со следующим прототипом:

```
int NewtonRaphson (double *x, double (*f)(double),
                   double (*df)(double), double eps);
```

Здесь `x` — указатель на начальное приближение x_0 и получающийся в результате корень; `f` — указатель на функцию, задающую уравнение; `df` — указатель на функцию, вычисляющую производную функции `f`; `eps` — точность. Возвращаемое значение 0, если точность достигнута, -1, если уравнение решить не удалось.

Идеи реализации. Возьмем некоторое начальное приближение $x_0 \approx z$ и заменим исходную задачу $f(x) = 0$ на приближенный линейный аналог

$$f'(x_0)(x - x_0) + f(x_0) = 0,$$

откуда найдем x_1, и т. д. Таким образом, метод Ньютона (его еще называют методом касательных) состоит в последовательном вычислении приближений $x_1, x_2, \ldots$ к корню z по формуле

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{при заданном } x_0.$$

В качестве предварительного критерия окончания вычислений можно взять стандартное условие $|f(x_n)/f'(x_n)| < \varepsilon$, что в данном алгоритме эквивалентно $|x_n - x_{n+1}| < \varepsilon$. Так как указанное неравенство только условно гарантирует требуемую точность ε, в случае корней нечетной кратности можно сделать дополнительную проверку знаков функции f на краях отрезка $[x_{n+1} - \varepsilon, x_{n+1} + \varepsilon]$.

Если начальное приближение выбрано удачно, то метод Ньютона сходится исключительно быстро. Поэтому если точность не достигается за значительное число итераций (например, 100), то вычисления разумно прекратить и вернуть значение -1.

На задачах с известным решением численно проверьте, что если z — корень кратности единица (так называемый простой

корень), то метод Ньютона имеет асимптотически квадратичную скорость сходимости, т. е. в окрестности корня верна оценка

$$|z - x_{n+1}| \leqslant c|z - x_n|^2.$$

Если же кратность корня $p > 1$, то метод сходится со скоростью геометрической прогрессии:

$$|z - x_{n+1}| \approx \frac{p-1}{p}|z - x_n|.$$

В качестве пробных функций $f(x)$ можно, например, взять $x^2 - 17$, $\sin x - 1$, $x^3 \cos^2 x$.

Замечание. Метод секущих является аналогом метода Ньютона, так как производную $f'(x_n)$ можно приближенно вычислить по формуле $(f(x_n) - f(x_{n-1}))/(x_n - x_{n-1})$. Это упрощает расчеты, так как не нужно явно искать $f'(x)$. Однако для метода секущих имеет место только сверхлинейная скорость сходимости $|z - x_{n+1}| \leqslant c|z - x_n|^m$, $m = (1 + \sqrt{5})/2 \approx 1{,}618$.

Задача 2-7-4. Реализуйте модифицированный метод Ньютона, заменив точное вычисление значения производной на приближенный аналог:

$$f'(x_n) = \frac{f(x_n + h) - f(x_n - h)}{2h} + O(h^2).$$

Численно проверьте, как влияет выбор величины $h = 10^{-k}$, $k = 0, 1, \ldots, 15$, на скорость сходимости метода.

Проведите тестирование методов деления пополам, Ньютона, секущих на решении уравнений

$$\sin x - a = 0, \qquad e^x - a = 0, \qquad \log_2 x - a = 0, \qquad x^3 + ax = 0$$

при различных значениях a. Сравните число итераций этих методов при одном и том же значении точности.

Задача 2-7-5. Численно и аналитически изучите зависимость сходимости метода Ньютона от начального приближения при решении уравнения $x^3 - x = 0$.

Указание. Расчетная формула имеет вид: $x_{n+1} = 2x_n^3/(3x_n^2 - 1)$. Если $x_0 = \pm 1/\sqrt{5}$, то $x_1 = \mp 1/\sqrt{5}$ и далее итерационный процесс «зацикливается». Область сходимости к корню $x = 0$ имеет вид $(-1/\sqrt{5}, 1/\sqrt{5})$. Метод не определен в точках $x^{\pm}_0 = \pm 1/\sqrt{3}$, а также в их прообразах $x^{\pm}_{-n}$, т. е. в тех точках, откуда итерационный алгоритм «попадет» в точки $x^{\pm}_0$ через n шагов. Можно показать, что $-1/\sqrt{3} = x^-_0 < x^+_{-1} < x^-_{-2} <$

$< \ldots < -1/\sqrt{5}$ и $1/\sqrt{5} < \ldots < x^+_{-2} < x^-_{-1} < x^+_0 = 1/\sqrt{3}$, а области сходимости к корням $x = \pm 1$ состоят из объединения бесконечного числа указанных перемежающихся открытых интервалов. Таким образом, в сколь угодной малой окрестности точек $\pm 1/\sqrt{5}$ можно найти начальные условия для вычисления каждого из корней уравнения. Численно проверьте данный результат.

Задача 2-7-6. Реализуйте функцию для решения системы уравнений

$$\begin{cases} f_1(x^1, x^2, \ldots, x^m) = 0, \\ f_2(x^1, x^2, \ldots, x^m) = 0, \\ \quad \ldots \\ f_m(x^1, x^2, \ldots, x^m) = 0, \end{cases} \quad \Leftrightarrow \quad F(\mathbf{x}) = 0,\ \mathbf{x} = (x^1, \ldots, x^m)^{\mathrm{T}},$$

с точностью ε методом Ньютона со следующим прототипом:

```
int root(double *x,
         void (*F)(double *, const double *, int),
         void (*dF)(double *, const double *, int),
         int m, double eps);
```

Здесь **x** — указатель на начальное приближение и полученный корень; **F** — указатель на функцию, задающую систему уравнений; **dF** — указатель на функцию, вычисляющую матрицу частных производных функции **F**; **eps** — точность. Возвращаемое значение 0, если точность достигнута, -1, если систему решить не удалось.

Идеи реализации. Получим расчетные формулы метода Ньютона на примере двумерной системы, выписав первые слагаемые ряда Тейлора в точке $\mathbf{x} = (x^1, x^2)^{\mathrm{T}}$ из окрестности текущего приближения $\mathbf{x}_n = (x^1_n, x^2_n)^{\mathrm{T}}$:

$$\begin{cases} 0 = f_1(x^1, x^2) \approx f_1(x^1_n, x^2_n) + \dfrac{\partial f_1(\mathbf{x}_n)}{\partial x^1}(x^1 - x^1_n) + \dfrac{\partial f_1(\mathbf{x}_n)}{\partial x^2}(x^2 - x^2_n), \\ 0 = f_2(x^1, x^2) \approx f_2(x^1_n, x^2_n) + \dfrac{\partial f_2(\mathbf{x}_n)}{\partial x^1}(x^1 - x^1_n) + \dfrac{\partial f_2(\mathbf{x}_n)}{\partial x^2}(x^2 - x^2_n). \end{cases}$$

Решив полученную линейную задачу относительно $\mathbf{x} = (x^1, x^2)^{\mathrm{T}}$, найдем очередное приближение $\mathbf{x}_{n+1} = (x^1_{n+1}, x^2_{n+1})^{\mathrm{T}}$. В общем случае для системы уравнений расчетная формула метода Ньютона имеет вид

$$F'(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) = -F(\mathbf{x}_n), \quad F'(\mathbf{x}) = \left[\frac{\partial f_i}{\partial x^j}\right] \in \mathbf{R}^{m \times m}.$$

В качестве критерия окончания вычислений можно взять неравенство

$$\max_{i=1,\dots,m} |x_k^i - x_{k+1}^i| < \varepsilon.$$

Изучите сходимость метода Ньютона для системы

$$\begin{cases} x^3 - y^2 = 4, \\ xy^3 - y = 14, \end{cases} \qquad F'(x) = \begin{pmatrix} 3x^2 & -2y \\ y^3 & 3xy^2 - 1 \end{pmatrix},$$

в зависимости от выбора начального приближения.

— Сколько будет 2./3. * 3./2.*?*
— Может, 5? Или 3?
— Нет.
— Значит, полтора.
— Вот-вот. Полтора, или даже еще меньше. Но никак не 5 и не 3.

2.8. Дифференциальные уравнения

Построение адекватных математических моделей в форме дифференциальных уравнений для описания различных физических, биологических, химических, социальных процессов и последующее прямое моделирование представляют собой весьма нетривиальную прикладную задачу, лежащую на стыке различных научных областей и требующую математической грамотности и широкой эрудиции. Приводимые далее модели в большинстве случаев носят исключительно демонстрационный характер, поэтому следует скептически относиться к получаемым с их помощью прогнозам и помнить высказывание ученого с исключительной эрудицией, академика Валентина Павловича Дымникова: «Каждая модель воспроизводит только те эффекты, которые в нее заложены».

При решении дифференциальных задач данного раздела следует уделить внимание вопросам сходимости численно найденных дискретных функций к искомому решению при измельчении сеточного шага. Отдельная подзадача — визуализация полученных результатов, т. е. построение статических либо анимационных картинок. Отметим, что при численном моделировании реальных систем соответствующие дифференциальные уравнения и начальные данные необходимо предварительно

нормировать так, чтобы в процессе расчета значения искомых функций изменялись в окрестности единицы. Это позволяет уменьшить влияние вычислительной погрешности.

Задача 2-8-1. Проведите прямое моделирование процесса размножения биологического вида без учета возраста особи и внешних факторов на основе модели однотипной популяции с неограниченным ростом и дискретным шагом по времени:

$$N_{k+1} = N_k + aN_k - bN_k, \quad k = 0, 1, \ldots,$$

где N_k — число особей в k-й год. Начальное значение N_0, коэффициенты рождаемости a и смертности b считаются заданными.

Указание. В терминах коэффициента прироста $p = 1 + a - b$ имеем геометрическую прогрессию вида $N_k = p^k N_0$, N_0 — начальное условие. Отметим, что даже при «безобидном» годовом удвоении за сто лет получаем $2^{100} \approx 10^{30}$ особей. Для сравнения: поверхность суши порядка $150\,000\,000$ *км*$^2 \approx 1{,}5 \cdot 10^{20}$ *мм*^{2}, среднее расстояние до Солнца $150\,000\,000$ *км* $\approx 1{,}5 \cdot 10^{14}$ *мм*, диаметр диска галактики Млечный путь порядка $100\,000$ световых лет, т. е. 10^{18} *км* $\approx 10^{24}$ *мм*. Оцените прогностическое время «заселения» нашей галактики конкретным земным видом, получаемое согласно модели.

В связи с рассмотренной задачей приведем высказывание выдающегося отечественного математика, академика Николая Сергеевича Бахвалова: «Неоправданное привлечение абстрактных понятий математики редко приносит реальную пользу».

Задача 2-8-2. Исследуйте поведение модели ограниченного роста с отловом

$$N_{k+1} = \left(1 + p\,\frac{\overline{N} - N_k}{N_k}\right) N_k - M, \quad k = 0, 1, \ldots,$$

где $\overline{N}$ — оптимальное число особей, M — запланированный отлов, p — нормировочный коэффициент. Подберите значения параметров, порождающие различные типы динамики.

Задача 2-8-3. Одна из популярных моделей роста потребительского спроса имеет вид $y_{k+1} = Cy_k^2$, что означает увеличение коэффициента геометрической прогрессии при увеличении числа элементов. Докажите, что $y_k = C^{-1}(Cy_0)^{2^k}$.

Указание. Сделайте замену $v_k = Cy_k$ и по индукции покажите, что $v_k = v_0^{2^k}$. Отметим, что для $C = 1$, $y_0 = 2$, $n = 10$

имеем $y_{10} = 2^{1024} \approx 10^{308}$. Для сравнения: предположительное число атомов видимой части Вселенной порядка 10^{80}.

Задача 2-8-4. Проведите теоретический анализ процесса размножения популяции с двумя ступенями развития (N^0_k — молодняк, N^1_k — зрелые особи) и неограниченным ростом на основе дискретной модели

$$N^0_{k+1} = p_{10}N^1_k, \quad N^1_{k+1} = p_{11}N^1_k + p_{01}N^0_k$$

для различных начальных условий N^0_0, N^1_0 и значений коэффициентов p_{ij}. Численно проверьте правильность теоретических выкладок.

Указание. Покажите, что $N^1_{k+1} = aN^1_k + bN^1_{k-1}$, где $a = p_{11}$, $b = p_{10}p_{01}$ и значения N^1_0 и $N^1_1 = p_{11}N^1_0 + p_{01}N^0_0$ заданы. Далее проверьте, что если μ_1 и μ_2 — различные корни уравнения $\mu^2 - a\mu - b = 0$, то $N^1_k = c_1\mu_1^k + c_2\mu_2^k$; если $\mu_1 = \mu_2$, то $N^1_k = c_1\mu_1^k + c_2 k\mu_1^k$. Соответствующие коэффициенты c_1, c_2 находятся из начальных условий N^1_0, N^1_1. См. также задачу 1-1-36.

Задача 2-8-5. Для замкнутой дискретной модели «караси—щуки»

$$\begin{cases} C_{k+1} = (1 + a_c)C_k - b_c P_k C_k, \\ P_{k+1} = (1 - a_p)P_k + b_p C_k P_k \end{cases}$$

подберите параметры a_c, a_p, b_c, b_p и начальные условия C_0, P_0, порождающие качественно правдоподобную динамику.

Указание. Коэффициенты a_c и a_p отвечают за естественный прирост карасей C_k и убыль щук P_k соответственно. Величина C_kP_k характеризует вероятность встреч, что сказывается на количестве особей каждого вида. Например, можно взять $a_c = 0{,}06$, $b_c = 0{,}0008$, $a_p = 0{,}01$, $b_p = 0{,}000008$, $C_0 = 800$, $V_0 = 200$. Изучите, как изменится динамика, если на каждом шаге число особей округлять до целого числа.

Задача 2-8-6. Сформулируйте алгоритм взаимодействия в системе «Озеро: ряска, караси, щуки» на основе трехмерного массива «Озеро». Реализуйте прямое моделирование при различных входных параметрах. Попробуйте получить систему с устойчивой (квазипериодической) динамикой.

Указание. За основу можно взять правила математической игры «Жизнь».

Задача 2-8-7. Численно найдите приближенное решение дифференциального уравнения $y'(x) = y(x) + e^x \cos x$ на отрезке

$[0, \pi]$ при начальном условии $y(0) = 0$. Сравните его с точным решением $y(x) = e^x \sin x$.

Идеи реализации. Разобьем отрезок $[0, \pi]$ на N частей с шагом $h = \pi/N$ и будем искать приближенные значения $y(x_k) \approx \approx y_k$ только в точках $x_k = kh$, $k = 0, 1, \ldots, N$; $x_0 = 0$, $x_N = \pi$. Для этого заменим производную $y'(x)|_{x=x_k}$ по формуле

$$y'(x_k) = \frac{y(x_k + h) - y(x_k)}{h} + O(h),$$

что позволит записать уравнение на дискретную функцию $[y_0, y_1, \ldots, y_N]$:

$$y_{k+1} = y_k + hf(x_k, y_k), \quad k = 0, 1, \ldots, N-1; \quad y_0 = y(x_0).$$

Данный подход называется явным методом Эйлера.

Задача 2-8-8. Используя метод Эйлера, численно найдите приближенное решение уравнения $y'(x) = -ay(x)$ на отрезке $[0, 1]$ при начальном условии $y(0) = 1$. Сравните его с точным решением $y(x) = e^{-ax}$ для $a = 10, 1000$ при $N = 10, 10^2, 10^3, 10^4$.

Задача 2-8-9. Численно найдите приближенное решение дифференциального уравнения $y'(x) = -ay(x)$ на отрезке $[0, 1]$ при начальном условии $y(0) = 1$, используя неявный метод Эйлера:

$$y_{k+1} = y_k - hay_{k+1}, \quad k = 0, 1, \ldots, N-1; \quad y_0 = y(x_0).$$

Сравните его с точным решением $y(x) = e^{-ax}$ для $a = 10, 1000$ при $N = 10, 10^2, 10^3, 10^4$.

Задача 2-8-10. Напишите программу для численного решения дифференциального уравнения $y'(x) = f(x, y(x))$ на отрезке $[a, b]$ с начальным условием $y(a) = y_a$, используя расчетные формулы метода Адамса второго порядка точности:

$$y_0 = y(x_0), \quad x_0 = a, \quad x_k = x_0 + kh, \quad k = 0, \ldots, N, \quad h = \frac{b-a}{N},$$
$$y^*_{k+1} = y_k + hf(x_k, y_k),$$
$$y_{k+1} = y_k + \frac{h}{2}(f(x_k, y_k) + f(x_{k+1}, y^*_{k+1})).$$

Решение постройте на N-кратном вызове функции

```
int Step (double * yk1, double *yk, double h,
          (void) (*fun)(double *,
          const double *, const double)),
```

где `fun` является указателем на функцию, отвечающую за вычисление правой части f(x, y). Для дифференциальной задачи с известным решением сравните найденное решение с точным ответом.

Рассмотренные методы Эйлера, Адамса, а также алгоритмы, которые будут приведены далее, можно применять для решения систем дифференциальных уравнений $\mathbf{y}' = \mathbf{F}(x, \mathbf{y})$, $\mathbf{y} = (y_1(x), \ldots, y_m(x))^T$. Здесь и далее верхний индекс T означает транспонирование, т. е. запись соответствующей строки в виде вектора-столбца. В этом случае дискретная функция $\mathbf{y}_k$ также является вектором-столбцом: $\mathbf{y}_k = (y_{1,k}, \ldots, y_{m,k})^T$, где первый индекс i компоненты $y_{i,k}$ отвечает за номер координаты вектора, а второй индекс k показывает соответствие вектора точке x_k.

Задача 2-8-11. Напишите программу для численного решения дифференциального уравнения $y'(x) = f(x, y(x))$ на отрезке $[a, b]$ с начальным условием $y(a) = y_a$, используя расчетные формулы метода Рунге—Кутты третьего порядка точности:

$$k_1 = hf(x_i, y_i), \quad k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right),$$
$$k_3 = hf(x_i + h, y_i - k_1 + 2k_2), \quad y_{i+1} = y_i + \frac{k_1 + 4k_2 + k_3}{6}.$$

Для следующих задач с известным решением

$y'(x) = 3x^2$, $y(0) = 1$, $x \in [0, 10]$, при $y(x) = x^3 + 1$,
$y'(x) = y(x)$, $y(0) = 1$, $x \in [0, 1]$, при $y(x) = e^x$,
$y'(x) = e^x$, $y(0) = 1$, $x \in [0, 1]$, при $y(x) = e^x$,

исследуйте близость функций $y(x_k)$ и y_k, $k = 0, 1, \ldots, N$, при $h = 10^{-m}$, $m = 0, 3, 5, 7$.

Задача 2-8-12. Напишите программу для численного решения дифференциального уравнения $y'(x) = f(x, y(x))$ на отрезке $[a, b]$ с начальным условием $y(a) = y_a$, используя расчетные формулы метода Рунге—Кутты четвертого порядка точности:

$$k_1 = hf(x_i, y_i), \quad k_2 = hf\left(x_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right),$$
$$k_3 = hf\left(x_i + \frac{h}{2}, y_i + \frac{k_2}{2}\right), \quad k_4 = hf(x_i + h, y_i + k_3),$$
$$y_{i+1} = y_i + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}.$$

Исследуйте близость функций $y(x_k)$ и y_k, $k = 0, 1, \ldots, N$, на примере задач с известным решением (см. задачу 2-8-11).

Задача 2-8-13. Численно найдите решение системы Лотки—Вольтерры

$$\begin{cases} x'(t) = \alpha x(t) - \mu x(t)y(t), \ \alpha = 1, \ \mu = 3, \\ y'(t) = -\beta y(t) + \nu y(t)y(t), \ \beta = 1, \ \nu = 1, \end{cases}$$

для различных начальных значений $x(0)$, $y(0)$.

Указание. Реализуйте, например, метод Рунге—Кутты. Для полученных дискретных функций x_k и y_k, $k = 0, 1, \ldots, N$, постройте график $y_k(x_k)$ и нарисуйте параметрическую кривую $\{(x_k, y_k)\}$. Сравните с задачей 2-8-5.

Задача 2-8-14. Для системы Лоренца

$$\begin{cases} x'(t) = \delta(y - x), \ \delta = 10{,}0, \\ y'(t) = rx - y - xz, \ r = 28{,}0, \\ z'(t) = xy - bz, \ b = \dfrac{8}{3}, \end{cases}$$

численно найдите решение для различных начальных условий $x(0)$, $y(0)$, $z(0)$ и постройте в пространстве $\mathbf{R}^3$ параметрическую кривую $\{(x_k, y_k, z_k), \ k = 0, 1, \ldots, N\}$.

Задача 2-8-15. Пусть из некоторого положения с координатами $\mathbf{S}(0) = (x_0, y_0, z_0)^T$ брошено точечное тело с начальной скоростью $\mathbf{U}(0) = (u_0, v_0, w_0)^T$. Считая, что на тело действует только сила тяжести, запишите систему дифференциальных уравнений, описывающую изменение вектора координат $\mathbf{S}(t) = (x(t), y(t), z(t))^T$ и вектора скорости $\mathbf{V}(t) = (u(t), v(t), w(t))^T$. Аналитически и численно найдите ее решение. Постройте соответствующие графики для различных значений начальной координаты и скорости.

Решение. Второй закон Ньютона и уравнения движения приводят к следующей системе:

$$\begin{cases} u'(t) = 0, \ v'(t) = 0, \ w'(t) = -g(x, y, z); \\ x'(t) = u(t), \ y'(t) = v(t), \ z'(t) = w(t); \\ u(0) = u_0, \ v(0) = v_0, \ w(0) = w_0; \\ x(0) = x_0, \ y(0) = y_0, \ z(0) = z_0. \end{cases}$$

Если $g(x, y, z) = \text{const}$, то решение системы имеет вид

$$\begin{cases} u(t) = u_0, \ v(t) = v_0, \ w(t) = w_0 - gt; \\ x(t) = x_0 + u_0 t, \ y(t) = y_0 + v_0 t, \ z(t) = z_0 + w_0 t - \dfrac{gt^2}{2}. \end{cases}$$

При численном решении исходной системы можно использовать, например, явный метод Эйлера:

$$\begin{cases} \frac{u_{n+1}-u_n}{\tau}=0,\ \frac{v_{n+1}-v_n}{\tau}=0,\ \frac{w_{n+1}-w_n}{\tau}=-g(x_n,y_n,z_n); \\ \frac{x_{n+1}-x_n}{\tau}=u_n,\ \frac{y_{n+1}-y_n}{\tau}=v_n,\ \frac{z_{n+1}-z_n}{\tau}=w_n \end{cases}$$

при заданных значениях u_0, v_0, w_0, x_0, y_0, z_0. Если g считать константой, то решение системы разностных уравнений также можно выписать в явном виде.

Задача 2-8-16. Пусть из некоторого положения с координатами $\mathbf{S}(0)=(x_0,y_0,z_0)^{\mathrm{T}}$ брошено точечное тело с начальной скоростью $\mathbf{U(0)}=(u_0,v_0,w_0)^{\mathrm{T}}$. Считая, что на тело действует сила тяжести, зависящая от высоты, сила сопротивления воздуха, зависящая от скорости, и управляющая сила, вызывающая дополнительное ускорение $\mathbf{A}(t)=\big(a_x(t),a_y(t),a_z(t)\big)^{\mathrm{T}}$, выпишите систему дифференциальных уравнений, описывающих изменения координаты $\mathbf{S}(t)=\big(x(t),y(t),z(t)\big)^{\mathrm{T}}$ и скорости $\mathbf{V}(t)=\big(u(t),v(t),w(t)\big)^{\mathrm{T}}$. Численно найдите решение системы и постройте соответствующие графики для различных параметров задачи.

Указание. Например, можно считать, что

$$\begin{cases} u'(t)=-\alpha|u(t)|-\beta u^2(t)+a_x(t), \\ v'(t)=-\alpha|v(t)|-\beta v^2(t)+a_y(t), \\ w'(t)=-g+3\cdot 10^{-6}z(t)-\alpha|w(t)|-\beta w^2(t)+a_z(t), \\ x'(t)=u(t),\ y'(t)=v(t),\ z'(t)=w(t). \end{cases}$$

Если реализовать вывод текущих значений $\mathbf{S}(t)$, $\mathbf{V}(t)$, $\mathbf{A}(t)$ с возможностью интерактивного ввода с клавиатуры $\mathbf{A}(t)$, то можно смоделировать управление «мягкой посадкой».

Задача 2-8-17. Пусть в пространстве имеется два точечных тела с массами M, m, где $M\gg m$. Проведите численное моделирование движения данной системы в «приближении пробного тела», т. е. учитывая только гравитационные силы и считая, что тяжелое тело покоится в начале координат.

Указание. Введем обозначения $X(t)=\big(x_1(t),x_2(t),x_3(t)\big)^{\mathrm{T}}$ для вектора координат и $U(t)=\big(u_1(t),u_2(t),u_3(t)\big)^{\mathrm{T}}$ для вектора скорости второго тела. Тогда согласно закону всемирного

тяготения соответствующие уравнения движения имеют вид

$$\begin{cases} mu_i'(t) = -G\dfrac{mM}{r^2(t)}\dfrac{x_i(t)}{r(t)}, \ r(t) = \sqrt{x_1^2(t) + x_2^2(t) + x_3^2(t)}, \\ x_i'(t) = u_i(t), \ i = 1, 2, 3. \end{cases}$$

Данную систему необходимо дополнить нормированными начальными условиями, положив, например, $GM = 1$, $X(0) = (0, 1, 1)$, $U(0) = (1, 0, 0)$.

Задача 2-8-18. Пусть в пространстве имеется n точечных тел, каждое из которых характеризуется массой m_i, вектором координат $X_i(t) = \big(x_{i,1}(t), x_{i,2}(t), x_{i,3}(t)\big)^T$ и вектором скорости $U_i(t) = \big(u_{i,1}(t), u_{i,2}(t), u_{i,3}(t)\big)^T$, $i = 1, 2, \ldots, n$. Провести численное моделирование движения данной системы, учитывая только гравитационные силы.

Указание. Согласно закону всемирного тяготения и правилу сложения сил на i-е тело в каждый момент времени t действует суммарная сила $F_i(t) = \sum\limits_{j \neq i}^{n} F_{i,j}(t)$. При этом вектор $F_{i,j}(t)$ направлен от i-го тела к j-му, а его длина может быть найдена по формуле $F_{i,j}(t) = G\dfrac{m_i m_j}{r_{i,j}^2(t)}$. Здесь G — гравитационная постоянная, $r_{i,j}(t)$ — евклидова длина вектора $(X_i(t) - X_j(t))$. Применяя второй закон Ньютона $m_i a_i(t) = F_i(t)$ и учитывая уравнения движения $U_i'(t) = a_i(t)$, $X_i'(t) = U_i(t)$, получаем искомую систему

$$\begin{cases} U_i'(t) = G\sum\limits_{j \neq i}^{n} \dfrac{m_j}{r_{i,j}^3(t)}\big(X_j(t) - X_i(t)\big); \\ X_i'(t) = U_i(t), \ i = 1, \ldots, n, \end{cases}$$

при заданных начальных условиях $X_i(0)$, $U_i(0)$, $i = 1, 2, \ldots, n$. Напомним, что для получения разумных результатов численного моделирования, например, движения планет нашей Солнечной системы следует провести перенормировку уравнений (см. задачу 2-8-17).

Задача 2-8-19. Приближенно найдите температуру $u(x)$ однородного тонкого стержня длины L с коэффициентом температуропроводности μ при условии, что на его концах температура фиксирована и равна $u(0) = \nu_0$, $u(L) = \nu_L$, а также задана функция нагрева внешним источником $f(x)$, $x \in (0, L)$.

Для расчета используйте дифференциальное уравнение

$$-\mu u''(x) = f(x), \quad x \in (0, L), \quad u(0) = \nu_0, \quad u(L) = \nu_L.$$

Указание. Разобьем отрезок $(0, L)$ на M частей с шагом $h = L/M$ и будем искать приближенные значения u_m для $u(x_m)$ только в точках $x_m = mh$, $m = 0, 1, \ldots, M$; $x_0 = 0$, $x_M = L$. Для этого заменим производную $u''(x)|_{x=x_m}$ по формуле $u''(x_m) = (u(x_m + h) - 2u(x_m) + u(x_m - h))/h^2 + O(h^2)$ и получим систему уравнений на компоненты вектора неизвестных $\mathbf{u} = [u_0, u_1, u_2, \ldots, u_{N-2}, u_{N-1}, u_N]^T$

$$-\mu \frac{u_{m+1} - 2u_m + u_{m-1}}{h^2} = f(x_m), \quad m = 1, \ldots, M-1;$$
$$u_0 = \nu_0, \quad u_M = \nu_L.$$

Для решения полученной системы уравнений $A\mathbf{u} = \mathbf{b}$ с матрицей A вида

$$\begin{pmatrix}
1 & & 0 & 0 & \ldots & 0 & 0 & 0 & 0 \\
-\frac{\mu}{h^2} & \frac{2\mu}{h^2} & -\frac{\mu}{h^2} & 0 & \ldots & 0 & 0 & 0 & 0 \\
0 & -\frac{\mu}{h^2} & \frac{2}{h^2} & -\frac{\mu}{h^2} & \ldots & 0 & 0 & 0 & 0 \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
0 & 0 & 0 & 0 & \ldots & -\frac{\mu}{h^2} & \frac{2\mu}{h^2} & -\frac{\mu}{h^2} & 0 \\
0 & 0 & 0 & 0 & \ldots & 0 & -\frac{\mu}{h^2} & \frac{2\mu}{h^2} & -\frac{\mu}{h^2} \\
0 & 0 & 0 & 0 & \ldots & 0 & 0 & & 1
\end{pmatrix}$$

и правой частью $\mathbf{b} = [\nu_0, f(x_1), f(x_2), \ldots, f(x_{N-2}), f(x_{N-1}), \nu_L]^T$ можно применить метод Гаусса, но гораздо лучше — метод прогонки, являющийся эффективной модификацией метода Гаусса для трехдиагональной системы (см. задачу 2-6-9). Считая, что искомое решение имеет вид

$$u(x) = \sin\left(\frac{\pi x}{L}\right) + \nu_0 + \frac{\nu_L - \nu_0}{L} x,$$

найдем, два раза дифференцируя $u(x)$, соответствующую правую часть:

$$f(x) = \mu \left(\frac{\pi}{L}\right)^2 \sin\left(\frac{\pi x}{L}\right).$$

Пусть $\mu = 0{,}1$, $L = 1$, $\nu_0 = 1$, $\nu_L = 2$. Для полученной задачи с известным ответом сравните вычисленное решение $[u_0, \ldots, u_N]^T$ с точными значениями $[u(x_0), \ldots, u(x_N)]^T$ при $N = 3, 10, 1000$.

Задача 2-8-20. Приближенно вычислите, как меняется со временем температура $u(t,x)$ в однородном тонком стержне длины L с коэффициентом температуропроводности μ при условии, что заданы начальная температура стержня $u(0,x) = u^0(x)$, температура на его концах $u(t,0) = \nu_0(t)$, $u(t,L) = \nu_L(t)$ и функция внешнего нагрева $f(t,x)$, $x \in (0,L)$, $t \in [0,T]$. Для расчета используйте уравнение

$$\frac{\partial}{\partial t}u(t,x) = \mu\frac{\partial^2}{\partial x^2}u(t,x) + f(t,x),\ x \in (0,L),\ t \in [0,T],$$
$$u(t,0) = \nu_0(t),\ u(t,L) = \nu_L(t),\ u(0,x) = u^0(x).$$

Указание. Разобьем отрезок $[0,L]$ на M частей с шагом $h = L/M$, отрезок $[0,T]$ на N частей с шагом $\tau = T/N$ и будем искать приближенные значения $u(t_n,x_m) \approx u_m^n$ в точках $x_m = mh$, $m = 0,1,\ldots,M$, $t_n = n\tau$, $n = 0,1,\ldots,N$. Для этого заменим производные по времени и пространству на приближенные разностные аналоги

$$\frac{\partial}{\partial t}u(t_n,x_m) = \frac{u(t_n+\tau,x_m) - u(t_n,x_m)}{\tau} + O(\tau),$$
$$\frac{\partial^2}{\partial x^2}u(t_n,x_m) = \frac{u(t_n,x_m+h) - 2u(t_n,x_m) + u(t_n,x_m-h)}{h^2} + O(h^2).$$

Это позволит получить систему уравнений на дискретные вектор-функции $[u_0^n, u_1^n, \ldots, u_M^n]$, $n = 0,1,\ldots,N$:

$$\frac{u_m^{n+1} - u_m^n}{\tau} = \mu\frac{u_{m+1}^n - 2u_m^n + u_{m-1}^n}{h^2} + f(t_n,x_m),$$
$$n = 0,\ldots,N-1,\ m = 1,\ldots,M-1;$$
$$u_m^0 = u^0(x_m),\ m = 0,\ldots,M;$$
$$u_0^n = \nu_0(t_n),\ u_M^n = \nu_L(t_n),\ n = 0,\ldots,N.$$

Данный метод аппроксимации называется явной двухслойной схемой. Отметим, что необходимым условием использования схемы является условие устойчивости $\tau \leqslant h^2/(2\mu)$. Нарушение условия приводит к ошибочному ответу при $\tau, h \to 0$. Исследуйте вопрос близости найденного и точного решений на примере следующей задачи: $L = \pi$, $\mu = 1$, $f(x) = \sin x$, $u(t,x) = (1 + e^{-t})\sin x$, $\nu_0(t) = \nu_1(t) = 0$.

Задача 2-8-21. Приближенно вычислите, как меняется со временем температура $u(t,x)$ в однородном тонком стержне,

используя полностью неявную схему

$$\frac{u_m^{n+1}-u_m^n}{\tau}=\mu\frac{u_{m+1}^{n+1}-2u_m^{n+1}+u_{m-1}^{n+1}}{h^2}+f(t_{n+1},x_m),$$
$$n=0,\dots,N-1,\ m=1,\dots,M-1;$$
$$u_m^0=u^0(x_m),\ m=0,\dots,M;$$
$$u_0^n=\nu_0(t_n),\ u_M^n=\nu_L(t_n),\ n=0,\dots,N.$$

Указание. Отметим, что схема пригодна для расчетов при любом соотношении шагов $\tau, h\to 0$, однако на каждом шаге по времени требует решения системы уравнений $A\mathbf{u}^{n+1}=\mathbf{b}$ относительно вектора $\mathbf{u}^{n+1}=[u_0^{n+1},\dots,u_M^{n+1}]^T$. Здесь A имеет вид

$$\begin{pmatrix}
1 & 0 & 0 & \dots & 0 & 0 & 0 & 0\\
-\frac{\mu}{h^2} & \frac{2\mu}{h^2}+\frac{1}{\tau} & -\frac{\mu}{h^2} & \dots & 0 & 0 & 0 & 0\\
0 & -\frac{\mu}{h^2} & \frac{2}{h^2}+\frac{1}{\tau} & -\frac{\mu}{h^2} & \dots & 0 & 0 & 0\\
\dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots\\
0 & 0 & 0 & \dots & -\frac{\mu}{h^2} & \frac{2\mu}{h^2}+\frac{1}{\tau} & - & 0\\
0 & 0 & 0 & \dots & 0 & -\frac{\mu}{h^2} & \frac{2\mu}{h^2}+\frac{1}{\tau} & -\frac{\mu}{h^2}\\
0 & 0 & 0 & \dots & 0 & 0 & 0 & 1
\end{pmatrix},$$

$\mathbf{b}=[\nu_0(t^{n+1}),f(x_1)+u_1^n/\tau,\dots,f(x_{N-1})+u_{N-1}^n/\tau,\nu_L(t^{n+1})]^T$. Исследуйте близость найденного дискретного решения u_m^n к точному непрерывному решению $u(t_n,x_m)$ на примере дифференциальной задачи с известным ответом (см. задачу 2-8-20).

Ассистент к ассистенту:
— А что такое условный рефлекс?
Тот ему:
— Вот обучил обезьян: захотят есть — дергают рычаг.
Обезьяна к приятелю:
— Слушай, ты понял, что они там говорили про условный рефлекс?
— Ага, вот я щас дерну за этот рычаг, и вон тот, в белом халате, сразу же побежит мне за бананом.

Вопрос: что еще, кроме шара, не зависит от точки зрения?

Глава 3
СТРУКТУРЫ ДАННЫХ

В данном разделе рассматриваются реализации классических структур данных — стека, дека, очереди, списка, дерева. Каждая из этих структур позволяет хранить некоторое количество элементов, добавлять новые, извлекать и удалять существующие, а также предоставлять непосредственный доступ к отдельным значениям. Структуры различаются между собой правилами доступа к элементам и внутренними логическими связями.

В задачах, приведенных в этом разделе, предлагается построить структуры данных для хранения элементов некоторого абстрактного типа `Type`. Можно считать, что для программы на языке C этот тип определен при помощи оператора **typedef**. Например, для элементов, являющихся целыми числами, имеем **typedef int** `Type;`. В этом случае программная реализация каждой структуры данных должна представлять собой отдельный файл, в котором содержатся объявления некоторого набора глобальных статических объектов (для реализации внутреннего представления данных) и определения функций (для реализации требуемых действий с данными), а также заголовочный файл с описаниями используемых типов данных и прототипами функций.

При использовании языка C++ реализация выливается в построение некоторого класса (или иерархии классов) с соответствующими скрытыми членами и общедоступными методами. Для обеспечения возможности использовать реализацию для различных типов данных можно создать шаблон построенного класса с помощью конструкции **template** <**class** `Type`>.

Практическая реализация каждой структуры данных должна сопровождаться набором юнит-тестов, позволяющих проверить корректность работы кода в различных ситуациях.

Современные библиотеки языка C++ включают развитые реализации так называемых «контейнерных классов», к которым относятся вышеупомянутые списки, деревья, множества, очереди и т. д. Это сделано в рамках библиотеки STL (Standard Template Library). Таким образом, работа с подобными структурами может опираться на реализации STL, что обычно и происходит в большинстве производственных проектов. Однако мы предлагаем читателю все же попытаться разобраться во «внутренней кухне» построения и использования разнообразных

способов организации данных и самостоятельно реализовать базовые алгоритмические идеи и приемы работы со структурами с заданной дисциплиной доступа к данным. Вдумчивый разбор особенностей реализации таких структур поможет в полной мере понимать достоинства и недостатки различных программных решений и делать правильный выбор между библиотечными и собственными реализациями в зависимости от накладываемых требований и специфики решаемой задачи.

3.1. Стек, дек, очередь

Стек. Так называется структура данных, в которую можно последовательно добавлять элементы, но в каждый момент времени доступным является только элемент, добавленный в стек последним. Только этот элемент, называемый вершиной стека, можно извлечь (или удалить) из стека, после чего становится доступным предыдущий добавленный элемент. Эта дисциплина доступа часто обозначается аббревиатурой LIFO (Last In, First Out), т. е. последний добавленный элемент извлекается из стека первым.

Задача 3-1-1. Постройте реализацию стека элементов абстрактного типа `Type`.

Идеи реализации. Для реализации достаточно выделить участок памяти для размещения необходимого количества элементов, размещать поступающие элементы последовательно один за другим рядом друг с другом и хранить указатель на последний элемент стека, т. е. на вершину. После добавления (извлечения) элемента указатель текущей вершины будет перемещаться к следующей (предыдущей) позиции выделенного участка памяти.

Решение. Будем считать, что текущее состояние стека описывают три переменные: `mem` содержит адрес начала области элементов стека, `top` — адрес текущей вершины, `size` — текущее количество элементов. При создании стека положим `top = mem + maxsize`, `size = 0`. Для наглядности кода опишем переменные на внешнем уровне, хотя для гибкости было бы лучше поместить их в отдельную структуру и передавать указатель на нее в каждую функцию.

Считая для определенности, что тип `Type` есть `int`, приведем реализацию стека на языке C (а далее — на C++).

```
/*---   Файл stack.h   ---*/
typedef int Type;
/* создать стек на maxsize элементов: */
int St_Init(size_t maxsize);
/* закончить работу:                  */
void St_Term(void);
/* добавить элемент  val:             */
int St_Push(Type val);
/* удалить вершину:                   */
int St_Del(void);
/* удалить вершину и поместить ее
   значение по указателю dst:      */
int St_Pop(Type *dst);
/* указатель на вершину:              */
Type *St_Top(void);
/* текущее количество элементов:         */
size_t St_Size(void);
/* количество свободных мест:      */
size_t St_Room(void);
/*---   конец файла stack.h   ---*/
```

Функции `St_Init`, `St_Push`, `St_Del`, `St_Pop` возвращают 0 при успешном выполнении требуемой операции и -1 в противном случае.

```
/*---   Файл  stack.c   ---*/
#include <stdlib.h>
#include "stack.h"
/* начало области элементов стека: */
static Type *mem;
/* указатель на вершину стека: */
static Type *top;
/* текущее количество элементов: */
static size_t size;
int St_Init(size_t maxsize) {
  size = 0;
  top = mem = (Type *)malloc(sizeof(Type)
                                    * maxsize);
  if (mem) top += maxsize;
  return (mem) ? 0 : -1;
}
```

```
void St_Term(void) {
  if (mem) free(mem);
  size = 0;
  mem = top = NULL;
  return;
}
int St_Push(Type val) {
  return
   (top != mem) ? *(--top) = val, ++size, 0 : -1;
}
int St_Pop(Type *dst) {
  return
    (size > 0) ? *dst = *(top++), --size, 0 : -1;
}
int St_Del(void) {
  return (size > 0) ? ++top, --size, 0 : -1;
}
Type *St_Top(void) {
  return (size > 0) ? top : NULL;
}
size_t St_Size(void) { return size; }
size_t St_Room(void) {
  return (size_t) (top - mem);
}
```

Реализация на языке C++.

```
#define DEFAULT_SIZE 10
template <class Type>
class Stack {
 private:
  Type *mem, *top;
  size_t size;
 public:
  /* Конструктор и деструктор: */
  Stack(size_t maxsize = DEFAULT_SIZE) {
    mem = new Type[maxsize];
    top = mem + maxsize;
    size = 0;
  }
  ~Stack() {
```

```
    delete[] mem;
  }
  /* Методы работы со стеком: */
  int Push(const Type &value) {
    return (top != mem) ?
               *(--top) = value, ++size, 0 : -1;
  }
  int Pop(Type &dst) {
    return (size) ?
                dst = *(top++), --size, 0 : -1;
  }
  int Del() {
   return (size) ? ++top, --size, 0 : -1;
  }
  Type &Top() { return *top; }
  bool Empty() { return size == 0; }
  size_t Room() { return (size_t) (top - mem); }
  bool Success() { return mem != NULL; }
};
```

Замечание. Функция `Success()` добавлена для того, чтобы мы могли обнаружить отказ в выделении памяти под стек, не привлекая механизма исключений языка C++.

Задача 3-1-2. Напишите функцию **void** `St_Print(void)` печати содержимого стека в наглядной форме (с отображением пустых ячеек). Составьте тестирующую программу, которая бы позволяла в режиме диалога вызывать все функции стека и выводила на экран состояние стека после каждой операции. Проверьте работоспособность реализации для типов **int** и **char**.

Замечание. Работа со стеком на основе глобальных переменных в некоторых случаях может оказаться обременительной. Более гибкие и защищенные от ошибок конструкции можно реализовать, поместив соответствующие переменные в структуру и передавая в функции указатель на ее содержимое.

Задача 3-1-3. Постройте реализацию стека элементов абстрактного типа `Type` на основе структуры следующего вида:

```
struct Stack {
  Type *mem; // начало области элементов стека
  size_t size; // текущее количество элементов
  size_t maxsize; // размер стека
```

```
};
```

Указание. Функция инициализации в этом случае может иметь, например, следующий вид:

```
struct Stack *St_Init(size_t maxsize) {
  struct Stack *St = NULL;
  St =
    (struct Stack *)malloc(sizeof(struct Stack));
  if (St) {
    St->mem =
          (Type *)malloc(sizeof(Type) * maxsize);
    if (St->mem != NULL) {
      St->size = 0;
      St->maxsize = maxsize;
    } else {
      free(St);
      St = NULL;
    }
  }
  return St;
}
```

Задача 3-1-4. Пусть имеется некоторое строковое выражение, содержащее скобки трех видов {, }, [,], (,). На основе стека с элементами типа **char** реализуйте функцию проверки баланса скобок.

Идеи реализации. Будем посимвольно анализировать имеющееся выражение и добавлять все встречающиеся скобки в стек. Если при добавлении очередной закрывающей скобки вершина содержит открывающую скобку такого же вида, то оба элемента из стека удаляются. Выражение корректно, если по завершении стек окажется пустым.

Задача 3-1-5. Постройте реализацию стека, элементом которого является структура вида

```
struct User {
  char FirstName[10];
  char LastName[10];
  unsigned int UId;
};
```

Проведите тестирование программы, подготовив файл с последовательностью команд работы со стеком и файл с входными данными. Функция `main` должна открыть эти файлы и, последовательно считывая из первого файла команды, а из второго данные (при выполнении инструкций Init и Push), вызывать соответствующие функции. Все результаты выполнения команд `Print` (см. задачу 3-1-2) должны быть сохранены в одном файле.

Задача 3-1-6. Постройте реализацию двух стеков, элементами которого являются целые числа.

Идеи реализации. В функции `main()` создадим массив переменных `struct Stack st[2];` и при вызове базовых функций `St_Init`, `St_Push`, `St_Del`, `St_Pop` будем дополнительно передавать в них адрес структуры `&st[k]` активного в данный момент k-го стека.

Задача 3-1-7. Постройте реализацию N стеков с элементами типа `Type`.

Задача 3-1-8. Пусть имеется стек чисел (аргументов) и символьный стек бинарных операций. Напишите функцию вычисления результата арифметического выражения по следующему правилу: пока стек операций не пуст, выполняем соответствующее действие с двумя верхними числами из стека; удаляем использованные аргументы из стеков; результат вычисления кладем в стек аргументов. Приоритет арифметических действий в данном случае определяется их последовательностью в стеке.

Задача 3-1-9. Пусть имеется некоторое строковое выражение, содержащее корректную математическую формулу с бинарными операциями, записанную в обратной польской нотации (см. раздел 4.4). По определению выражение состоит из чисел и знаков операций и обрабатывается слева направо. Если встречается знак операции, то указанное действие выполняется с двумя идущими перед ним числами в порядке их записи. Результат записывается вместо участвовавших чисел и оператора. Процесс повторяется до тех пор, пока не останется последнее число, являющееся ответом. Реализуйте процедуру считывания формулы в обратной польской нотации и процедуру вычисления соответствующего результата.

Задача 3-1-10. Постройте реализацию стека, элементами которого являются указатели на строки, содержащие не более ста символов.

Идеи реализации. Определим следующую структуру:

```
struct Stack {
  char **mem; // начало области элементов стека
  int size; //текущее количество элементов
  int maxsize;  // размер стека
  char buf[100];
};
```

Очередную строку будем считывать в `buf[]`, далее вычислять ее реальную длину, выделять для хранения с помощью функции `malloc()` память требуемого размера, сохраняя указатель в стеке, и копировать из буфера элемент в выделенную память. При удалении элемента выделенная для его хранения память высвобождается функцией `free`.

Задача 3-1-11. Постройте реализацию стека, элементами которого являются указатели на строки, если известно ограничение на суммарную длину всех строк.

Идеи реализации. На первом шаге выделим не только место для хранения элементов стека, но и блок памяти `char *buf` для хранения всех строк. Будем поддерживать указатель `char *top` на начало свободной области в блоке, последовательно укладывая строки в выделенную область и записывая в `mem[i]` адрес начала `i`-й строки.

Задача 3-1-12. Реализуйте стек строк на базе одного большого блока памяти.

Идеи реализации. Выделим достаточно большой блок памяти и будем укладывать строки — последовательности символов, заканчивающиеся символом с кодом 0x0A, — одну за другой в этом блоке. Для добавления достаточно поддерживать указатель на начало свободной области в блоке и сравнивать длину этой области с длиной добавляемой строки. Для удаления можно отыскивать в занятой части блока конец предыдущей строки и передвигать указатель свободной позиции на следующий за этим концом символ.

Задача 3-1-13. Добавьте к решению задачи 3-1-12 возможность эффективного сохранения стека в файл и считывания построенного стека из такого файла, используя функции `fread`, `fwrite`.

Задача 3-1-14. Реализация стека строк из задачи 3-1-13 требует последовательного поиска конца предыдущей строки при удалении элемента. Модифицируйте реализацию, помещая

вслед за каждой добавленной строкой ее длину. Это позволит сразу вычислить значение адреса начала последней добавленной строки по адресу начала свободной области в стеке.

Задача 3-1-15. Реализуйте стек строк (см. задачу 3-1-12) при условии, что стек хранится в файле, причем начало файла соответствует дну стека, а вершиной стека является последняя строка в файле. При добавлении строки в стек размер файла должен увеличиваться, а при удалении строки — уменьшаться, так как файл должен перезаписываться без последней строки (для работы с файлом следует использовать функции `fread`, `fwrite` и т. п.). Реализуйте следующие функции:

```
int OpenFileStack(char *filename);
int PushString(char *str);
int PopString(char *str);
int CloseFileStack(void);
```

Каждая из этих функций возвращает 0 в случае успешного выполнения требуемой операции и −1, если выполнение операции по каким-либо причинам невозможно.

Модифицируйте реализацию так, чтобы уменьшение длины файла при удалении строк происходило не каждый раз, а только при накоплении достаточно большой суммарной длины удаленных строк, например 1024 байт). Проверьте, как повлияет такая модификация на быстроту функционирования стека.

Задача 3-1-16. Реализуйте (см. задачу 3-1-15) файловый стек, элементами которого будут байтовые записи произвольной длины, хранимые в следующем формате:

```
<запись 1>
<длина записи 1>
<запись 2>
<длина записи 2>
.....
.....
<запись k>
<длина записи k>
```

где `<длина записи n>` есть поле, содержащее значение числа типа `uint64_t` — длины соответствующей записи. Последняя запись в файле (т. е. `<запись k>`) соответствует вершине стека.

Реализуйте функции со следующими прототипами:

```
/* функция открывает файл: */
int OpenFileStack(char *filename);
/* функция помещает в стек length байт,
   размещенных начиная с адреса record:  */
int PushRecord(void *record, size_t length);
/* функция берет вершину стека и помещает ее
   в области памяти по адресу record, значение
   длины записи помещается по адресу length: */
int PopRecord(void *record, size_t *length);
/* функция возвращает длину записи на
   вершине стека: */
uint64_t TopLength(void);
/* функция закрывает файл: */
int CloseFileStack(void);
```

Функции возвращают 0 в случае успешного выполнения и -1 при отказе.

Задача 3-1-17. Постройте реализацию файлового стека строк объединяющую идеи задач 3-1-15, 3-1-16. Строки частично хранятся в памяти, частично — на диске. В памяти выделяются два блока. Строки накапливаются сначала в первом блоке, а когда он заполнится — во втором. При заполнении второго блока первый блок записывается в файл (по стековому принципу) и блоки обмениваются своими номерами: второй (занятый) блок становится первым, а первый (теперь свободный) становится вторым. При удалении процедура обращается: пока имеются строки в блоках, работа идет в памяти, а когда оба блока становятся пустыми, в первый считываются данные из файлового стека блоков.

Дек. Эту структуру можно рассматривать как обобщение стека за счет возможности добавлять и удалять элементы «с обоих концов» набора данных, т. е. имеется два текущих элемента, к которым разрешен доступ и которые называются началом и концом дека.

Дисциплина доступа к началу и концу дека аналогична дисциплине работы с вершиной стека: разрешено чтение, добавление и извлечение элементов.

Задача 3-1-18. Постройте реализацию дека элементов абстрактного типа `Type`.

Идеи реализации. Будем размещать элементы в выделенном участке памяти вплотную друг к другу и хранить два

указателя — на начало и конец дека. При добавлении или удалении элементов эти указатели будем перемещать на позицию следующего или предыдущего элементов. Для корректной работы дека нужно формально «замкнуть в кольцо» выделенную область: при необходимости новая позиция «перебрасывается» с позиции за концом области в начало или с позиции перед началом — в конец. Если принять, что начало, конец выделенной области памяти, голова, хвост и размер дека задаются переменными

```
Type *begmem, *endmem;
Type *head, *tail;
size_t size;
```

(глобальными для компактности кода), то процедуры определения следующего и предыдущего элементов могут выглядеть, например, так:

```
Type *Next(Type *pos) {
  return (pos == endmem) ? begmem : pos + 1;
}
Type *Prev(Type *pos) {
  return (pos == begmem) ? endmem : pos - 1;
}
```

Реализуйте на языке C дек со следующим интерфейсом:

```
/*---   Файл deque.h   ---*/
typedef .....  Type;
/* создать дек: */
int Dq_Init(size_t maxsize);
/* закончить работу: */
void Dq_Term(void);
/* добавить элемент val в начало: */
int Dq_PushHead(Type val);
/*     или конец дека: */
int Dq_PushTail(Type val);
/* взять начало или конец дека
   и поместить по указателю dst: */
int Dq_PopHead(Type *dst);
int Dq_PopTail(Type *dst);
/* вернуть указатель на начало: */
Type *Dq_Head(void);
/*     или конец дека: */
```

```
Type *Dq_Tail(void);
/* текущее количество элементов: */
size_t Dq_Size(void);
/* количество свободных мест: */
size_t Dq_Room(void);
```

Реализуйте на языке C++ дек в виде следующего класса:

```
/*---   Файл deque.h   ---*/
template <class Type>
class Deque {
 private:
  Type *begmem, *endmem;
  Type *head, *tail;
  size_t size;
  Type *Next(Type *pos);
  Type *Prev(Type *pos);

 public:
  Deque(size_t maxsize);
  ~Deque();
  int Success();
  int PushHead(Type val);
  int PushTail(Type val);
  int PopHead(Type &dst);
  int PopTail(Type &dst);
  Type &Head();
  Type &Tail();
  size_t Size();
  size_t Room();
};
```

По поводу функции `Success()` см. замечание к задаче 3-1-1 о реализации стека.

Задача 3-1-19. Постройте реализацию дека элементов абстрактного типа `Type` на основе структуры

```
struct Deque {
  /* начало области элементов дека: */
  Type* mem;
  /* размер дека: */
  size_t maxsize;
```

```
  /* текущее количество элементов: */
   size_t size;
 /* позиция для очередного головного элемента: */
   size_t front;
 /* позиция для очередного хвостового элемента:*/
   size_t back;
};
```

Идеи реализации. Считая, что `maxsize > 0`, будем размещать элементы дека в выделенном участке памяти, `mem[0],...,mem[maxsize-1]`, и хранить два целых числа `left` и `right` из диапазона от 0 до `maxsize - 1`, содержащие номера ячеек, куда будут добавляться очередные элементы (при наличии свободного места). В начальный момент

```
left = 0;
right = (left + 1) % maxsize;
```

При добавлении слева имеем

```
if (size == maxsize) return -1;
mem[left] = val;
size++;
left--;
if (left < 0) left = maxsize - 1;
```

Добавление справа осуществляется аналогично:

```
if (size == maxsize) return -1;
mem[right] = val;
size++;
right = (right + 1) % maxsize;
```

Задача 3-1-20. Выпуклой оболочкой L_N множества точек называется наименьший выпуклый многоугольник с вершинами в некоторых точках этого множества, содержащий внутри себя все остальные точки множества. Требуется построить выпуклую оболочку заданного множества точек плоскости (т. е. указать точки, являющиеся вершинами многоугольника).

Идеи реализации. Занумеруем числами от 1 до N точки множества в произвольном порядке. Отметим, что линейной оболочкой произвольного набора точек, лежащих на одной прямой является отрезок, задаваемый парой граничных точек. Далее будем считать, что линейная оболочка L_n для первых n точек построена и отлична от отрезка. Возьмем $(n+1)$-ю точку

и проверим, принадлежит ли она L_n. Для этого будем последовательно перебирать ребра L_n и проверять, в какой полуплоскости относительно прямой, содержащей текущее ребро, лежит рассматриваемая точка множества. Если точка находится внутри L_n, то $L_{n+1} = L_n$. В противном случае ее необходимо включить в выпуклую оболочку L_{n+1}, т. е. удалить некоторые ребра из L_n и добавить два новых ребра с общей вершиной в этой точке. Для хранения множества вершин выпуклой оболочки выделим дек. Точки, лежащие в начале и конце дека, будут определять текущее ребро выпуклой оболочки; перебор ребер осуществляется перекладыванием элементов дека, например из начала в конец.

Очередь. Для дисциплины работы с очередью используется сокращение FIFO (First In, First Out), т. е. элемент, первым добавленный в очередь (в ее конец), будет также первым взят из (начала) очереди. Как нетрудно видеть, очередь является частным случаем дека, если запретить доступ к элементу, находящемуся в конце (и его удаление), а также добавление элемента в начало. Доступ к значению хранимых данных разрешается только для элемента, расположенного в начале очереди.

Задача 3-1-21. Реализуйте на языке C очередь элементов абстрактного типа `Type` со следующим интерфейсом:

```
/*---   Файл queue.h   ---*/
typedef ..... Type;
/* создать очередь: */
int Qu_Init(size_t maxsize);
/* закончить работу: */
void Qu_Term(void);
/* добавить элемент в очередь: */
int Qu_Add(Type val);
/* взять элемент из очереди: */
int Qu_Take(Type *dst);
/* указатель на начало очереди: */
Type *Qu_Head(void);
/* текущее количество элементов: */
size_t Qu_Size(void);
/* количество свободных мест: */
size_t Qu_Room(void);
```

Задача 3-1-22. Реализуйте на языке C++ очередь в виде следующего класса:

```
/*---   Файл queue.h   ---*/
template <class Type>
class Queue {
 private:
  Type *begmem, *endmem;
  Type *head, *tail;
  int size;
  Type *Next(Type *pos);

 public:
  Queue(int maxsize);
  ~Queue();
  int Success();
  int Add(Type val);
  int Take(Type &dst);
  Type &Head();
  int Size();
  int Room();
};
```

Задача 3-1-23. Считая, что время добавления и изъятия элементов из очереди — случайные величины, смоделируйте, как со временем изменяется длина очереди.

— Хочешь анекдот?
— Опять старье про Винни-Пуха, Пятачка и Иа-Иа?
— Да нет же, свежий! Значит так, идут три программиста по пустыне: Винни, Пятачок и Иа-Иа...

3.2. Односвязные и двусвязные списки

Список можно представить как структуру данных, элементы которой связаны в линейную цепочку и вместе с каждым элементом данных хранится адрес места размещения соседнего с ним элемента. Различают односвязный (однонаправленный) список, когда в узлах хранят некоторое содержимое и адрес только следующего элемента, и двусвязный (двунаправленный) список, когда помимо содержимого хранятся адреса следующего и предыдущего элементов списка.

Задача 3-2-1. Пусть имеется файл, содержащий целочисленную последовательность заранее не известной длины. Требуется за один просмотр файла сохранить все элементы в памяти компьютера, а затем распечатать результат на экране.

Идеи реализации. Для хранения элементов будем использовать однонаправленный список, т. е. цепочку структур следующего типа:

```
struct ListItem {
  int val;
  struct ListItem *next;
};
```

В поле `val` первой структуры цепочки сохраним первый элемент последовательности, в поле `next` — указатель на структуру, содержащую второй элемент, и так далее. Тогда в последней структуре цепочки в поле `val` будет записан последний элемент последовательности, а поле `next` положим равным `NULL`. В этом случае решение задачи может иметь, например, следующий вид.

```
struct ListItem *begin = NULL, *end = NULL, *cur;
int v;
if (fscanf(in, "%d", &v) != 1) return -1;
cur = (struct ListItem *)malloc(
                          sizeof(struct ListItem));
cur->val = v;
cur->next = NULL;
begin = end = cur;
while (fscanf(in, "%d", &v) == 1) {
  cur = (struct ListItem *)malloc(
                          sizeof(struct ListItem));
  cur->val = v;
  cur->next = NULL;
  end->next = cur;
  end = cur;
}
cur = begin;
printf("< ");
while (cur != NULL) {
  printf("%d ", cur->val);
  cur = cur->next;
```

```
}
printf(">\n");
```

Задача 3-2-2. Реализуйте решение задачи 3-2-1 на основе однонаправленного списка, добавляя на каждом шаге очередной элемент последовательности перед текущим, т. е. ячейка `begin->val` должна содержать последний считанный элемент.

Задача 3-2-3. Постройте для задачи 3-2-1 рекурсивную реализацию процедуры печати.

Идеи реализации.

```
void prn(struct ListItem *cur) {
  static int num = 0;
  num++;
  if (num == 1) {
    printf("\n<");
  }
  if (cur != NULL) {
    printf(" %d ", cur->val);
    cur = cur->next;
    prn(cur);
  } else {
    num = 0;
    printf(">\n");
  }
  return;
}
```

Отметим, что рекурсивные алгоритмы для работы с одно- и двунаправленными списками, в отличие от деревьев (см. раздел 3.3), обычно не используются.

Задача 3-2-4. Постройте реализацию стека элементов абстрактного типа `Type` на базе однонаправленного списка.

Идеи реализации. Для описания элемента списка будем использовать следующий тип данных:

```
typedef struct ListItem_L1 {
  Type value;
  struct ListItem_L1 *next;
} ListItem;
```

Здесь `next` является указателем на следующий элемент по отношению к данному. Указатель на начало списка (на

вершину стека) будем хранить в переменной `ListItem *head`. Зафиксируем для работы со стеком следующий интерфейс:

```
/* добавить элемент val: */
int StL1_Push(ListItem **phead, Type val);
/* удалить вершину и поместить ее
   значение по указателю dst: */
int StL1_Pop(ListItem **phead, Type *dst);
/* удалить вершину: */
int StL1_Del(ListItem **phead);
/* распечатать содержимое стека: */
void StL1_Prn(ListItem **phead);
/* закончить работу: */
void StL1_Term(ListItem **phead);
```

и приведем для примера реализации нескольких функций:

```
int StL1_Push(ListItem **phead, Type val) {
  ListItem *cur = (ListItem *)malloc(
                                sizeof(ListItem));
  if (cur == NULL) return -1;
  cur->value = val;
  cur->next = *phead;
  (*phead) = cur;
  return 0;
}
void StL1_Term(ListItem **phead) {
  ListItem *cur;
  while ((*phead) != NULL) {
    cur = *phead;
    *phead = (*phead)->next;
    free(cur);
  }
  return;
}
```

При работе со списком в каждый момент времени определен один текущий элемент, с которым допустимо выполнять какие-либо действия (возможны реализации, при которых доступ разрешен также к непосредственным соседями этого элемента). Положение текущего элемента определяется соответствующим указателем списка. Указатель можно перемещать по направ-

лению ссылок и тем самым получать доступ к остальным элементам. Добавление и удаление элементов происходит только в окрестности текущего положения указателя списка, при этом новые добавляемые элементы «раздвигают» список. Можно по-разному определять, какой элемент станет текущим в результате добавления или удаления. Например, можно считать, что при добавлении текущим останется старый текущий элемент, а при удалении новым текущим будет (по возможности) следующий элемент.

Задача 3-2-5. Постройте реализации однонаправленного списка элементов абстрактного типа `Type` на языках C и C++.

Идеи реализации. Указатель на начало списка будем хранить в переменной `head`, текущую позицию в списке задавать указателем `current`, а в переменной `num_elem` сохраним число элементов списка. Наряду с `current` будем хранить указатель на предыдущий элемент `before`, что существенно упростит процедуру удаления. Считая, что указанные переменные содержатся в отдельной структуре

```
typedef struct List_L1 {
  ListItem *current;
  ListItem *before;
  ListItem *head;
  unsigned int num_elem;
} List;
List list;
```

зафиксируем для работы со списком следующий интерфейс:

```
/* создать (пустой) список: */
int L1_Init(List *plist);
/* освободить память и удалить список: */
int L1_Term(List *plist);
/* список пустой?: */
int L1_Empty(List *plist);
/* добавить элемент до текущего: */
int L1_AddBefore(List *plist, Type val);
/* добавить элемент за текущим: */
int L1_AddAfter(List *plist, Type val);
/* удалить текущий элемент: */
int L1_DelCurrent(List *plist);
```

```
/* перейти к следующему элементу: */
int L1_ToNext(List *plist);
/* перейти к предыдущему элементу: */
int L1_ToPrev(List *plist);
/* перейти к началу списка: */
int L1_ToHead(List *plist);
/* перейти к концу списка:  */
int L1_ToTail(List *plist);
```

и приведем для примера реализацию нескольких функций. Будем считать, что если `current` — это `head`, то `before` — это `NULL`, а случай, когда `current` — это `NULL`, соответствует пустому списку.

```
int L1_Init(List *plist) {
  plist->head = plist->current =
                        plist->before = NULL;
  plist->num_elem = 0;
  return 0;
}
int L1_Empty(List *plist) {
  if (plist->num_elem == 0) return 1;
  return 0;
}
int L1_ToHead(List *plist) {
  if (L1_Empty(plist)) return -1;
  plist->current = plist->head;
  plist->before = NULL;
  return 0;
}
int L1_ToTail(List *plist) {
  if (L1_Empty(plist)) return -1;
  while (plist->current->next != NULL) {
    plist->before = plist->current;
    plist->current = plist->current->next;
  }
  return 0;
}
int L1_AddAfter(List *plist, Type val) {
  ListItem *pos;
  pos = (ListItem *)malloc(sizeof(ListItem));
```

```
  if (pos == NULL) return -1;
  pos->value = val;
  pos->next = NULL;
  if (plist->current == NULL) {
    plist->head = plist->current = pos;
  } else {
    pos->next = plist->current->next;
    plist->current->next = pos;
  }
  plist->num_elem++;
  return 0;
}
int L1_DelCurrent(List *plist) {
  if (plist->num_elem == 0) return -1;
  if (plist->num_elem == 1) {
    free(plist->current);
    plist->current = plist->before = NULL;
    plist->head = NULL;
    plist->num_elem = 0;
    return 0;
  }
  if (plist->current->next == NULL){
    L1_ToPrev(plist);
    free(plist->current->next);
    plist->current->next = NULL;
    plist->num_elem--;
  return 0;
  }
  else{
    ListItem *pos = plist->current;
    plist->current = plist->current->next;
    free(pos);
    if (plist->before != NULL)
      plist->before->next = plist->current;
    plist->num_elem--;
    return 0;
  }
  return 0;
}
int L1_ToNext(List *plist) {
```

```
  if (plist->num_elem == 0) return -1;
  if (plist->current->next == NULL) return -1;
  plist->before = plist->current;
  plist->current = plist->current->next;
  return 0;
}
```

Задача 3-2-6. Для удобства работы со списком можно ввести дополнительный элемент `ListItem base`: его поле `value` всегда считается пустым, а поле `next` либо указывает на голову, либо равно `NULL` для пустого списка. Постройте соответствующую реализацию однонаправленного списка элементов абстрактного типа `Type` на языке C.

Задача 3-2-7. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` на языке C.

Идеи реализации. Для описания элемента списка будем использовать следующий тип данных:

```
typedef struct ListItem_L2 {
  Type value;
  struct ListItem_L2 *next, *prev;
} ListItem;
```

Здесь `next` и `prev` являются соответственно указателями на следующий и предыдущий элементы по отношению к данному.

Введем дополнительный элемент `ListItem base` и свяжем весь список в кольцо через этот элемент. Таким образом, начальный и конечный элементы списка будут ссылаться на адрес элемента `base`. Текущую позицию в списке будем задавать указателем на текущий элемент `ListItem *current`.

Считая (для прозрачности программного кода), что все переменные с параметрами списка хранятся на внешнем уровне, примем для работы со списком следующий интерфейс:

```
/*---   Файл  list2.h   ---*/
typedef struct ListItem_L2 {
  Type value;
  struct ListItem_L2 *next, *prev;
} ListItem;
/* создать (пустой) список: */
int L2_Init(void);
/* закончить работу: */
```

```
void L2_Term(void);
/* добавить элемент до текущего: */
int L2_AddBefore(Type val);
/* добавить элемент за текущим: */
int L2_AddAfter(Type val);
/* удалить текущий элемент: */
int L2_DelCurrent(void);
/* перейти к следующему элементу: */
int L2_ToNext(void);
/* перейти к предыдущему элементу: */
int L2_ToPrev(void);
/* перейти к началу списка: */
int L2_ToHead(void);
/* перейти к концу списка: */
int L2_ToTail(void);
/* позиция в начале списка?: */
int L2_AtHead(void);
/* позиция в конце списка?: */
int L2_AtTail(void);
/* доступ к текущему элементу: */
Type *L2_Current(void);
/* количество элементов в списке: */
int L2_NumElem(void);
/* список пуст?: */
int L2_Empty(void);
```

Функция `L2_NumElem` возвращает количество элементов в списке; остальные функции со значением типа `int` возвращают 0 в случае успеха и -1 в случае отказа.

Решение. Приведем в качестве примера код нескольких функций, остальные реализуйте самостоятельно.

```
#include <stdlib.h>
#include "list2.h"
static ListItem base;
static ListItem *current = NULL;
static int num_elem = 0;
int L2_Init(void) {
  current = &base;
  base.next = base.prev = &base;
  num_elem = 0;
```

```
  return 0;
}
void L2_Term(void) {
  for (L2_ToHead(); L2_NumElem(); L2_DelCurrent());
}
int L2_AddBefore(Type val) {
  ListItem *pos;
  pos = (ListItem *)malloc(sizeof(ListItem));
  if (!pos) return -1;
  pos->prev = current->prev;
  pos->next = current;
  current->prev->next = pos;
  current->prev = pos;
  pos->value = val;
  num_elem++;
  return 0;
}
int L2_DelCurrent(void) {
  ListItem *pos;
  if (L2_Empty()) return -1;
  current->prev->next = current->next;
  current->next->prev = current->prev;
  pos = current;
  if (current->next == &base)
    current = current->prev;
  else
    current = current->next;
  num_elem--;
  free(pos);
  return 0;
}
int L2_ToNext(void) {
  if (L2_AtTail()) return -1;
  current = current->next;
  return 0;
}
int L2_AtHead(void) {
  return (current == base.next);
}
Type *L2_Current(void) {
```

```
  return &(current->value);
}
int L2_NumElem(void) { return num_elem; }
```

Отметим, что обращение к большинству из указанных функций до начала инициализации списка приведет к аварийному останову. Уберите данное ограничение.

Задача 3-2-8. Добавьте к реализации списка из предыдущей задачи функцию последовательного поиска элемента с заданным значением поля `value` (текущая позиция `current` при успехе устанавливается на найденный элемент, при неудаче не изменяется) и функцию, применяющую заданную процедуру к каждому элементу списка (текущая позиция не изменяется). Заголовки этих функций могут иметь вид

```
int L2_Search(Type val);
void L2_Process(void (*action)(Type));
```

Функция поиска должна возвращать 0, если элемент найден, и -1 при отсутствии элемента в списке.

Задача 3-2-9. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` на языке C++.

Задача 3-2-10. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` без использования элемента `base`.

Идеи реализации. Позиции начального, конечного и текущего элементов будем хранить в указателях `ListItem *header`, `ListItem *back`, `ListItem *list` соответственно. Для пустого списка значения этих указателей равны `NULL`. В процедурах добавления и удаления теперь появятся дополнительные проверки, необходимые для корректной работы, когда текущая позиция находится в начале или конце списка. Будем использовать следующий интерфейс:

```
/* указатели на начало и конец списка,
NULL соответствует пустому списку: */
ListItem *header = NULL;
ListItem *back = NULL;
/* принимает указатель на элемент,
перед которым нужно вставить ячейку
со значением value, возвращает указатель на
созданную ячейку, которая становится текущей: */
```

```
ListItem *L2_AddBefore(ListItem *list, Type value);
/* возвращает указатель либо
на следующую за list ячейку, либо NULL, если list
является  последней (концевой) ячейкой: */
ListItem *L2_Next(ListItem *list);
/* возвращает указатель либо
на предыдущую для list ячейку, либо NULL,
если list является первой ячейкой: */
ListItem *L2_Prev(ListItem *list);
/* возвращает указатель
на первую ячейку списка: */
ListItem *L2_Front(ListItem *list);
/* возвращает указатель
на последнюю  ячейку списка: */
ListItem *L2_Back(ListItem *list);
/* удаляет указанную ячейку из списка
и возвращает либо указатель на следующую
ячейку (если она есть), либо на предыдущую,
если list была последней (концевой): */
ListItem *L2_Del(ListItem *list);
```

В качестве примера приведем реализацию двух функций.

```
ListItem *L2_AddBefore(ListItem *list,
                                    Type value) {
  ListItem *newElem;
  if (list == NULL) {
    newElem =
            (ListItem *)malloc(sizeof(ListItem));
    if (newElem == NULL) return NULL;
    newElem->next = newElem->prev = NULL;
    newElem->value = value;
    header = back = newElem;
    return newElem;
  }
  newElem = (ListItem *)malloc(sizeof(ListItem));
  if (newElem == NULL) return list;
  newElem->value = value;
  newElem->next = list;
  newElem->prev = list->prev;
  list->prev = newElem;
```

```
  if (newElem->prev)
    newElem->prev->next = newElem;
  else
    header = newElem;
  return newElem;
}
ListItem *L2_Del(ListItem *list) {
  if (!list) return NULL;
  if (!(list->next) && !(list->prev)) {
    header = back = NULL;
      free(list);
      return NULL;
  }
  if (list->next) {
    ListItem *next = list->next;
    list->next->prev = list->prev;
    if (list->prev)
      list->prev->next = list->next;
    else
      header = next;
    free(list);
    return next;
  } else {
    list->prev->next = list->next;
      back = list->prev;
      free(list);
      return back;
  }
}
```

Отметим, что в некоторых случаях весь список бывает удобно замкнуть в кольцо (т. е. связать ссылками первый и последний элементы).

Задача 3-2-11. Добавьте к реализациям списков из задач 3-2-7—3-2-10 процедуры упорядочивания списка по некоторому критерию. Заголовок соответствующей функции может иметь вид

```
void Sort(int (*compare)(Type a, Type b));
```

где `compare` — указатель на функцию сравнения значений двух элементов списка. Естественно, при упорядочивании нужно ограничиться только перестройкой взаимных ссылок, не пере-

мещая в памяти сами элементы списка. Реализуйте на списках следующие алгоритмы сортировки (см. задачи 1-4-9—1-4-17):

1) сортировка выбором максимального элемента (сложность алгоритма $O(n^2)$ переходов, $O(n^2)$ сравнений);
2) пузырьковая сортировка (сложность алгоритма $O(n^2)$ переходов, $O(n^2)$ сравнений);
3) сортировка просеиванием;
4) вставка в упорядоченную часть списка с последовательным поиском;
5) слияние подсписков (алгоритм Неймана);
6) быстрая сортировка (алгоритм quicksort) для двунаправленного списка (сложность алгоритма $O(n^2)$ переходов, $O(n \log_2 n)$ сравнений).

Для каждого алгоритма оцените требуемое количество сравнений и переходов. Какие из этих алгоритмов могут быть эффективно реализованы на однонаправленных списках?

Задача 3-2-12. На основе списочной реализации постройте полный словарь встречающихся в заданном файле слов и для каждого слова подсчитайте частоту вхождения. На примере нескольких произведений известного автора найдите его любимые слова.

Задача 3-2-13. Постройте реализацию массива списков на языке C. Каждая функция работы со списком при этом получит дополнительный параметр — номер списка. Процедура инициализации получает в качестве параметра начальное количество списков. Прототипы функций могут иметь следующий вид (ср. с задачей 3-2-7):

```
/* создать k списков: */
int L_Init(int k);
/* добавить в k-й список: */
int L_AddAfter(int k, Type val);
```

Реализуйте задачу 3-2-12 на основе массива списков, собирая в отдельные списки слова с одинаковой начальной буквой.

Задача 3-2-14. Матрица, в которой число ненулевых элементов значительно меньше общего количества элементов, называется разреженной. Для компактного представления разреженной матрицы, имеющей m строк, удобно использовать массив

из m списков: в i-м списке как пары (j, a_{ij}) последовательно хранятся все ненулевые значения i-й строки.

На основе решения предыдущей задачи реализуйте на языке C процедуры работы с разреженной матрицей со следующим интерфейсом:

```
/* создать m*n матрицу: */
int CreateMatr(int m, int n);
/* добавить элемент a(i,j): */
int Put(double a, int i, int j);
/* прочитать элемент a(i,j): */
double Get(int i, int j);
```

Обратите внимание, что нулевые элементы не должны храниться в матрице, т. е. при записи нулевого (либо близкого к нулю) значения в существующий элемент списка он должен из списка удаляться.

Задача 3-2-15. Добавьте к реализации разреженной матрицы из предыдущей задачи функции, выполняющие линейные комбинации и перестановки строк (и столбцов). Используя построенные функции, напишите программу решения системы линейных уравнений с разреженной матрицей методом Гаусса.

Задача 3-2-16. Реализуйте на языке C++ разреженную матрицу со следующим интерфейсом:

```
class sparse_matrix {
  private:
    .....
  public:
    /* m - число строк, n - число столбцов */
    sparse_matrix(int m, int n);
    ~sparse_matrix();
    /* чтение */
    double operator()(int i, int j) const;
    /* запись */
    double& operator()(int i, int j);
}
```

Задача 3-2-17. Добавьте в класс `sparse_matrix` из предыдущей задачи переопределения арифметических операций с матрицами (сложения, вычитания, умножения).

Задача 3-2-18. На основе реализации разреженной матрицы постройте электронную таблицу, поддерживающую автоматическую модификацию значений некоторых элементов матрицы при изменении значений других элементов. Простейшим примером может служить числовая матрица, в которой последний столбец содержит суммы элементов соответствующих строк, а последняя строка содержит суммы элементов из соответствующих столбцов. При изменении элементов матрицы эти суммы соответствующим образом модифицируются. Для повышения универсальности реализации можно продумать способы предоставления пользователю возможности менять алгоритмы пересчета элементов матрицы.

Учительница в первом классе.
— Дети, бородавочник, он какой?
— Бородавочник — Бдительный! Ежик — Седой, Барсук — Бодрый, Селезень — Сметливый, Тритон — Торопливый, Олененок — Отважный, Гиббон — Геройский,..., Гиппопотам — Косматый, Индри — Озорной, Медуза — Везучая, а Куда будет — Кинетический!!!

3.3. Деревья

Прежде чем формулировать задачи данного раздела, напомним необходимые понятия. Дерево — это иерархическая структура данных, состоящая из вершин и ребер. Одна (выделенная) вершина называется корнем дерева. Связи между вершинами (задаваемые ребрами) описываются в терминах *родитель — его непосредственные* (*дочерние*) *потомки*. Каждая вершина дерева, кроме корня, связана с одной и только одной родительской вершиной и с некоторым (возможно, нулевым) числом дочерних элементов. Соответственно для своих дочерних элементов вершина является родительской. Корень не имеет родительской вершины. Формально дерево является плоским графом без циклов и с одной выделенной (корневой) вершиной.

Если каждая вершина дерева имеет не более двух дочерних элементов, то дерево называется бинарным, в противном случае — произвольным (иногда используется термин *сильно ветвящееся дерево*). Для бинарного дерева естественным

образом вводятся понятия левого и правого поддеревьев. Пример бинарного дерева представлен на следующей схеме.

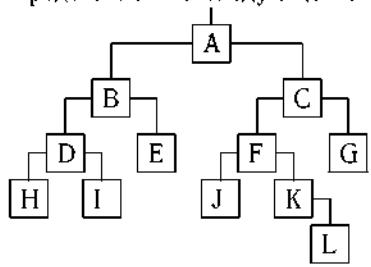


Рис. 3.3.1. Пример бинарного дерева

Будем говорить, что корень составляет нулевой уровень дерева. Непосредственные потомки корня составляют первый уровень дерева. Непосредственные потомки вершин k-го уровня образуют $(k+1)$-й уровень дерева.

Вершина дерева является концевой (или листом), если она не имеет потомков. Для выбранной вершины каждый связный участок от нее до произвольной вершины, имеющей не более одного потомка, называется ветвью. Длина ветви равна количеству вершин в соответствующем связном участке. Длиной (глубиной) дерева называется длина максимальной ветви в этом дереве. Дерево считается идеально сбалансированным, если длины двух любых его ветвей, исходящих из корня, отличаются не более чем на единицу. Как следствие, в идеально сбалансированном дереве для каждой его вершины количество вершин в левом и правом поддеревьях отличается не более чем на единицу.

Бинарное дерево называется сбалансированным (или АВЛ-деревом), если для любой его вершины длины левого и правого поддеревьев отличаются не более чем на единицу.

Существует целый класс прикладных задач, где требуется обеспечить эффективный поиск и размещение данных, для которых задано некоторое отношение порядка. Простейшим примером может служить набор чисел с естественным отношением «меньше». Если мы рассмотрим упорядоченный числовой массив, то процедура поиска конкретного значения в таком массиве может быть реализована методом деления пополам с вычислительной сложностью (т. е. числом арифметических операций, необходимых для выполнения работы) порядка $O(\log_2 N)$, где N — количество элементов в массиве. Однако

процедура вставки элемента в упорядоченный массив требует уже O(N) операций, так как при этом нужно «раздвинуть» элементы массива, чтобы освободить место для нового элемента. Добавление элемента в неупорядоченный массив выполняется всего за O(1) действий (элемент просто приписывается в конец массива), но поиск уже приходится выполнять последовательным просмотром, что требует O(N) действий. Эти две крайние точки в оценках трудоемкости добавления и поиска вызывают желание построить реализацию хранения и поиска набора данных с промежуточными характеристиками трудоемкости. Идеальным средством для этого служат бинарные деревья поиска.

Пусть на данных, хранящихся в вершинах дерева, задано отношение порядка (т. е. их можно сравнивать между собой на больше-меньше). Бинарное дерево называется упорядоченным (или деревом поиска), если для любой вершины все элементы левого поддерева меньше элемента, расположенного в этой вершине, а все элементы правого поддерева больше элемента из этой вершины либо равны ему.

При работе с упорядоченными деревьями можно выделить две процедуры.

- Формирование дерева по указанным правилам. Важным частным случаем здесь является работа с обычным или сбалансированным деревом поиска с возможностью добавления, поиска и удаления заданного элемента.
- Обход дерева: требуется выполнить некоторую заданную операцию для каждой вершины дерева. Для бинарных деревьев можно выделить три способа обхода: *сверху вниз* (сначала обрабатывается текущая вершина, а затем ее левое и правое поддеревья), *снизу вверх* (сначала обрабатываются поддеревья, а затем текущая вершина) и *слева направо* (сначала обрабатывается левое поддерево, затем вершина, затем правое поддерево). Процедуры обходов можно обобщить на случай произвольных деревьев.

3.3.1. Бинарные упорядоченные деревья

Для реализации бинарных деревьев (например, с элементами целого типа) удобно использовать структуру

```
struct TreeNode {
  int value;
```

```
  struct TreeNode *left, *right;
};
```

Если в алгоритме необходимо явное перемещение по направлению к корню, то в эту структуру можно добавить ссылку на родительскую вершину `struct TreeNode *parent`. В рекурсивных процедурах обработки дерева подобное движение обычно реализуется автоматически за счет возврата из последовательности рекурсивных вызовов, поэтому при использовании таких алгоритмов указатель `parent` не нужен.

Добавление элемента в упорядоченное дерево

Здесь следует сделать одно замечание. Мы можем подходить к формированию дерева с двух различных позиций, а именно дублировать или не дублировать элементы с одинаковыми ключами. В первом случае процедура добавления будет всегда включать в дерево новый элемент, во втором случае при обнаружении вершины с тем же ключом процедура будет просто завершаться (раз такой элемент уже есть, еще раз его добавлять не надо). Наш пример будет относиться к первому варианту.

Задача 3-3-1. Постройте нерекурсивную процедуру добавления элемента в бинарное дерево поиска.

Решение.

```
struct TreeNode *Tree_Add(int x,
                          struct TreeNode *root) {
  struct TreeNode *cur, *node;
  node = Tree_CreateNode();
  if (node == NULL) return NULL;
  node->value = x;
  node->right = node->left = NULL;
  if (root == NULL) return node;

  cur = root;
  while (1) {
    if (x >= cur->value) {
      if (cur->right == NULL) {
        cur->right = node;
        return root;
      } else {
        cur = cur->right;
```

```
      }
    } else {
      if (cur->left == NULL) {
        cur->left = node;
        return root;
      } else {
        cur = cur->left;
      }
    }
  }
  return root;
}
struct TreeNode *Tree_CreateNode(void) {
  return (struct TreeNode *)malloc(
                  sizeof(struct TreeNode));
}
void Tree_FreeNode(struct TreeNode *root) {
  if (root) free(root);
}
```

Отметим, что приведенные функции будут использоваться при решении последующих задач.

Задача 3-3-2. Измените предложенную реализацию, объединив ветви алгоритма, отвечающие за «подклеивание» нового узла.

Указание. Возьмите за основу конструкцию следующего типа:

```
struct TreeNode **pnode = &root;
/* здесь выделяем память под node, инициируем
величиной x и нулевыми ссылками на потомков */
cur = root;
while (cur != NULL) {
  if (x >= cur->value) {
    pnode = &(cur->right);
    cur = cur->right;
  } else {
    pnode = &(cur->left);
    cur = cur->left;
  }
}
*pnode = node;
return root;
```

В данном случае переменная **pnode** хранит адрес ячейки, куда по завершении цикла запишется адрес созданного узла.

Задача 3-3-3. Постройте рекурсивную процедуру добавления элемента в бинарное дерево поиска.

Идеи реализации. Начиная с корня, определяем, в какое поддерево должен попасть данный элемент, и рекурсивно вызываем процедуру добавления для требуемого поддерева. Возвращаемое значение функции — указатель на вершину текущего уровня. Если добавить элемент не удалось, то на соответствующем уровне возвращаем **NULL**.

Решение. С учетом функций из задачи 3-3-1 имеем

```
struct TreeNode *Tree_Add(int x,
                          struct TreeNode *root) {
  if (!root) {
    if (!(root = Tree_CreateNode())) return NULL;
    root->value = x;
    root->right = root->left = NULL;
    return root;
  } else {
    if (x >= root->value) {
      root->right = Tree_Add(x, root->right);
    } else {
      root->left = Tree_Add(x, root->left);
    }
    return root;
  }
}
/* Вызов из main:
struct TreeNode *root = NULL;
while(fscanf(in, "%d", &x) == 1)
  root = Tree_Add(x, root); */
```

Задача 3-3-4. Модифицируйте рекурсивную процедуру добавления элемента в бинарное дерево поиска из задачи 3-3-3 так, чтобы функция возвращала указатель на корень всего дерева в случае успеха либо **NULL**, если добавить элемент не удалось.

Решение. С учетом функций из задачи 3-3-1 имеем

```
TreeNode *Tree_AddElement(Type x,
                          TreeNode *root) {
```

```
  TreeNode *cur;
  if (!root) {  // т. е. пустое дерево
    root = Tree_CreateNode();
    if (root) {
      root->value = x;
      root->left = root->right = NULL;
    }
  } else {  // непустое дерево -> рекурсия
    if (x < root->value) {
      cur = Tree_AddElement(x, root->left);
      if (cur)
        root->left = cur;
      else
        return NULL;
    } else {
      cur = Tree_AddElement(x, root->right);
      if (cur)
        root->right = cur;
      else
        return NULL;
    }
  }
  return root;
}
/* Вызов из main:
struct TreeNode *root=NULL, *cur;
while(fscanf(in, "%d", &x) == 1){
  if (cur = Tree_Add(x, root)) root = cur;
  else printf("Элемент добавить не удалось!\n");
} */
```

Печать элементов бинарного дерева

Задача 3-3-5. Реализуйте три упомянутые выше процедуры обхода и печати элементов бинарного дерева, предполагая, что в вершинах хранятся целые числа (Type есть int).

Решение. Пусть корень указанного дерева определяется указателем root. Тогда процедуры могут иметь следующий вид.

```
void Tree_PrnUpDown(struct TreeNode *root) {
  if (!root) return;
```

```
  printf("%d ", root->val);
  Tree_PrnUpDown(root->left);
  Tree_PrnUpDown(root->right);
  return;
}
void Tree_PrnDownUp(struct TreeNode *root) {
  if (!root) return;
  Tree_PrnDownUp(root->left);
  Tree_PrnDownUp(root->right);
  printf("%d ", root->val);
  return;
}
void Tree_PrnLeftRight(struct TreeNode *root) {
  if (!root) return;
  Tree_PrnLeftRight(root->left);
  printf("%d ", root->val);
  Tree_PrnLeftRight(root->right);
  return;
}
```

Каждая из этих функций обхода перебирает вершины дерева по-своему. Например, для дерева, изображенного на рис. 3.3.1, последовательность обработки вершин будет следующей:

для обхода сверху вниз:	A B D H I E C F J K L G;
для обхода снизу вверх:	H I D E B J L K F G C A;
для обхода слева направо:	H D I B E A J F K L C G.

Разберите работу программ, считая, что дерево поиска было построено по входной последовательности 8, 15, 5, 6, 2, 3, 7, 9, 13, 26, 23.

Отметим, что в данных функциях не используется ссылка на родительский элемент (`parent`). Действительно, ссылка нужна для перемещения по дереву по направлению к корню. Однако в приведенных процедурах это перемещение происходит автоматически за счет запоминания текущего локального значения параметра `root` в системном стеке при организации последовательности рекурсивных вызовов. Таким образом, указатель `parent` не нужен, если обработка вершин дерева производится с помощью рекурсивных процедур. Включать или

нет этот указатель в описание структуры вершины дерева, зависит от решаемых задач и способов обработки вершин дерева.

Задача 3-3-6. Используйте три упомянутые выше процедуры обхода для подсчета суммы элементов бинарного дерева, предполагая, что в вершинах хранятся целые числа.

Указание. Будем считать, что функция получает указатель на вершину дерева и возвращает сумму его элементов. Тогда рекурсивная реализация функции, основанная на обходе сверху вниз, состоит, по сути, из одного оператора: к элементу, хранящемуся в вершине, прибавляем сумму элементов левого и правого поддеревьев и возвращаем найденный ответ. Глубина рекурсии ограничивается пустым деревом — сумма его элементов равна нулю.

```
int Tree_SumUpDown(struct TreeNode *root) {
  if (!root) return 0;
  return root->val + Tree_SumUpDown(root->left) +
         Tree_SumUpDown(root->right);
}
```

При работе с очень «длинными» деревьями из-за большой вложенности рекурсивных вызовов в процессе выполнения программы могут возникнуть определенные трудности: существенное увеличение времени работы, «падение» из-за нехватки памяти. Поэтому, несмотря на то что рекурсивные алгоритмы зачастую выглядят более элегантными и короткими, в ряде случаев более эффективными могут оказаться нерекурсивные алгоритмы обработки вершин дерева.

Поиск элементов в бинарном упорядоченном дереве

Задача 3-3-7. Постройте рекурсивную и нерекурсивную процедуры поиска элемента в бинарном упорядоченном дереве. Функция поиска должна возвращать указатель на найденный элемент или **NULL**, если такого элемента в дереве нет.

Решение. Реализуем рекурсивную процедуру поиска данного элемента в поддереве. В качестве параметров этой процедуры разумно выбрать указатель на стартовую вершину поиска и сам искомый элемент; возвращаемое значение — указатель на найденную вершину с требуемым ключом либо **NULL**, если такая вершина отсутствует в дереве.

```
struct TreeNode *Tree_RecurSearch(Type x,
                                  struct TreeNode *root) {
```

```
  if (!root) return NULL; // дерево пусто
  if (x == root->value) return root; // нашли
  if (x < root->value)
    return Tree_RecurSearch(x, root->left);
  else
    return Tree_RecurSearch(x, root->right);
}
```

Нерекурсивный вариант этой процедуры выглядит еще проще.

```
struct TreeNode *Tree_DirectSearch(Type x,
                        struct TreeNode *root) {
  while (root) {
    if (x == root->value) break;
    root = (x < root->value) ?
                      root->left : root->right;
  }
  return root;
}
```

Обход бинарного дерева и вычисление его характеристик

Задача 3-3-8. Постройте подходящие процедуры обхода бинарного (но не обязательно упорядоченного) дерева с целочисленными элементами для решения следующих задач.

1. Распечатайте все концевые элементы (листья).
2. Распечатайте элементы, лежащие на уровне k.
3. Модифицируйте функции печати из задачи 3-3-5 так, чтобы элементы k-го уровня располагались в k-м столбце.
4. Вычислите количество концевых элементов.
5. Найдите сумму концевых элементов.
6. Определите количество элементов, лежащих на уровне k.
7. Найдите максимальный элемент среди всех элементов k-го уровня.
8. Найдите сумму элементов, лежащих на уровне k.
9. Найдите поддерево, для которого сумма элементов совпадает с указанным числом.
10. Определите глубину дерева (т. е. длину максимальной ветви).
11. Определите количество ветвей, имеющих максимальную длину.

12. Проверьте, является ли бинарное дерево идеально сбалансированным.
13. Распечатайте ветвь дерева (исходящую из корня) с заданным значением листа.
14. Выведите наглядное представление дерева целых чисел в текстовой файл.
15. Определите длину связного пути между двумя вершинами с заданными значениями неупорядоченного дерева.

Указание. Для решения первого пункта достаточно реализовать некоторый обход дерева (см. задачу 3-3-5), добавив проверку условия

```
if ((root->left == NULL) && (root->right == NULL))
  printf("%d\n", root->val);
```

Для решения второй задачи потребуется изменить прототип функции обхода, добавив параметры, отвечающие за текущий уровень `level` и уровень печати k:

```
void Tree_PrnUpDownLevelk(struct TreeNode *root,
                          int level, const int k);
```

Для реализации функций подсчета числа листьев можно передавать из `main` адрес ячейки `int leaves;`, предназначенной для хранения искомого значения:

```
void Tree_FindNumLeaves(struct TreeNode *root,
                                    int *pleaves);
```

Это позволит в процессе обхода дерева при необходимости изменять содержимое самой переменной. Однако задачи обхода с подсчетом также полезно уметь реализовывать на основе идей динамического программирования. Будем считать, что функция с прототипом

```
int Tree_DynFindNumLeaves(struct TreeNode *root);
```

возвращает *количество листьев в поддереве, корень которого передан в качестве входного параметра*. С ее помощью для текущей вершины `root` легко найти количество листьев в ее левом и правом поддеревьях: `Tree_DynFindNumLeaves(root->left)`, `Tree_DynFindNumLeaves(root->right)`. При этом сумма этих величин дает число листьев для исходного поддерева

root. Таким образом, задача подсчета для root сведена к двум аналогичным задачам для поддеревьев меньшей длины root->left и root->right. Осталось отметить, что количество листьев в пустом дереве равно нулю, а в дереве из одной вершины — единице. Приведем полный вариант кода:

```
int Tree_DynFindNumLeaves(struct TreeNode *root) {
  if (root == NULL) return 0;
  if ((root->left == NULL) &&
                          (root->right == NULL))
    return 1;
  return Tree_DynFindNumLeaves(root->left) +
         Tree_DynFindNumLeaves(root->right);
}
```

Сравните результат с решением задачи 3-3-6.

Задача 3-3-9. Постройте подходящие процедуры обхода бинарного дерева с целочисленными элементами для решения следующих задач.

1. Проверьте, является ли бинарное дерево сбалансированным.
2. Подсчитайте число несбалансированных вершин в бинарном дереве.
3. Подсчитайте показатель сбалансированности для бинарного дерева (т. е. максимальную разницу между длинами правого и левого поддеревьев по всем вершинам).

Задача 3-3-10. Пусть построено бинарное дерево на основе структуры

```
struct TreeNode {
  Type value;
  struct TreeNode *left, *right;
  struct TreeNode *next;
};
```

Все указатели next равны NULL. Напишите функцию, получающую на вход указатель на корень дерева и связывающую вершины каждого уровня в однонаправленный список по указателю next.

Идеи реализации. Реализуйте рекурсивный алгоритм, настраивающий список $(k+1)$-го уровня при условии, что k-й уровень уже настроен.

Трудоемкость процедур поиска, добавления и удаления элемента в бинарном упорядоченном дереве зависит от длины максимальной ветви. Если дерево содержит N элементов, то эта величина равна N в наихудшем (вырожденном) случае и $\log_2 N$ в наилучшем случае. На данный момент разработаны алгоритмы, позволяющие организовать работу с деревом поиска (т. е. добавление и удаление элементов) так, чтобы оно всегда оставалось в некотором смысле сбалансированным: длина максимальной ветви не будет превосходить величины $1{,}5 \log_2 N$. Отметим, что придется ограничиться деревьями, все элементы которых различны. Дерево, содержащее только одинаковые элементы, не может быть одновременно упорядоченным и сбалансированным в смысле приведенных выше определений.

Для реализации сбалансированных деревьев потребуется модифицировать структуру вершины. Назовем балансом вершины разность между значениями длин его правого и левого поддеревьев. Будем хранить значение баланса каждой вершины вместе с остальной информацией о данной вершине и использовать следующее определение типа вершины:

```
typedef struct _TreeNode {
  Type value;
  int balance;
  struct _TreeNode *down, *next;
} TreeNode;
```

Для сбалансированного дерева значения поля `balance` в каждой вершине могут быть -1, 0, 1. Заметим, что если при добавлении элемента в некоторое поддерево его длина не возросла, то баланс вершины, родительской по отношению к этому поддереву, не изменился. Следовательно, балансировка (т. е. перестройка) дерева не нужна. Если же длина поддерева возросла, то, возможно, придется перестраивать дерево для восстановления утраченного баланса. Алгоритмы балансировки после добавления/удаления элементов приводятся в разделе 3.3.4.

Задача 3-3-11. Реализуйте функцию обхода бинарного дерева (не обязательно сбалансированного) с записью в поле `balance` значения баланса каждой вершины.

Задача 3-3-12. Считая, что бинарное дерево построено, а поле `balance` содержит значение баланса каждой вершины, реализуйте требуемые в задаче 3-3-9 функции.

Удаление элемента в бинарном упорядоченном дереве

Рассмотрим алгоритм удаления элемента в дереве поиска без учета баланса. Для удаления концевого элемента или элемента, имеющего только одного потомка, достаточно очевидным образом изменить ссылку от родительской вершины. В случае двух потомков удалить элемент так просто не удастся. Здесь уже имеются три ссылки (от родителя к вершине и от нее к двум потомкам), и их невозможно замкнуть друг на друга. Идея удаления состоит в том, что удаляемая вершина как структурный элемент дерева на самом деле остается на месте, но данные, хранящиеся в этой вершине, заменяются на другие. Поясним это на примере. Пусть мы хотим удалить элементы с ключами 5 и 12 из дерева, изображенного на рис. 3.3.2.

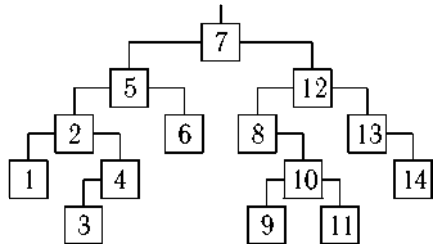


Рис. 3.3.2. Дерево до удаления элементов

Найдем в дереве вершины, содержимое которых можно подставить вместо удаляемых данных так, чтобы дерево сохранило свойство упорядоченности (осталось деревом поиска). В нашем примере этими вершинами, очевидно, будут вершины с ключами 4 (для 5) и 11 (для 12). Перенесем данные из этих вершин в «удаляемые» вершины (с ключами 5 и 12).

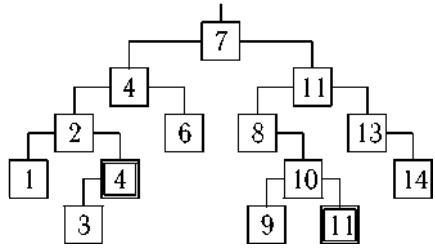


Рис. 3.3.3. Сделана
замена значений, должны быть удалены помеченные вершины

Теперь можно удалить вершины (отмеченные двойной рамкой), хранившие эти данные ранее, поскольку они имеют не более одного потомка.

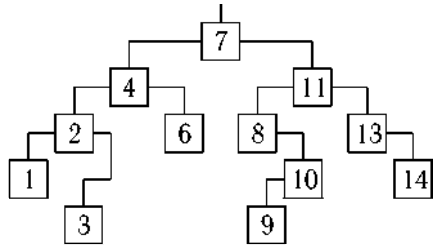


Рис. 3.3.4. Дерево после удаления помеченных вершин

Анализируя рассмотренный пример, мы приходим к следующему неформальному алгоритму удаления.

1. Если удаляемая вершина является концевой (не имеет потомков), то достаточно освободить память, занимаемую этой вершиной, и обнулить соответствующую ссылку от родительской вершины.
2. Если удаляемая вершина имеет ровно одного потомка, то после удаления вершины из памяти ссылка от родительской вершины переставляется на этого потомка.
3. В случае двух потомков сначала в левом поддереве удаляемой вершины ищется вершина с максимальным ключом. Заметим, что эта «максимальная» вершина обязательно является либо концевой, либо имеет только одного потомка. Для того чтобы ее найти, достаточно спускаться по самой правой ветви левого поддерева до тех пор, пока не обнаружится нулевая ссылка направо. Найдя нужную «максимальную» вершину, переносим данные из нее в удаляемую вершину. Теперь «максимальную» вершину можно удалить, воспользовавшись пунктом 1 или 2.

Заметим, что пункты 1 и 2 можно реализовать одной и той же последовательностью операторов, а пункт 3 следует рассматривать не столько как удаление элемента динамической структуры данных, сколько как удаление указанных данных из этой структуры.

Задача 3-3-13. Реализуйте рекурсивную процедуру удаления элемента из бинарного дерева поиска.

Идеи реализации. Напомним, что если удаляемая вершина имеет не более одного потомка, то удаление сводится к пере-

становке ссылки от родительской вершины на соответствующего потомка (возможно, пустого). Если удаляемая вершина имеет двух потомков, то сначала в левом поддереве ищется замещающая вершина `tmp` с максимальным значением поля `val`. Затем ее значение записывается в поле `val` удаляемой вершины, и далее из дерева удаляется вершина `tmp` (докажите, что вершина `tmp` имеет не более одного потомка).

Решение. Возвращаемое значение функции — указатель на корень дерева после удаления элемента.

```
struct TreeNode* Tree_Del(int x,
                          struct TreeNode* root) {
  struct TreeNode* tmp;
  if (root == NULL) {
    return NULL;  // пустое дерево
  }
  if (root->value == x) {  // удаление cur
    if (root->left == NULL &&
                            root->right == NULL) {
      free(root);
      return NULL;  // как листа
    }
    if (root->left == NULL) { // с потомком right
      tmp = root->right;
      free(root);
      return tmp;
    }
    if (root->right == NULL) { // с потомком left
      tmp = root->left;
      free(root);
      return tmp;
    }
    tmp = root->left; // поиск замещающей вершины
    while (tmp->right != NULL) {
      tmp = tmp->right;
    }
    root->value = tmp->value;
    root->left = Tree_Del(tmp->value, root->left);
    return root;
  } else { // поиск вершины с полем val, равным x
```

```
    if (x < root->value) {
      root->left = Tree_Del(x, root->left);
      return root;
    } else {
      root->right = Tree_Del(x, root->right);
      return root;
    }
  }
}
```

Задача 3-3-14. Постройте нерекурсивную процедуру удаления элемента в бинарном дереве поиска.

Задача 3-3-15. Если в алгоритме требуется явное перемещение по направлению к корню, то в структуру стоит добавить ссылку на родительскую вершину. Постройте процедуры добавления, удаления, поиска и печати элементов для бинарного дерева поиска с элементами абстрактного типа `Type` при условии, что связи между вершинами задаются тремя указателями:

```
typedef struct _TreeNode {
  Type value;
  struct _TreeNode *left, *right, *parent;
} TreeNode;
```

Задача 3-3-16. В рассмотренных выше процедурах добавления в дерево поиска элементы с одинаковыми значениями могут оказаться в дереве далеко друг от друга. Измените процедуры добавления так, чтобы элементы с одинаковыми значениями всегда оказывались рядом друг с другом (т. е. образовывали в ветви линейную цепочку).

Задача 3-3-17. Пусть концевые вершины бинарного дерева содержат числа, а всем остальным вершинам дополнительно поставлены в соответствие арифметические операции (в начальный момент числовые значения в неконцевых вершинах не определены). Назовем вычислением по дереву процедуру, записывающую в родительскую вершину значение, полученное в результате выполнения указанной арифметической операции со значениями в ее дочерних узлах. Разработайте форму представления такого дерева и реализуйте процедуру вычисления по дереву.

3.3.2. Сильно ветвящиеся деревья

Вершины произвольных деревьев удобно реализовывать с помощью структуры с двумя ссылками:

```
typedef struct _TreeNode {
  Type value;
  struct _TreeNode *down, *next;
} TreeNode;
```

В этом случае смысл указателей несколько меняется. Поле `down` указывает на начало *однонаправленного списка* прямых потомков данной вершины. По указателю `next` мы получаем доступ к следующему в списке прямому потомку родителя данной вершины. Отсутствие следующей вершины в списке потомков обозначается значением `NULL` соответствующего указателя (`next` или `down`).

Идея состоит в следующем. Возьмем всех потомков конкретной вершины и свяжем их в однонаправленный список по полю `next` в том порядке, в котором они появляются в дереве. В поле `down` родительской вершины поместим указатель на начало этого списка. Таким образом, в каждой вершине будут храниться два указателя: на следующий элемент в списке «братьев» и на начало списка потомков данной вершины. При необходимости прямого перемещения назад к корню, можно добавить еще один указатель: на родительскую вершину. Как всегда, нулевое значение указателя соответствует отсутствию последующего элемента (т. е. концу списка потомков или концу ветви).

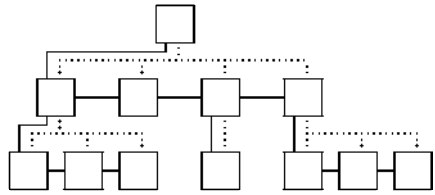


Рис. 3.3.5. Схема ссылок в произвольном дереве

На этом рисунке пунктирными линиями изображены логические связи между вершинами исходного дерева, а сплошными линиями — связи, определяемые указателями, хранящимися в реализации дерева. Приведем пример печати значения поля `value` для потомков вершины `cur`.

```
pos = cur->down;
while (pos) {
  printf("%d ", pos->value);
  pos = pos->next;
}
```

Процедуры обхода произвольного дерева строятся по аналогии с бинарным деревом, хотя обход слева направо теряет смысл, так как неясно, в какой момент (т. е. между какими потомками) следует обрабатывать родительскую вершину. В качестве примера приведем процедуру обхода сверху вниз для вычисления максимума среди элементов дерева.

```
void HBTree_VisUpDown(TreeNode *pos, Type *max) {
  if (!pos) return;
  if (*max < pos->value) *max = pos->value;
  pos = pos->down;
  while (pos) {
    HBTree_VisUpDown(pos, max);
    pos = pos->next;
  }
}
```

Вызов функции выглядит следующим образом:

```
if (root) {
  max_value = root->value;
  HBTree_VisUpDown(root, &max_value);
}
```

Отметим, что чисто топологически рассмотренная реализация произвольного дерева соответствует реализации некоторого бинарного дерева (так как каждый элемент имеет две ссылки на «последующие» элементы). Поэтому в задачах, где не требуется отслеживать принадлежность вершин определенному уровню, можно применять стандартные алгоритмы обхода бинарных деревьев. В остальных случаях наличие в узлах, по сути, явных ссылок на все дочерние элементы позволяет естественным образом модифицировать программный код.

Задача 3-3-18. Постройте процедуру обхода снизу вверх для произвольных деревьев.

Задача 3-3-19. Пусть произвольное дерево построено по следующим правилам. Каждая ветвь дерева, начиная с первого

уровня, соответствует некоторому слову (текстовой строке). Вершины k-го уровня дерева соответствуют k-й букве в записи слова. Значением вершины является пара

```
struct {
  char sym;
  char flag;
};
```

где `sym` — очередная буква слова, а значение `flag` равно 1, если существует слово, оканчивающееся на этой букве, и 0, если слово продолжается дальше. Например, в ветви, задающей слово «столовая», `flag` равен 1 для первой буквы «о» и букв «л» и «я». Каждый горизонтальный список потомков одной вершины упорядочен по значению поля `sym` (по алфавиту). Требуется реализовать процедуры добавления, удаления и поиска слова в таком дереве.

3.3.3. B-деревья

Встречаются ситуации, когда обрабатываемые данные невозможно полностью разместить в оперативной памяти. В этом случае приходится частично хранить их на диске и считывать в память по мере необходимости. Не используемую в данный момент информацию необходимо записывать обратно на диск, освобождая соответствующую часть оперативной памяти. В такой постановке задачи самыми неэффективными по времени являются именно операции обмена с диском. Таким образом, возникает проблема минимизации количества обращений к диску, возможно за счет некоторого увеличения накладных расходов по памяти и времени доступа к данным, уже находящимся в оперативной памяти. Для достижения этой цели удобным средством являются так называемые B-деревья. Дадим определение этой структуры.

B-*деревом порядка* n называется древовидная структура, удовлетворяющая следующим условиям:

— каждая вершина (узел) содержит массив, достаточный для хранения $2n$ значений данных, и массив для хранения $2n + 1$ ссылок на дочерние узлы;

— каждая вершина, кроме корневой, содержит не менее n и не более $2n$ значений данных;

— в каждой вершине данные хранятся в массиве, упорядоченном по возрастанию ключей;

— вершина, содержащая k ($n \leqslant k \leqslant 2n$) значений данных, имеет либо ровно $k+1$ дочерних элементов, либо является концевой, т. е. не имеет потомков;
— если в вершине хранится k значений данных и $k+1$ ссылок на дочерние вершины, то ключ данных с номером i, $i = 0, 1, \ldots, k-1$, больше любого ключа данных из i-го дочернего поддерева и меньше либо равен любого ключа данных из $(i+1)$-го дочернего поддерева;
— все концевые вершины лежат на одном уровне дерева.

B-дерево можно рассматривать как обобщение идеально сбалансированного дерева поиска на случай, когда потомков много.

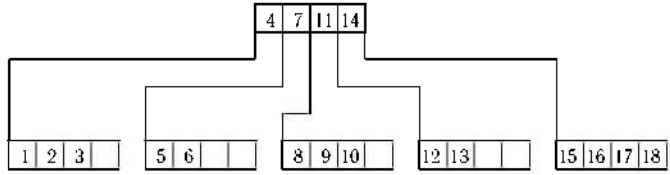


Рис. 3.3.6. Числовое B-дерево порядка 2 и глубины 2

Рассмотрим для простоты ситуацию, когда порядок дерева n — некоторое фиксированное число (например, 2048). В этом случае реализацию B-дерева порядка n можно построить на основе структуры

```
#define n 2048
struct BTreeNode {
  /* значения элементов: */
  Type value[2 * n];
  /* ссылки на дочерние вершины: */
  struct BTreeNode *child[2 * n + 1];
  /* текущее количество элементов: */
  int k;
};
```

Массив `value` предназначен для хранения элементов данных, k — фактическое количество элементов данных в вершине, массив `child` определяет ссылки на дочерние вершины.

При размещении B-дерева в памяти мы вынуждены резервировать место для 2n элементов в каждой вершине. Следовательно, возможны потери памяти, которые в наихудшем случае сравнимы с количеством размещенных элементов данных. Нетрудно подсчитать, что длина B-дерева, содержащего N

элементов данных, не превосходит $O(\log_n N)$, что при $n > 2$ существенно меньше, чем у бинарного дерева. B-деревья обычно применяются в тех случаях, когда данные для каждой вершины B-дерева имеют значительный объем и изначально хранятся на диске. При работе с таким деревом его вершины считываются с диска по мере необходимости. Таким образом, доступ к любому элементу требует прохождения по ветви дерева от корня до требуемого элемента, что можно осуществить за $O(\log_n N)$ обращений к диску, считывая каждый раз содержимое одной вершины.

Поскольку реализация B-дерева должна обеспечивать возможность считывания с диска данных для вершин-потомков, в структуру `BTreeNode` можно ввести еще массив ссылок на место размещения данных этих вершин-потомков на диске, например в виде смещений от начала соответствующего дискового файла:

```
#define n 2048
struct BTreeNode {
  /* значения элементов: */
  Type value[2 * n];
  /* ссылки на дочерние вершины: */
  struct BTreeNode *child[2 * n + 1];
  /* текущее количество элементов: */
  int k;
  /* смещения в файле: */
  size_t file_offset[2 * n + 1];
};
```

В этом случае ненулевой указатель `child[k]` определяет местоположение соответствующей вершины-потомка в памяти, а при нулевом значении `child[k]` мы прочитаем вершину из файла по смещению `file_offset[k]`, разместим ее в памяти, а адрес этого размещения занесем в `child[k]`. Для концевой вершины можно считать, что все значения массива `file_offset` равны нулю.

Поиск элемента в B-дереве

Опишем алгоритм поиска конкретного элемента `x`. Так как по условию построения B-дерева массив `value` в каждой вершине является упорядоченным, то мы можем методом деления пополам либо найти искомый элемент x, либо определить

позицию, где можно вставить x в этот массив. Будем считать, что в нашем распоряжении есть процедура бинарного поиска элемента в заданной вершине B-дерева

```
Type *BNodeSearch(Type x, BTreeNode *node,
                                        int *ind);
```

которая возвращает указатель на найденный элемент или **NULL** в случае неудачи. По указателю **ind** функция записывает индекс найденного элемента в массиве в случае успеха либо индекс потомка, в котором нужно далее искать **x**, т. е. записывает

0, если **x < node->value[0]**;

k, если **x > node->value[node->k - 1]**;

i, если **node->value[i-1] < x < node->value[i]** для **0 < i < node->k**.

Будем также считать, что для чтения вершины с диска реализована процедура **BReadNodeFromDisk**, которая получает в виде параметра нужное смещение и возвращает адрес, по которому она разместила в памяти прочитанную вершину.

Приведем реализацию функции поиска элемента, которая получает в качестве параметров искомый элемент **x**, стартовую вершину поиска **node** и возвращает или указатель на найденный элемент, или **NULL** при отсутствии такового.

```
Type *BTreeSearch(FILE *BTin, Type x,
                                BTreeNode *node) {
  int ind;
  Type *find;
  // для пустого дерева ничего не делаем:
  if (!node) return NULL;
  // пытаемся найти требуемый элемент:
  if (find = BNodeSearch(x, node, &ind)) {
    // нашли в корневой вершине:
    return find;
  }
  // переходим к поиску в потомке:
  if (!node->child[ind]) {// потомка нет в памяти,
    // надо считать его с диска:
    if (!node->offset[ind])
      // концевая вершина:
      return NULL;
```

```
    else {// пытаемся прочитать с диска:
      node->child[ind] =
       BReadNodeFromDisk(BTin, node->offset[ind]);
      // проверим, удалось или нет:
      if (!node->child[ind]) return NULL;
    }
  }
  // теперь потомок в памяти, ищем в нем:
  return BTreeSearch(BTin, x, node->child[ind]);
}
```

Если не принимать во внимание действия по определению файла с данными и различные проверки корректности, использование этой процедуры может выглядеть, например, так:

```
BTreeNode *root;
Type *find;
FILE *BTin = fopen("BTree", "rb");
if (BTin == NULL) return -1;
root = BReadNodeFromDisk(BTin, 0);
find = BTreeSearch(BTin, x, root);
```

Здесь предполагается, что данные корневой вершины записаны в самом начале дискового файла.

Нетрудно подсчитать, что сложность поиска в B-дереве порядка n с N элементами в наихудшем случае сводится к проходу от корня до концевой вершины с выполнением бинарного поиска в вершинах каждого уровня, что составляет $O(\log_2(2n)\log_n N) \sim$ $\sim O(\log_n N)$ арифметических операций и сравнений.

Добавление элемента в B-дерево

Опишем схему добавления элемента в B-дерево поиска. Сначала отыскивается концевая вершина, в которую можно добавить требуемый элемент в соответствии с упорядоченностью существующих элементов. Если добавление элемента в массив этой вершины не превышает допустимый размер (напомним, что это 2n), то выполняется вставка в упорядоченный массив и на этом процесс завершается. Если в концевой вершине (обозначим ее A) уже размещено 2n элементов, то, добавляя новое значение, мы получаем упорядоченный набор из $2n + 1$ элементов. По определению B-дерева в вершине не может быть размещено такое количество элементов, поэтому требуется перестроение.

При перестроении первым способом создается новая концевая вершина (обозначим ее B) и первые n элементов набора размещаются в старой вершине, последние n — во вновь созданной вершине, а средний элемент набора (обозначим его y) вставляется в родительскую вершину. При этом справа от y в родительской вершине вставляется ссылка на нового потомка B. Если прямое добавление y в родительскую вершину невозможно (она уже имела 2n элементов), то опять выполняется ее разделение на две вершины, а средний элемент переносится в вершину выше по ветви. При полной занятости всех вершин в ветви этот процесс переноса среднего элемента распространяется до корня. Если корень также заполнен, то он аналогично разделяется на две вершины, а средний элемент образует новый корень.

Второй способ перестроения заключается в попытке перемещения части «лишних» данных в соседнюю правую (либо левую) вершину, что потребует затем модификации родительской вершины. Если соседние вершины заполнены, то производится расщепление первым способом.

Третий способ заключается в объединении данных двух соседних вершин и их расщеплении в три новых вершины. При этом необходимо будет модифицировать родительскую вершину. Если у расщепляемой вершины нет соседних, то это корень. Корень расщепляется на два узла.

В отличие от бинарных деревьев поиска, B-дерево растет снизу вверх. Самым трудоемким шагом в процедуре добавления является процесс разделения вершин, который требует «раздвигания» массивов данных для вставки нового элемента. Таким образом, наилучшая оценка трудоемкости составляет в среднем $O(\log_n N \log_2 n + n)$ операций (спуск по ветви и одна вставка), а наихудшая оценка — $O(n \log_n N)$ (происходит разделение вершин во всей ветви). Отметим, что последняя оценка не является такой уж обременительной, поскольку в этом случае появляется много вершин, заполненных лишь наполовину, и последующие добавления уже долго не будут вызывать разделения вершин.

Удаление элемента из B-дерева

Алгоритм удаления элемента из B-дерева имеет общие черты с удалением из бинарного дерева, и его можно получить обращением описанной выше процедуры добавления элемента. Если нужный элемент лежит в концевой вершине, то удаление не

составляет труда. Если же этот элемент расположен в другой вершине, то сначала он заменяется на максимальный элемент из поддерева, берущего свое начало от «левой» ссылки удаляемого элемента. Несложно показать, что этот элемент лежит в концевой вершине. Теперь можно удалить этот максимальный элемент. В результате удаления элемента из концевой вершины (обозначим ее A) может оказаться, что количество элементов в ней станет равным $n - 1$, т. е. нарушится одно из условий B-дерева. В этом случае производится балансировка, состоящая в объединении элементов вершины A, элемента y родительской вершины, для которого «левая» ссылка указывает на A, и элементов соседней концевой вершины B (определяемой правой ссылкой от y). Полученный набор данных делится на равные (с точностью до одного элемента) «левую» и «правую» части, разделяемые «средним» элементом. Средний элемент заменяет y в родительской вершине, левая часть записывается в A, а правая — в B. Если в вершине B только n элементов, то такое слияние вершин невозможно (опять нарушится условие B-дерева). В этом случае весь объединенный набор (его длина равна $2n$) записывается в A, вершина B удаляется и из родительской вершины также удаляется элемент y вместе со ссылкой на B. При удалении y из родительской вершины число элементов в ней также может уменьшиться до $n - 1$. В этом случае аналогично перераспределяются элементы двух соседних вершин данного уровня (либо вершины сливаются в одну). Описанный процесс может распространиться до корня. Если количество элементов в корне при этом станет равным нулю (а это возможно только при слиянии двух его потомков), то корень удаляется, а новым корнем становится единственная объединенная при последнем слиянии вершина. Глубина дерева при этом уменьшается на единицу. Легко подсчитать, что сложность удаления совпадает со сложностью добавления и составляет от $O(\log_n N \log_2 n + n)$ до $O(n \log_n N)$ операций в зависимости от текущей заполненности вершин.

Задача 3-3-20. Реализуйте процедуры добавления, поиска, удаления и обхода (например, печати элементов) для B-дерева целых чисел при $n = 1$, т. е. для так называемого 2-3-дерева. Вершина такого дерева может хранить одно или два целых числа и две или три ссылки на своих потомков.

Задача 3-3-21. Реализуйте стандартные процедуры для работы с B-деревом порядка n, содержащим элементы некоторого типа Type.

Задача 3-3-22. Разработайте формат файла для сохранения B-дерева на диске и реализуйте процедуры добавления, удаления и поиска элементов некоторого типа Type при условии, что все дерево не помещается в оперативную память.

Идеи реализации. Пронумеруем все вершины дерева последовательными натуральными числами. В начале файла должна содержаться таблица, которая устанавливает соответствие между номером вершины и смещением от начала файла, определяющим местоположение этой вершины. Поскольку количество вершин в дереве может меняться, то следует предусмотреть способы для удлинения этой таблицы. За этой таблицей размещаются вершины B-дерева. Прочитав данную таблицу в оперативную память, мы получаем возможность считывания любой вершины по ее номеру.

Для хранения вершины B-дерева воспользуемся структурой следующего вида:

```
struct BTreeNode {
  /* значения элементов: */
  Type value[2 * n];
  /* ссылки на дочерние вершины: */
  struct BTreeNode *child[2 * n + 1];
  /* текущее количество элементов: */
  int k;
  /* номер вершины: */
  int number;
};
```

Пусть мы имеем возможность разместить в памяти N вершин. Создадим список (массив), в котором будем хранить номер вершины, указатель на начало ее размещения в памяти, смещение в файле для данной вершины, количество обращений к элементам данной вершины. Если нужная нам вершина уже расположена в памяти, то указанная информация обеспечивает возможность доступа к ее элементам (при этом следует корректировать поле, относящееся к количеству обращений), а также запись вершины из памяти обратно в файл. Если вершины в памяти нет, то она считывается с диска и информация о ней

записывается в свободную ячейку этого списка (массива). Если список уже содержит N значений, то мы должны освободить один из его элементов, переписав одну из вершин обратно на диск. При этом можно воспользоваться одной из следующих стратегий:

1) освобождается та вершина, которая имела наименьшее количество обращений (т. е. наименее используемая вершина);
2) освобождается та вершина, которая имела наибольшее количество обращений (возможно, элементы этой вершины уже обработаны и далее не потребуются);
3) подсчитывается общее число m обращений ко всем элементам B-дерева; при достижении числом m некоторого порогового значения количество обращений к каждой отдельной вершине обнуляется; освобождается та вершина, которая на данный момент имеет наименьшее количество обращений (т. е. последнее время реже всего использовалась).

Задача 3-3-23. Постройте полные реализации описанных выше деревьев в виде параметризованных классов на языке C++.

Задача 3-3-24. Для каждого из деревьев реализуйте итератор по его элементам. Итератором обычно называется набор функций, позволяющих последовательно перебирать все элементы из заданного множества. Например, можно предложить следующий интерфейс для итератора:

```
/* начать итерирование с начала: */
void StartIteration(struct Tree *tree);

/* получить указатель на очередное значение,
   возвращает NULL, если больше элементов нет: */
Type *GetNextValue(struct Tree *tree);
```

В данном случае считаем, что все необходимые для работы объекты хранятся в структуре по указателю `tree`. Для такого итератора пример печати всех значений из множества при помощи некоторой функции `Print` может выглядеть, например, так:

```
Type *v;
for (StartIteration(tree); v = GetNextValue();) {
  Print(v);
}
```

Реализация итератора для деревьев отличается от процедур обходов: обходы используют рекурсию и выполняются «за один прием», а функции итератора вызываются независимо друг от друга. Следовательно, данная реализация должна где-то хранить состояние обхода. Проще всего для этого в каждой вершине завести ссылку на родителя и отдельно указатель на текущую вершину либо формировать список посещенных вершин, задающий текущую ветвь дерева.

3.3.4. Сбалансированные бинарные деревья

Деревья поиска хороши тем, что вычислительная сложность выполнения процедур поиска, добавления и удаления элемента зависит от длины пути от корня дерева до интересующей нас вершины и в худшем случае определяется длиной максимальной ветви дерева. Если в бинарном дереве почти все вершины имеют двух потомков, то длина максимальной ветви оценивается величиной $O(\log_2 N)$, где N — общее число вершин дерева. Однако надеяться на то, что дерево поиска, сформированное с помощью описанного выше алгоритма добавления, окажется именно таким, в общем случае мы не можем. Действительно, если данные, последовательно поступающие на вход процедуры добавления, окажутся упорядоченными, например, по убыванию, то каждый новый элемент будет добавляться в левое поддерево относительно последнего добавленного элемента. Таким образом, дерево выродится в линейный список, а поиск в таком дереве будет в среднем иметь сложность $O(N)$. С другой стороны, те же самые данные можно организовать в виде другого дерева поиска, где каждая ветвь будет иметь длину порядка $O(\log_2 N)$.

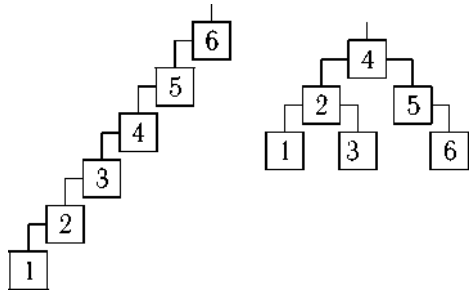


Рис. 3.3.7. Два
варианта дерева поиска при одних и тех же данных

Нельзя ли так организовать процесс добавления элементов в дерево поиска, чтобы все его ветви «росли» по возможности равномерно? Ответ на этот вопрос положительный. Для большей четкости дальнейшего изложения напомним следующие определения.

Длина ветви равна количеству вершин в соответствующем связном участке.

Бинарное дерево называется *идеально сбалансированным*, если длины любых двух его ветвей (считая от корня) отличаются не более чем на единицу.

Идеально сбалансированное дерево решает поставленную задачу, так как в нем длина любой ветви не превосходит величины $[\log_2 N] + 1$. К сожалению, формирование такого дерева — достаточно трудоемкая процедура. Поэтому мы несколько ослабим требования на идеальность балансировки дерева и попытаемся за счет этого построить более эффективные алгоритмы. Напомним определение сбалансированного дерева.

Длиной (*глубиной*) *дерева* будем называть длину его максимальной ветви. Бинарное дерево называется *сбалансированным*, если в любой его вершине длины левого и правого поддеревьев отличаются не более чем на 1. Такие структуры известны в литературе как АВЛ-деревья (названы в честь авторов: Г. М. Адельсона-Вельского и Е. М. Ландиса, 1962 г.).

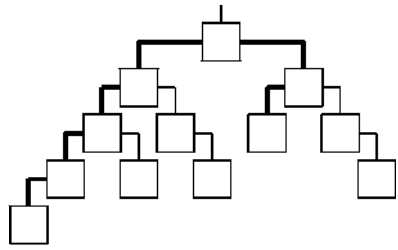


Рис. 3.3.8. Сбалансированное, но не идеально сбалансированное дерево

Очевидно, что идеально сбалансированное дерево является сбалансированным. Обратное неверно, как видно из примера, изображенного на рис. 3.3.8. В данном случае в каждой вершине выполнено условие сбалансированности, но длины двух ветвей, исходящих от корня и выделенных жирной линией, отличаются на 2.

Несложно показать, что длина сбалансированного дерева превосходит длину идеально сбалансированного дерева с таким же числом вершин не более чем в 1,5 раза. Для этого, например, можно построить «наихудшее» сбалансированное дерево, в котором в каждой вершине (за исключением концевых) левое поддерево длиннее правого на единицу. Обозначив через $N(k)$ количество элементов в таком дереве, имеющем длину k, можно получить рекуррентное соотношение $N(0) = 0$, $N(1) = 1$, $N(k) = N(k-1) + N(k-2) + 1$, из которого и выводится требуемое утверждение. Например, методами теории разностных уравнений можно получить явное выражение для $N(k)$. Это выражение дает асимптотику $N(k) \sim \left((1+\sqrt{5})/2\right)^k$. Таким образом, $k \sim \log_2 N / \log_2\left((1+\sqrt{5})/2\right) \leqslant 1{,}45 \cdot \log_2 N$.

Данная оценка показывает, что трудоемкость поиска в сбалансированном дереве также величина порядка $O(\log_2 N)$. Но самое главное — добавление и удаление элементов в таком дереве также можно выполнить со сложностью $O(\log_2 N)$ действий. Опишем эти процедуры. Прежде всего отметим, что мы имеем дело с деревом поиска и размещение элементов в узлах дерева должно удовлетворять условию упорядоченности ключей в левом и правом поддеревьях. Следовательно, наличие нескольких элементов с одинаковыми ключами может войти в противоречие либо с требованием упорядоченности, либо с требованием сбалансированности. Действительно, достаточно рассмотреть случай, когда все элементы имеют один и тот же ключ. Обычное дерево поиска в этом случае выродится в одну (правую) ветвь. Организовав те же самые элементы в сбалансированное дерево, мы нарушим требование, что ключи в правом поддереве должны быть больше ключей в левом поддереве. Поэтому далее, рассматривая сбалансированные деревья, мы будем считать, что все элементы имеют различные ключи. При этом для наглядности изучаемых алгоритмов мы ограничимся описанием только рекурсивных процедур.

Добавление элемента в АВЛ-дерево

Добавление элемента в сбалансированное дерево происходит по тому же принципу, что и добавление в обычное дерево поиска. Но добавленный в дерево элемент может нарушить сбалансированность в вершинах, расположенных выше. Следовательно, нам придется восстанавливать сбалансированность, т. е. некоторым образом перестраивать дерево на обратном проходе рекурсии.

Для этого нам необходимо иметь информацию о текущей сбалансированности дерева, поэтому мы будем в каждой вершине хранить показатель ее сбалансированности, например значение разности длин правого и левого поддерева:

```
class TreeNode {
 public:
  Type value;
  int balance;
  TreeNode *left, *right;
};
```

Если поле `balance` имеет значения -1, 0, 1, то данная вершина сбалансированна. В противном случае нам придется перестраивать дерево. Рассмотрим данный процесс подробнее.

При добавлении очередного элемента в поддерево с корнем A соответствующая процедура сравнивает значение добавляемого элемента со значением, расположенным в вершине A, и, в зависимости от результата сравнения, рекурсивно вызывает себя для левого либо правого поддерева вершины A. При возвращении из рекурсивных вызовов процедура добавления должна будет балансировать дерево. Для этого, вернувшись в вершину A, мы вычислим, как изменилась разность длин левого и правого поддеревьев в результате добавления, что позволит нам реализовать балансировку текущей вершины. Поднимаясь таким образом шаг за шагом к корню дерева, мы получим полное сбалансированное дерево с добавленным элементом.

Задача 3-3-25. Постройте рекурсивную процедуру добавления элемента в бинарное сбалансированное дерево поиска.

Указание. Изложим на схеме весь процесс добавления и балансировки. Способ балансировки в вершине A зависит от того, в какое поддерево (левое или правое) был добавлен элемент. Для определенности рассмотрим случай добавления в левое поддерево.

Рис. 3.3.9. Поддерево до и после добавления

Здесь AL′ и AR — это левое и правое поддеревья для вершины A, а AL — поддерево, получившееся после добавления элемента.

Отметим, что по предположению рекурсии дерево AL будет уже сбалансированным и наша задача теперь заключается в балансировке поддерева с корнем A. Обозначим через h_{AL}, h_{AR}, $h_{AL'}$ длины соответствующих поддеревьев. Имеем два случая.

1. $h_{AL} = h_{AL'}$. Ничего делать не надо, поскольку для вершины A в смысле баланса ничего не изменилось.

2. $h_{AL} = h_{AL'} + 1$ (длина левого поддерева увеличилась). Здесь возможны варианты в зависимости от первоначального значения баланса в вершине A.

Таблица 3.3.1.

Старый баланс в вершине A	Новый баланс в вершине A	Длина дерева в вершине A	Балансировка
1	0	не изменилась	не нужна
0	−1	увеличилась на 1	не нужна
−1	−2	увеличилась на 1	нужна

Заметим, что в последнем случае длина исходного дерева в вершине A равнялась $h_A = h_{AL'} + 1 = h_{AR} + 2$ (так как при балансе -1 мы имеем $h_{AL'} = h_{AR} + 1$). Обозначим $k = h_{AL'}$. Тогда для нового дерева имеем $h_{AL} = k + 1$, $h_{AR} = k - 1$. Так как баланс в вершине A был равен -1, то $k > 0$. Следовательно, длина поддерева AL не менее двух, мы можем записать его в виде корня B и соответствующих поддеревьев BL и BR, т. е. представить новое дерево следующим образом:

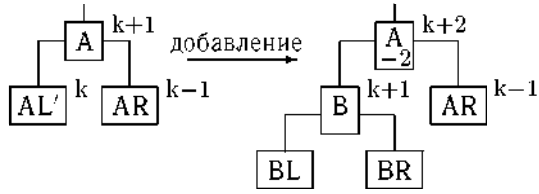


Рис. 3.3.10. Длины и балансы поддеревьев при добавлении в левое поддерево для баланса -1 в вершине A

На рисунке справа от вершин указаны длины соответствующих поддеревьев, а в вершине A проставлен реально получившийся баланс. Поскольку дерево с корнем B уже сбалансировано нашей рекурсивной процедурой, мы можем рассмотреть формально три случая.

1. Баланс B равен 0, т. е. $h_{BL} = k$, $h_{BR} = k$. Перестраиваем дерево следующим образом (для вершин указаны старые и новые балансы и длина).

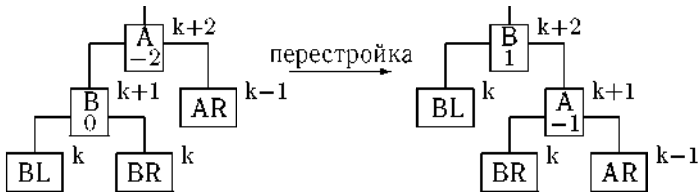


Рис. 3.3.11. Перестройка при балансе B, равном 0

Из этой диаграммы видно, что длина нового дерева увеличилась по сравнению со старой (было $k+1$, стало $k+2$). Нетрудно заметить, что упорядоченность вершин и поддеревьев сохранилась.

2. Баланс B равен -1, т. е. $h_{BL} = k$, $h_{BR} = k-1$.

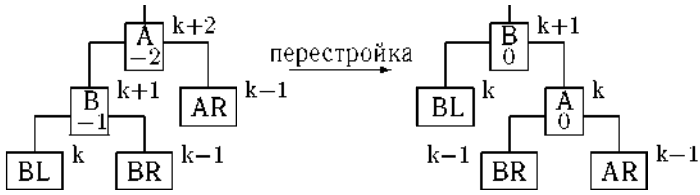


Рис. 3.3.12. Перестройка при балансе B, равном -1

Этот случай отличается от предыдущего только расстановкой балансов и тем, что длина нового поддерева с корнем A совпадает со старой длиной.

3. Баланс B равен 1, т. е. $h_{BL} = k-1$, $h_{BR} = k$.

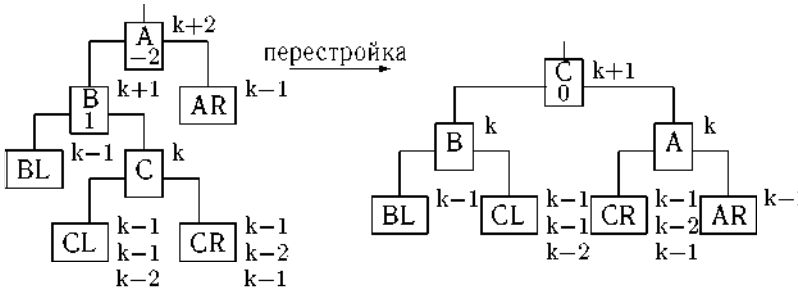


Рис. 3.3.13. Перестройка при балансе B, равном 1

Здесь длина нового дерева не увеличилась по сравнению со старым, баланс в новом корне C всегда равен нулю, а балансы в вершинах A и B зависят от старого баланса в вершине C (на диаграмме указаны возможные длины поддеревьев CL и CR).

Назовем L1-перестройкой преобразование дерева в соответствии с пунктами 1 и 2, а преобразование из пункта 3 — L2-перестройкой.

Суммируя сказанное, составим сводную таблицу балансировки для случая добавления в левое поддерево.

Таблица 3.3.2.

Балансы						Изм. длины	Перестройка
Старые			Новые				
A	B	C	A	B	C		
1	—	—	0	—	—	0	—
0	—	—	−1	—	—	1	—
−1	−1	—	0	0	—	0	L1
−1	0	—	−1	1	—	1	L1
−1	1	0	0	0	0	0	L2
−1	1	1	0	−1	0	0	L2
−1	1	−1	1	0	0	0	L2

Внимательный анализ диаграмм и таблиц показывает, что если после добавления и балансировки дерева баланс его текущего узла оказывается равным нулю, то длина этого дерева не возрастает, и следовательно, необходимости в преобразовании 1 (L1-перестройке с балансом B, равным нулю) никогда не возникает. Таким образом, при построении программы этот случай можно не рассматривать.

При добавлении элемента в правое поддерево рассуждения проводятся аналогично. Опираясь на построенные диаграммы и таблицы, можно реализовать процедуру добавления с балансировкой. Для удобства выделим процедуры перестройки дерева с расстановкой необходимых балансов в отдельные функции. Функция `Rebuild_L1` будет перестраивать дерево и корректировать балансы в соответствии со схемой L1-перестройки, функция `Rebuild_L2` — в соответствии со схемой L2-перестройки. Функции `Rebuild_R1` и `Rebuild_R2` являются симметричными аналогами `Rebuild_L1` и `Rebuild_L2` применительно к добавлению в правое поддерево. Все эти функции будут возвращать указатель на новый корень полученного поддерева.

```
TreeNode *Rebuild_L1(TreeNode *A) {
  TreeNode *B;
  B = A->left;
```

```
  A->left = B->right;
  B->right = A;
  if (B->balance) {
    A->balance = 0;
    B->balance = 0;
  } else {
    A->balance = -1;
    B->balance = 1;
  }
  return B;
}
TreeNode *Rebuild_L2(TreeNode *A) {
  TreeNode *B, *C;
  B = A->left;
  C = B->right;
  A->left = C->right;
  B->right = C->left;
  C->left = B;
  C->right = A;
  switch (C->balance) {
    case 0:
      A->balance = 0;
      B->balance = 0;
      break;
    case -1:
      A->balance = 1;
      B->balance = 0;
      break;
    case 1:
      A->balance = 0;
      B->balance = -1;
  }
  C->balance = 0;
  return C;
}
TreeNode *Rebuild_R1(TreeNode *A) {
  TreeNode *B;
  B = A->right;
  A->right = B->left;
  B->left = A;
```

```
  if (B->balance) {
    A->balance = 0;
    B->balance = 0;
  } else {
    A->balance = 1;
    B->balance = -1;
  }
  return B;
}
TreeNode *Rebuild_R2(TreeNode *A) {
  TreeNode *B, *C;
  B = A->right;
  C = B->left;
  B->left = C->right;
  A->right = C->left;
  C->left = A;
  C->right = B;
  switch (C->balance) {
    case 0:
      A->balance = 0;
      B->balance = 0;
      break;
    case 1:
      A->balance = -1;
      B->balance = 0;
      break;
    case -1:
      A->balance = 0;
      B->balance = 1;
  }
  C->balance = 0;
  return C;
}
```

Теперь реализация функции добавления с балансировкой не составляет труда. Как видно из схем перестройки дерева, после балансировки корнем может стать другая вершина, поэтому естественно в качестве возвращаемого значения для нашей функции выбрать указатель на новый полученный корень. Заметим также, что нам надо знать, увеличилась ли

длина дерева в процессе добавления. Поэтому нам потребуется еще один дополнительный параметр `(*grow)`, который будет равен удлинению дерева после добавления элемента. Итак, мы приходим к следующей реализации.

```
/* Добавление элемента x в сбалансированное
 деревос корнем root. Если root NULL,
 то создается новый узел. Возвращается либо root,
 либо NULL (только на последнем рекурсивном шаге
 и если нет памяти). Указатель grow отвечает
 за признак удлинения дерева. */
TreeNode *AddBalance(Type x, TreeNode *root,
                                        int *grow) {
  int incr;
  *grow = 0;
  /* пустое дерево: */
  if (!root) {
    root = Tree_CreateNode();
    if (root) {
      root->left = root->right = 0;
      root->value = x;
      root->balance = 0;
      *grow = 1;
    }
    return root;
  }
  /* непустое дерево: */
  if (x <= root->value) {  // левая балансировка:
    root->left = AddBalance(x, root->left, &incr);
    if (incr) {
      switch (root->balance) {
        case 0:
          root->balance = -1;
          *grow = 1;
          break;
        case 1:
          root->balance = 0;
          break;
        case -1:
          switch (root->left->balance) {
```

```
            case -1:
              root = Rebuild_L1(root);
              break;
            case 1:
              root = Rebuild_L2(root);
          }
      }
    }
  } else {  //  правая балансировка:
    root->right = AddBalance(x, root->right,
                                              &incr);
    if (incr) {
      switch (root->balance) {
        case 0:
          root->balance = 1;
          *grow = 1;
          break;
        case -1:
          root->balance = 0;
          break;
        case 1:
          switch (root->right->balance) {
            case 1:
              root = Rebuild_R1(root);
              break;
            case -1:
              root = Rebuild_R2(root);
          }
      }
    }
  }
  return root;
}
```

Замечание. Типичный фрагмент программы, заполняющей дерево, скажем, целыми числами, читаемыми из файла, может выглядеть так:

```
typedef int Type;
FILE *fin;
Type x;
```

```
TreeNode *root = NULL;
int incr;
.....
while (fscanf(fin, "%d", &x) == 1) {
  root = AddBalance(x, root, &incr);
  if (root == NULL) break;
}
```

Удаление элемента из АВЛ-дерева

Рассмотрим удаление из сбалансированного дерева. В целом эта процедура аналогична удалению из обычного дерева поиска. Если удаляемая вершина имеет не более одного потомка, то достаточно только изменить ссылку от родительской вершины. Если же потомков два, то, как и раньше, в левом поддереве ищется вершина для подмены значения удаляемой вершины. Отличие алгоритма связано с необходимостью балансировки текущей вершины, если при удалении элемента длина левого или правого поддерева уменьшилась.

Задача 3-3-26. Постройте рекурсивную процедуру удаления элемента из бинарного сбалансированного дерева поиска.

Указание. Отметим, что разбалансировка при удалении из левого поддерева совпадает с разбалансировкой при добавлении в правое поддерево, и наоборот. Поэтому и процедуры восстановления балансов в случае удаления будут, по сути, совпадать с соответствующими процедурами перестройки при добавлении элемента. Рассмотрим только случай удаления из левого поддерева.

Итак, мы имели поддерево с вершиной A и старой длиной $k+1$ и при удалении уменьшилась длина поддерева AL. Если баланс A был равен -1 или 0, то перестройка не нужна, а балансировка сводится только к коррекции баланса вершины A (баланс вершины A становится равен 0 или 1 соответственно). Если же баланс A был равен 1, то наше поддерево принимает вид

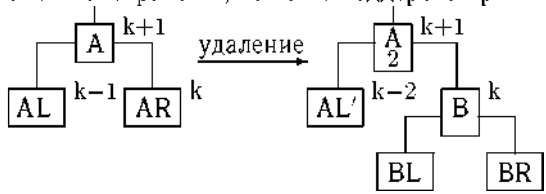


Рис. 3.3.14. Удаление элемента из левого поддерева

Опять рассмотрим балансы в вершине B.

1. Баланс B равен 0. Выполняется R1-перестройка.

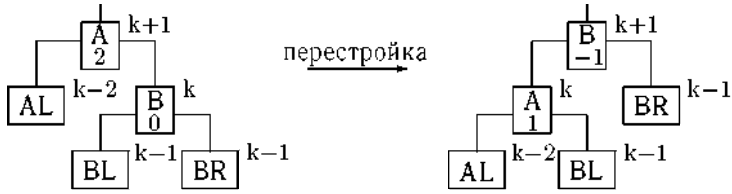


Рис. 3.3.15. Перестройка при балансе B, равном 0

Длина поддерева с вершиной A не изменилась.

2. Баланс B равен 1. Выполняется R1-перестройка.

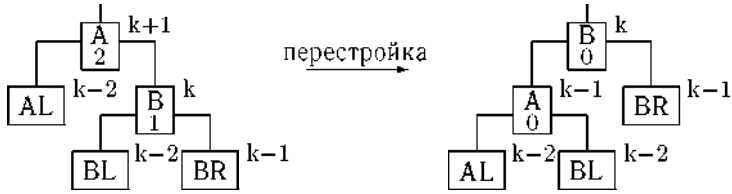


Рис. 3.3.16. Перестройка при балансе B, равном 1

Длина поддерева с вершиной A уменьшилась.

3. Баланс B равен -1. Выполняется R2-перестройка.

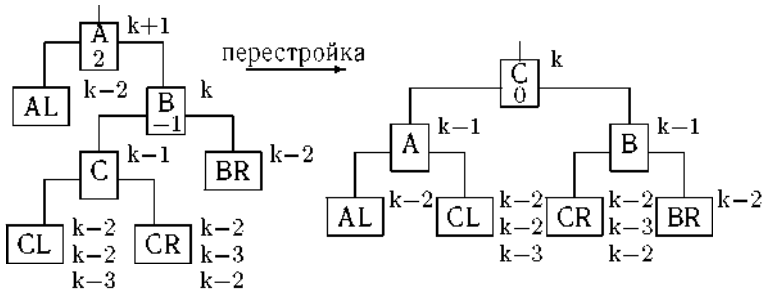


Рис. 3.3.17. Перестройка при балансе B, равном -1

Длина поддерева с вершиной A уменьшилась, а балансы в вершинах A и B зависят от старого баланса C.

Сводная информация о замене балансов при удалении из левого поддерева приведена в следующей таблице.

Таблица 3.3.3.

Балансы						Изм. длины	Перестройка
Старые			Новые				
A	B	C	A	B	C		
−1	—	—	0	—	—	−1	—
0	—	—	1	—	—	0	—
1	0	—	1	−1	—	0	R1
1	1	—	0	0	—	−1	R1
1	−1	0	0	0	0	−1	R2
1	−1	−1	0	1	0	−1	R2
1	−1	1	−1	0	0	−1	R2

Заметим, что если после удаления и балансировки длина дерева уменьшается, то баланс корня всегда будет равен нулю. К сожалению, в отличие от процедуры добавления, этот факт не позволит упростить программу, поскольку в данном случае удаление и соответствующая балансировка выполняются для левого поддерева, а перестройка захватывает правое поддерево (баланс поддерева AL нигде не участвует).

Параметры у функции для удаления элемента такие же, как и у функции добавления. Приведем ее код без дальнейших пояснений.

```
/* Удаление элемента x из сбалансированного
   дерева с корнем root. Возвращается указатель
   на корень нового дерева */
TreeNode *DelBalance(Type x, TreeNode *root,
                                    int *grow) {
  int incr;
  TreeNode *pos;
  *grow = 0;
  if (!root) return 0;    // пустое дерево
  /* непустое дерево: */
  if (x == root->value) {
    if (root->left == 0) {
      /* если не более одного потомка: */
      *grow = -1;
      pos = root->right;
      Tree_FreeNode(root);
      return pos;
    }
```

```
    if (root->right == 0) {
      *grow = -1;
      pos = root->left;
      Tree_FreeNode(root);
      return pos;
    }
    /* два потомка, ищем элемент для подмены: */
    for (pos = root->left; pos->right;)
      pos = pos->right;
    root->value = pos->value;
    root->left =
       DelBalance(pos->value, root->left, &incr);
    if (incr) {
      switch (root->balance) {
        case -1:
          root->balance = 0;
          *grow = -1;
          break;
        case 0:
          root->balance = 1;
          break;
        case 1:
          switch (root->right->balance) {
            case 1:
              *grow = -1;
              root = Rebuild_R1(root);
              break;
            case 0:
              root = Rebuild_R1(root);
              break;
            case -1:
              root = Rebuild_R2(root);
              *grow = -1;
          }
      }
    }
  } else {
    if (x < root->value) {
      root->left = DelBalance(x, root->left,
                                          &incr);
```

```
    if (incr) {
      switch (root->balance) {
        case -1:
          root->balance = 0;
          *grow = -1;
          break;
        case 0:
          root->balance = 1;
          break;
        case 1:
          switch (root->right->balance) {
            case 1:
              *grow = -1;
              root = Rebuild_R1(root);
              break;
            case 0:
              root = Rebuild_R1(root);
              break;
            case -1:
              root = Rebuild_R2(root);
              *grow = -1;
          }
      }
    }
  } else {
    root->right = DelBalance(x, root->right,
                                          &incr);
    if (incr) {
      switch (root->balance) {
        case 1:
          root->balance = 0;
          *grow = -1;
          break;
        case 0:
          root->balance = -1;
          break;
        case -1:
          switch (root->left->balance) {
            case -1:
              *grow = -1;
```

```
                root = Rebuild_L1(root);
                break;
              case 0:
                root = Rebuild_L1(root);
                break;
              case 1:
                root = Rebuild_L2(root);
                *grow = -1;
            }
        }
      }
    }
  }
  return root;
}
```

3.3.5. Красно-черные деревья

Логарифмическую сложность работы с деревьями поиска также предоставляют так называемые красно-черные деревья. Бинарное дерево называется *красно-черным*, если оно удовлетворяет следующим требованиям:

1) каждая вершина имеет свой цвет — красный или черный; корень дерева всегда черный;
2) красная вершина может иметь только черных потомков;
3) в любой ветке от корня до концевой вершины содержится одинаковое число черных вершин (это число называется черной длиной дерева).

Третье условие, собственно, и дает основание надеяться на необходимую логарифмическую сложность поиска, поскольку красно-черное дерево является «идеально сбалансированным по черной длине». Более строго этот факт устанавливает следующее утверждение: глубина красно-черного бинарного дерева с N вершинами не превосходит $2\log_2(N+1)$.

Действительно, пусть исходное красно-черное дерево имеет глубину H, а его черная глубина равна H_b. Напомним, что глубина дерева — это количество вершин в максимально длинной ветви. Из первых двух условий следует, что в каждой ветви красно-черного дерева количество красных вершин не превосходит количества черных (т. е. не превосходит H_b). Отсюда, учитывая

третье условие, получаем, что длины двух произвольных ветвей отличаются не более чем в два раза. Следовательно, выполняется оценка $H \leqslant 2H_b$ и до глубины H_b дерево является полностью заполненным. Но в заполненном дереве глубины H_b имеется $2^{H_b} - 1$ элементов, следовательно, $N \geqslant 2^{H_b} - 1$. Отсюда получаем требуемую оценку $H \leqslant 2H_b \leqslant 2\log_2(N+1)$.

Теперь нужно построить алгоритмы, позволяющие работать с красно-черным деревом (добавлять и удалять элементы), также с логарифмической оценкой сложности.

Добавление элемента в красно-черное дерево

Как и в сбалансированном дереве, алгоритм добавления предполагает некоторые перестройки дерева для сохранения указанных в определении свойств раскраски. Для единообразия при описании процедур добавления/удаления нам будет удобно сделать следующее допущение: каждую нулевую ссылку в имеющемся дереве (т. е. **NULL**-ссылку на потомка) будем интерпретировать как формальную ссылку на не содержащую данные вершину-лист черного цвета. Эти вершины нам потребуются для унификации алгоритма перекраски.

Итак, процедура добавления состоит из следующих этапов.

1. Добавляем элемент в дерево поиска по обычному алгоритму. Добавленный элемент становится концевой вершиной. Красим эту вершину в красный цвет.
2. Если возникает конфликт со вторым условием из определения красно-черного дерева, выполняем перестройки и перекраски дерева для восстановления правильных свойств (конфликт с третьим условием возникнуть не может, так как добавляемая вершина красная).

Задача 3-3-27. Постройте рекурсивную процедуру добавления элемента в красно-черное дерево поиска.

Идеи реализации. Ситуация перестройки распадается на несколько случаев. В отличие от сбалансированных деревьев, здесь недостаточно анализа параметров только родительской вершины — приходится рассматривать два уровня дерева. На приводимых далее диаграммах кружок обозначает красную вершину, квадрат — черную, а маленький прямоугольник — черную фиктивную концевую вершину или черный корень поддерева, лежащего ниже. Цифры в прямоугольниках обозначают корни поддеревьев с произвольным цветом, фиктивные концевые

вершины и даже отсутствие каких-либо вершин, если сами B, C, D являются фиктивными концевыми вершинами. Звездочка стоит около вершины, для которой нарушено второе условие.

С точностью до симметрии возможные ситуации описываются следующими схемами, определяемыми цветом «дяди» (вершины C) для проблемной вершины.

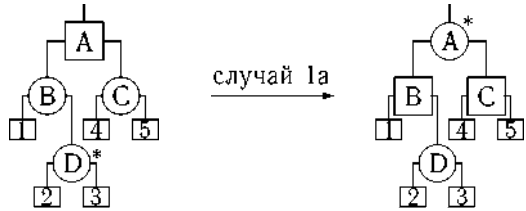


Рис. 3.3.18. Преобразования для вершины C красного цвета

В ситуации, изображенной на рисунке, выполняется только перекраска. Черный цвет «спускается» на один уровень вниз с вершины A на вершины B и C: в результате A становится красной, а B и C — черными. При этом вершина A может стать проблемной для своего родителя, но этот возможный конфликт переместился на два уровня вверх по ветви и будет обрабатываться на следующих итерациях алгоритма. Если проблемная вершина D располагается слева от вершины B (случай 1б), то перекраска выполняется абсолютно так же, без перестройки. Нетрудно убедиться, что черная длина любой ветви осталась без изменения. Если вершина A является корнем, то после перекраски вершин B и C она остается черной, т. е. черная длина дерева увеличивается на единицу. Отметим, что случай, когда корнем является вершина B (т. е. A и C не существуют), невозможен, поскольку корень имеет черный цвет.

Следующий вариант — вершина C черная — может получиться в результате очередной рекурсивной перестройки. Здесь случай правого потомка D сводится к случаю левого потомка, а последний уже перекрашивается и перестраивается.

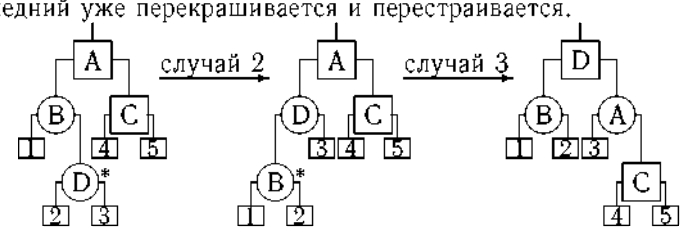


Рис. 3.3.19. Преобразования для вершины C черного цвета

После реализации случая 3 больше конфликтов не возникает, и преобразование дерева заканчивается. Дополнив эти случаи симметричными вариантами, когда конфликтная вершина находится справа от корня текущего поддерева A, мы получаем полный набор схем добавления. Трудоемкость преобразования дерева для случая 1 в худшем варианте оценивается длиной соответствующей ветви (если он будет возникать на каждом шаге рекурсивного подъема), а при реализации случаев 2 и 3 требуется конечное число операций. Учитывая утверждение о длине красно-черного дерева, получаем требуемую логарифмическую оценку.

Удаление элемента из красно-черного дерева

Удаление элемента из красно-черного дерева выполняется по следующей схеме. Сначала мы, как обычно, выполняем удаление элемента из дерева поиска: в левом поддереве вершины, которую требуется удалить, находим наибольший элемент (он будет иметь не более одного потомка) и его данные копируем в удаляемую вершину. Такая процедура не меняет имеющийся красно-черный баланс. А затем реально удаляем вершину, из которой скопировали данные: если это была концевая вершина, то заменяем ее на фиктивный черный лист; если у вершины был потомок, то он «поднимается» на ее место. Осталось учесть смену цветов. Если удаляемая вершина имела красный цвет, то ничего больше делать не надо. Если же вершина имела черный цвет, то этот цвет переносится на потомка. Если потомок был красным, то его цвет меняется на черный, и полный цветовой баланс сохраняется. Если потомка не было (вершина была концевой), то на ее месте появляется фиктивный лист «дважды черного» цвета. Если потомок имел черный цвет, то получаем «дважды черную» вершину дерева. Дальнейшие операции призваны устранить эту дважды черную покраску и привести дерево к правильному виду.

Задача 3-3-28. Постройте рекурсивную процедуру удаления элемента из красно-черного дерева поиска.

Идеи реализации. Итак, пусть некоторая вершина в дереве стала дважды черной. С учетом симметрии (слева или справа относительно корня текущего поддерева) разрешение проблемы сводится к четырем схемам. Первая схема не дает окончательного решения, но сводит ситуацию к оставшимся схемам. На последующих диаграммах мы будем обозначать дважды черную вершину двойным прямоугольником.

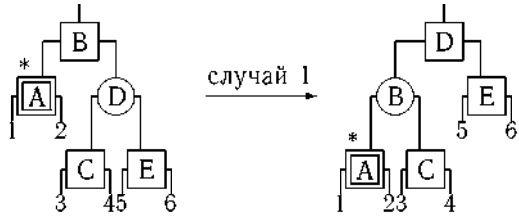


Рис. 3.3.20. Преобразование для вершины D красного цвета

Здесь цифры обозначают корни поддеревьев с произвольным цветом, фиктивные концевые вершины либо отсутствие каких-либо вершин, если сами A, C, E являются фиктивными концевыми вершинами.

Описанная выше перестройка просто меняет цвет «брата» для проблемной вершины. Если раньше проблемная вершина A имела красного брата D, то после перестройки она имеет черного брата C. А вот этот последний случай уже позволяет ликвидировать двойной черный цвет, но требует анализа цвета потомков этого черного брата.

Рассмотрим все возможные комбинации раскрасок потомков «брата» проблемной дважды черной вершины. На последующих схемах полукруг-квадрат обозначает вершину любого цвета.

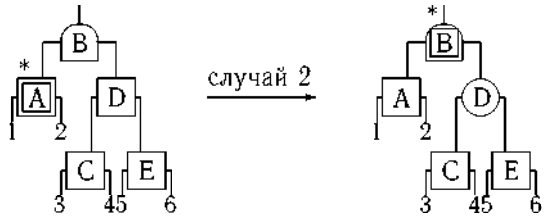


Рис. 3.3.21. Преобразование для вершин C и E черного цвета

В данном случае цвет перемещается на родительскую вершину. Если она была красной, то становится черной, и процесс завершается. Если она была черной, то становится дважды черной, и процесс поднимается на один уровень выше к корню. Если вершина B была черным корнем дерева, то лишний черный цвет ей не приписывается (т. е. она остается просто черной), а черная длина дерева в результате уменьшается на единицу.

Следующие две схемы относятся к оставшимся комбинациям цветов вершин C и E.

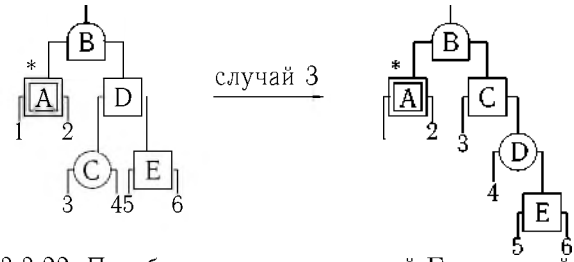


Рис. 3.3.22. Преобразование для черной E и красной C

Рассмотренная выше схема только сводит ситуацию к последнему варианту, когда вершина E красного цвета, а вершина C произвольная.

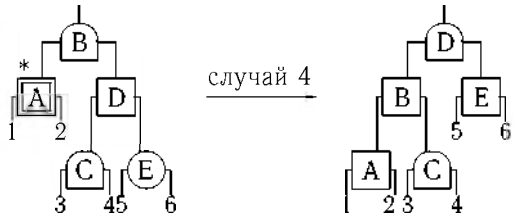


Рис. 3.3.23. Преобразование для красной E и произвольной C

В результате данного преобразования вершина C сохраняет свой цвет, а вершина D принимает цвет вершины B. После преобразования 4 никаких больше перестроек и перекрасок не требуется. Чтобы завершить описание алгоритма, нужно дополнить его симметричными вариантами для случая, когда проблемная вершина A является правым потомком B.

Теперь алгоритм удаления элемента выписывается достаточно просто. Вот его набросок.

```
1. Пусть A - текущая дважды черная вершина.
2. Если (A - корень)
       убрать лишний цвет
       и закончить.
3. Если (A слева от своего родителя)
        если (цвет D красный)
           выполнить преобразование 1
           и перейти к п. 1.
        иначе // цвет D черный
           если (цвет E черный)
              если (цвет C черный)
                 выполнить преобразование 2
```

```
                если (B дважды черная)
                   сделать B текущей
                   и перейти к п. 1.
                иначе
                   закончить.
                конец если
             иначе // цвет C красный
                выполнить преобразование 3
             конец если
          конец если
          выполнить преобразование 4
          закончить
       конец если
    конец если
4. Если (A справа от своего родителя)
       симметричный фрагмент кода
```

Нетрудно убедиться, что трудоемкость удаления также определяется лишь длиной ветви дерева и, следовательно, оценивается величиной $O(\log_2 N)$, где N — количество вершин в дереве. Внимательный анализ приведенных выше схем добавления/удаления показывает, что для реализации каждая вершина должна хранить ссылку на своего родителя. Следовательно, работу с красно-черным деревом можно организовать на основе структуры следующего вида (RB — red-black):

```
struct RBTreeNode{
  Type value;
  int color;
  RBTreeNode *left, *right, *parent;
};
```

Конечно, использование переменной типа **int** для хранения двух возможных значений цвета является излишеством. Но вряд ли с этим стоит бороться на начальном этапе изучения программирования.

— Тебя почему не взяли в первый класс?
— Спросили, сколько времен года я знаю. Ответил, что три... Мам, ну правда: «Времена года» Антонио Вивальди, Йозефа Гайдна и Петра Ильича Чайковского. Кто же там четвертый?

3.4. Графы

Граф — это пара (V, E), где V — множество вершин, а E — множество ребер (т. е. пар вершин). Каждой вершине и каждому ребру могут быть дополнительно поставлены в соответствие некоторые наборы данных, например, некоторые ребра могут быть ориентированными (имеется связь только в одном направлении) или содержать длину пути между соответствующими вершинами. В этом случае говорят о нагруженном графе, а его ребра называют дугами. Для описания графа из N вершин необходимо каким-то образом определить множества V и E. Вершины обычно нумеруют последовательностью целых чисел, т. е. $V = \{0, 1, \ldots, N-1\}$, а затем для каждой вершины указывают список ее соседей. Популярным форматом хранения графа также является матрица смежности adjMatr; элемент adjMatr[i][j] равен нагрузке ребра между i-й и j-й вершинами графа (например, единице, если имеется соответствующее ребро, и нулю иначе). Как правило, матрица adjMatr сильно разрежена и в общем случае несимметрична.

Характерная задача на графах — поиск различных путей, т. е. связных последовательностей ребер.

Задача 3-4-1. Пусть вершины графа пронумерованы целыми числами от 0 до N – 1, а конфигурация графа задана в текстовом файле в виде последовательности строк

```
<количество вершин>
<номер вершины><номер соседа>...<номер соседа>
...
```

Разработайте внутреннее программное представление графа и реализуйте функции считывания графа, печати его структуры, добавления/удаления ребер, сохранения полученного графа в файле либо в исходном формате, либо в виде матрицы смежности.

Задача 3-4-2. Пусть вершины графа пронумерованы целыми числами от 0 до N – 1, нагрузка (вес) ребра есть вещественное число, а конфигурация графа задана в текстовом файле в виде строк

```
<количество вершин>
<номер вершины> <номер соседа> <вес ребра>
...
```

Разработайте внутреннее программное представление графа и реализуйте базовые функции работы с нагруженным графом (см. задачу 3-4-1).

Задача 3-4-3. Пусть вершины графа пронумерованы целыми числами от 0 до N – 1, нагрузка (вес) ребра есть вещественное число, а конфигурация графа задана в текстовом файле в виде строк матрицы смежности

```
<количество вершин>
<вес ребра>...<вес ребра>
...
```

Разработайте внутреннее программное представление графа и реализуйте базовые функции работы с нагруженным графом (см. задачу 3-4-1).

Задача 3-4-4. Назовем путем между двумя вершинами связную последовательность ребер от первой вершины до второй. Реализуйте функцию, которая для ориентированного графа без циклов подсчитывает количество различных путей между двумя заданными вершинами. Усложните задачу, распечатав найденные пути в виде последовательности промежуточных вершин.

Указание. 1. Реализуйте естественный рекурсивный обход: от текущей вершины рекурсивно переходим ко всем соседям. Рекурсивные вызовы завершаются при достижении либо искомой (конечной), либо тупиковой вершины. Оцените сложность алгоритма (количество рекурсивных вызовов для графа из Ver вершин, у каждой из которых имеется Ed соседей).

2. Реализуйте алгоритм поиска в ширину: на первом шаге перебираем все вершины, лежащие на расстоянии одного ребра от стартовой (т. е. ближайших соседей); на втором — лежащие на расстоянии двух ребер от стартовой (т. е. ближайших соседей для соседей); и т. д. Обходы подобного типа удобно осуществлять на основе очереди, добавляя в «хвост» все найденные на текущем шаге вершины. Оцените сложность алгоритма.

3. Сравните время работы алгоритмов для различных значений Ver, Ed.

Задача 3-4-5. Реализуйте функцию, которая для произвольного ненагруженного графа и заданного M находит и печатает всевозможные пути из одной выбранной вершины до другой, содержащие не более M промежуточных вершин.

Задача 3-4-6. Назовем циклом замкнутую связную последовательность ребер (дуг) графа. Реализуйте функцию,

которая находит все циклы графа и печатает для каждого цикла соответствующую ему последовательность вершин.

Задача 3-4-7. Пусть каждому ребру графа приписан некоторый вес (нагрузка ребра — произвольное вещественное число). Назовем весом (длиной) пути сумму весов всех ребер, составляющих этот путь. Постройте функцию, которая для ориентированного графа без циклов с отрицательным весом определяет путь с минимальным весом между двумя указанными вершинами графа.

Указание. Покажите, что «жадный» алгоритм — от текущей вершины переходить к ближайшей (в смысле веса ребра) непосещенной вершине — в данной задаче не применим. Для решения задачи можно использовать алгоритм Флойда—Уоршелла (WFI-алгоритм). Запишем в матрицу distMatr[N][N] значения матрицы смежности, условно считая, что если дуга между вершинами i и j отсутствует, то distMatr[i][j] = $+\infty$. Тогда

```
for (k = 0; k < N; k++)
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
      d = distMatr[i][k] + distMatr[k][j];
      if (distMatr[i][j] > d) distMatr[i][j] = d;
    }
```

Таким образом, на k-м шаге алгоритма в ячейке distMatr[i][j] матрицы хранится длина минимального пути от вершины i до вершины j при условии, что путь может содержать только вершины с номерами $0, 1, \ldots, k-1$. Сложность алгоритма $O(N^3)$.

Задача 3-4-8. Постройте функцию, которая определяет длину пути с минимальным весом между двумя указанными вершинами нагруженного графа, все веса которого положительны (ср. с задачей 3-4-7).

Указание. Реализуйте алгоритм Дейкстры. Будем считать, что требуется найти длину кратчайшего пути от нулевой вершины до всех остальных вершин графа, заданного матрицей смежности adjMatr[N][N]. Создадим два массива len[N] и vis[N]. В len[i] будем хранить найденное к данному моменту минимальное расстояние от нулевой до i-й вершины. Массив vis является вспомогательным и содержит информацию об обработанных (посещенных) к данному моменту вершинах. В начальный момент все vis[i] равны нулю, а все len[i] хранят символ ∞

(например, произвольное отрицательное число, контролируемое отдельной веткой алгоритма). На первом шаге обрабатывается нулевая вершина: перебираются все ее соседи и расстояния до них записываются в соответствующие элементы массива len[i] вместо хранящихся там значений ∞. Затем записывается vis[0] = = 1 — нулевая вершина считается посещенной. Очередной шаг алгоритма заключается в обработке некоторой непосещенной вершины с номером k, являющейся на данный момент ближайшей к вершине с номером 0, т. е. имеющей нулевое значение vis[k] и минимальное значение len[k]. Если таких вершин несколько, то берем произвольную. Для этой k-й вершины перебираются все ее соседи и соответствующие им значения $\text{len}[i_k]$ заменяются на минимум из текущего $\text{len}[i_k]$ и суммы $\text{len}[k] + \text{adjMatr}[k][i_k]$. По сути, это означает, что путь до i_k-й вершины через текущую вершину k короче, чем было известно ранее. Затем записывается vis[k] = 1 — вершина считается посещенной. Алгоритм продолжается до тех пор, пока имеются непосещенные вершины с не равными ∞ значениями len[i], т. е. необработанные вершины, до которых можно добраться из вершины с номером 0. Приведем центральную часть алгоритма Дейкстры, считая для наглядности, что переменная **INF** означает недосягаемо большое целое число.

```
/* 1. Инициализируем массивы dist[] и vis[]: */
for (i = 0; i < N; i++) {
  vis[i] = 0;
  dist[i] = INF;
}
dist[0] = 0;
/* 2. Перебираем все вершины: если j-я вершина
является соседом нулевой вершины, то путь найден.
Далее нулевую вершину помечаем как посещенную: */
i = 0;
for (j = 0; j < N; j++) {
  if (adjMatr[i * N + j] != 0)
    dist[j] = adjMatr[i * N + j];
}
vis[0] = 1;
while (1) {
/* 3. Перебираем  все вершины: среди всех
непосещенных вершин, до которых смогли добраться,
```

```
выбираем вершину с минимальным расстоянием
до нулевой вершины: */
  imin = -1;
  dmin = INF;
  for (i = 0; i < N; i++) {
    if (vis[i] == 0 && dist[i] < INF) {
      if (dist[i] < dmin) {
        dmin = dist[i];
        imin = i;
      }
    }
  }
// 4. Если таких вершин нет, то заканчиваем:
  if (imin == -1) break;
  i = imin;
/* 5. Иначе перебираем все вершины:
если j-я вершина является соседом i-й вершины,
то заменяем dist[j], если путь окажется короче: */
  for (j = 0; j < N; j++) {
    if (adjMatr[i * N + j] != 0) {
      if (dist[j] > dist[i] + adjMatr[i * N + j])
        dist[j] = dist[i] + adjMatr[i * N + j];
    }
  }
// Вершину i помечаем как посещенную:
  vis[i] = 1;
// Возвращаемся к пункту 3:
}
```

Задача 3-4-9. Постройте функцию, которая распечатывает путь с минимальным весом от нулевой вершины до вершины с номером $N-1$ в виде последовательности посещаемых вершин (см. задачу 3-4-8).

Указание. Создадим массив way[N], положим way[0] $=0$ и заполним остальные ячейки значениями -1. Будем считать, что в way[k] содержится номер ячейки, из которой мы попадаем в ячейку k, двигаясь по кратчайшему пути из нулевой вершины. В этом случае если на очередном шаге 5 алгоритма Дейкстры изменяется значение len[j], то нужно присвоить way[j] $=$ i.

Задача 3-4-10. Разработайте форму представления плоского лабиринта в виде планарного графа и напишите программу, которая из заданной точки внутри лабиринта отыскивает путь к выходу (или говорит, что выход невозможен). Дополните программу графическим выводом результатов поиска.

Задача 3-4-11. Пусть даны два (ненагруженных) графа. Постройте функцию, которая определяет, совпадают ли эти два графа. Два графа совпадают, если они становятся тождественными после некоторой перенумерации вершин.

Указание. Полный перебор перестановок, задающих нумерацию вершин, от $[0, 1, \ldots, N-1]$ до $[N-1, \ldots, 1, 0]$ удобно реализовать на основе функции

```
int next_permutation(int *a, int N);
```

На вход функция получает массив, содержащий текущую перестановку, и его длину. Функция должна переставить элементы массива и получить следующую (в смысле лексикографического возрастания) перестановку. Возвращаемое значение — ключ, позволяющий контролировать результат работы программы. Для получения следующей перестановки

1) просмотром справа налево определяем первую пару элементов вида $a[i_0] < a[i_0 + 1]$ (отметим, что невозможно получить следующую перестановку, меняя местами между собой только элементы правее i_0);
2) среди элементов с индексами $j > i_0$ отыскиваем элемент $a[j] > a[i_0]$ с наибольшим индексом j_0 (это можно сделать методом деления пополам);
3) элементы $a[i_0]$ и $a[j_0]$ меняем местами;
4) подмассив от $i_0 + 1$ до $N - 1$ инвертируем.

Задача 3-4-12. Назовем приведением графа следующую процедуру. Если вершина a соединена ровно с двумя вершинами b и c, которые не были ранее связаны ребром bc, то она удаляется из графа, а ребра ab и ac заменяются на ребро bc. Указанная операция проводится до тех пор, пока в графе не останется вершин, которые можно было бы удалить. Реализуйте функцию, выполняющую приведение графа.

Задача 3-4-13. Назовем два графа подобными, если они совпадают после приведения (см. задачи 3-4-11, 3-4-12). Реализуйте функцию, выясняющую, подобны ли два графа.

Задача 3-4-14. Для заданного прямоугольника со сторонами Lx, Ly постройте его триангуляцию и результат сохраните в отдельном файле в следующем формате:

```
<число вершин>
<число треугольников>
<число внутренних ребер>
<число граничных ребер>
(для каждой вершины)
<номер вершины>:<x y> (координаты вершины)
...
(для каждого треугольника)
<номер треугольника>:<i j k> (номера вершин)
...
(для каждого внутреннего ребра)
<номер внутреннего ребра>:<m n> (номера вершин)
...
(для каждого граничного ребра)
<номер граничного ребра>:<m n> (номера вершин)
...
```

Указание. На первом шаге исходный прямоугольник делится на Nx×Ny равновеликих прямоугольников. На втором шаге каждый прямоугольник либо делится на два треугольника построением «северо-западной»/«северо-восточной» диагонали, либо делится на четыре треугольника проведением двух диагоналей. Полученный набор вершин и треугольников сохраняется в файл в указанном формате.

Задача 3-4-15. Написать функцию, строящую матрицу смежности по файлу, содержащему триангуляцию некоторой области в описанном в задаче 3-4-14 формате.

Локдаун, удаленное обучение, зачетная сессия, остров Бали. Студент усаживается под финиковой пальмой, открывает первую страницу скачанных конспектов:
— Та-а-ак, к базовым типам относятся: int, char, float. И, кажется, что-то еще было...
Длиннохвостая макака сверху:
— double! double! double!

3.5. Множества и контейнеры

Задачи данного раздела связаны с построением программных реализаций, предназначенных для хранения элементов данных без учета их взаимной связи. Фактически речь идет о реализации множества элементов заданного типа с общепринятыми операциями добавления элемента в множество, удаления элемента и ответа на вопрос о принадлежности данного элемента конкретному множеству.

Контейнером принято называть программную реализацию, предназначенную для размещения в памяти разнообразных наборов данных с дополнительной возможностью последующего освобождения места, занимаемого этими наборами.

Язык C++ фактически имеет встроенную реализацию контейнеров, основанную на операторах new и delete. Однако в ряде случаев использование дополнительной информации о типе размещаемых данных позволяет более эффективно организовать использование памяти. Кроме того, задача нахождения эффективного контейнерного способа работы с памятью интересна сама по себе.

Простейшую реализацию множества однородных элементов можно построить на базе массива.

Задача 3-5-1. Постройте реализацию множества элементов абстрактного типа **Type** на базе ограниченного массива при условии, что заранее задается максимальное количество элементов множества и элементы нельзя упорядочить. Для множества, содержащего N элементов, сложность поиска должна составлять в среднем O(N) операций, а сложность добавления и удаления элементов (без учета поиска) — O(1) операций.

Идеи реализации. Для реализации выделим участок памяти под максимальное допустимое количество элементов и будем размещать поступающие элементы последовательно, добавляя очередной элемент в конец цепочки. Для удаления элемента из середины цепочки просто запишем на его место элемент, стоящий на последнем месте в занятой части массива.

Задача 3-5-2. Постройте реализацию множества элементов абстрактного типа **Type** на базе ограниченного массива при условии, что заранее задается максимальное количество элементов множества и элементы можно упорядочить. Для множества, содержащего N элементов, сложность поиска должна

составлять в среднем $\log_2 N$ операций, а сложность добавления и удаления элементов (без учета поиска) — O(N) операций.

Идеи реализации. Считая, что элементы массива можно сравнивать на больше-меньше, будем поддерживать упорядоченный массив. Поиск элементов осуществляется методом деления пополам, а добавление и удаление — вставкой и удалением в упорядоченном массиве, что потребует сдвига части элементов массива.

3.5.1. Динамический массив

Ограничение на начальную длину массива можно снять, если реализовать динамический массив, в котором количество элементов может увеличиваться или уменьшаться в процессе работы. Идея реализации подобного массива состоит в следующем. Выделим блок некоторой длины и разместим элементы массива в этом блоке. Если одного блока недостаточно для размещения всех элементов массива, то выделим еще один блок и т. д. При добавлении элемента в массив этот элемент размещается в свободном участке последнего блока или в новом блоке, если последний блок не имеет свободного места. Таким образом, для доступа к элементу массива с конкретным индексом k необходимо определить блок, в котором находится этот элемент, и порядковый номер данного элемента внутри данного блока.

Задача 3-5-3. Реализуйте динамический массив элементов абстрактного типа `Type` на основе однонаправленного списка блоков следующего типа:

```
#define BLK_LEN 100
typedef struct _Block {
  struct _Block *next;
  Type elem[BLK_LEN];
} Block;
```

При увеличении количества элементов в массиве новый блок включается в этот список, а при уменьшении количества элементов последний блок исключается из списка.

Задача 3-5-4. Добавьте к предыдущей реализации функцию сортировки динамического массива (рассмотрите применение всех базовых методов сортировки) и функцию поиска заданного элемента в упорядоченном динамическом массиве методом деления пополам. Протестируйте реализацию, чтобы определить, во

сколько раз в среднем увеличивается время работы по сравнению с применением тех же алгоритмов для обычных массивов.

3.5.2. Битовая реализация

Пусть требуется реализовать работу с произвольными подмножествами множества целых чисел в диапазоне от 0 до некоторого p – 1. Выделим память, достаточную для хранения массива из p бит, и поставим в соответствие целому числу k элемент этого битового массива с индексом k. Теперь если k-й бит нашего массива равен 1, то число k принадлежит множеству, если k-й бит равен 0, то число k не принадлежит множеству. Таким образом, работа с множеством свелась к проверке или установке соответствующих битов. В стандартном языке C нет битового типа данных, поэтому нам придется организовывать работу с битами самим по аналогии с динамическим массивом. Возьмем массив четырехбайтовых целых чисел (отдельные ячейки), которые будут соответствовать блокам в динамическом массиве. В каждом таком числе размещается 32 бита. Теперь для данного конкретного числа (элемента множества) нам достаточно вычислить номер ячейки и номер бита внутри данной ячейки. Эти вычисления легко выполняются делением на 32 с остатком. Заметим, что деление на 32 (степень двойки) эффективнее выполнять сдвиговыми операциями, а вычисление остатка — побитовым логическим умножением на константу 0x1F.

Приведем одну из возможных реализаций на языке C++.

```
/* четырехбайтовая ячейка: */
typedef uint32_t Bitcell_t;
/* Bitcell - битовое множество на основе
 одного четырехбайтового числа: */
class Bitcell {
 private:
  Bitcell_t c;
 public:
  Bitcell() { c = 0; }
  ~Bitcell() {}
  void Put(int x) { c |= (1 << x); }
  void Del(int x) { c &= ~(1 << x); }
  bool Exists(int x) {
    return (bool)(c & (1 << x));
  }
```

```
  void Clear() { c = 0; }
  bool Empty() { return (bool)(!c); }
};
/* Bitset - битовое множество на основе
 массива элементов типа Bitcell: */
class Bitset {
 private:
  Bitcell *c;
  int m, mx;
  /*  проверка допустимого диапазона чисел: */
  bool Valid(int x) {
   return (x >= 0) && (x < mx);
  }
  /*  вычисление номера ячейки: */
  int cell_n(x) { return x >> 5; }
  /*  вычисление номера бита в ячейке: */
  int cell_b(x) { return x & 0x1F; }
 public:
  Bitset(int p) {
    mx = p;
    m = cell_n(n) + 1;
    c = new Bitcell[m];
  }
  ~Bitset() { delete[] c; }
  bool OK() { return if (c) ? true : false; }
  void Put(int x) {
    if (Valid(x)) c[cell_n(x)].Put(cell_b(x));
  }
  void Del(int x) {
    if (Valid(x)) c[cell_n(x)].Del(cell_b(x));
  }
  bool Exists(int x) {
    return (Valid(x)) ?
     c[cell_n(x)].Exists(cell_b(x)) : false;
  }
  void Clear() {
    for (int i = 0; i < m; i++) c[i].Clear();
  }
}
```

Задача 3-5-5. Добавьте к предыдущей реализации функцию-член, позволяющую для каждого элемента, присутствующего в множестве, выполнить указанную операцию, задаваемую указателем на функцию `void (*action)(int)`.

Задача 3-5-6. Реализуйте битовое множество на базе массива двухбайтовых ячеек.

Задача 3-5-7. Добавьте к реализации битового множества на языке C++ операции или функции, выполняющие объединение, пересечение, вычитание, дополнение множеств и т. п.

Задача 3-5-8. Составьте программу вычисления простых чисел на основе битовой реализации множества и алгоритма решета Эратосфена. Напомним, что этот алгоритм состоит в создании множества, содержащего все натуральные числа до некоторого максимального значения и последующем удалении чисел, делящихся на 2, 3, 5 и т. д. (т. е. на уже определенные ранее простые числа). Проведите тестирование соотношения (время работы)/(количество простых чисел) при использовании различных объемов оперативной памяти для хранения множества.

3.5.3. Хеш-реализация

Для реализации множества произвольных элементов можно использовать списки или деревья, поскольку эти структуры позволяют легко построить процедуры поиска требуемого элемента. Однако в ряде случаев можно ускорить поиск элемента, если применить хеширование. Суть хеширования состоит в том, что строится массив из m однотипных множеств и вводится некоторая так называемая хеш-функция `int Hash (Type x)`, возвращающая значения от 0 до m – 1. Теперь при поступлении элемента x первым делом вычисляется значение `k = Hash(x)`, и работа с данным элементом переадресуется множеству с индексом k. Если хеш-функция подобрана удачно, то мощность каждого k-го подмножества будет примерно в m раз меньше мощности всего множества, с которым мы в данный момент работаем. Таким образом, мы будем тратить в среднем в m раз меньше времени на работу с множеством (при условии, что хеш-функция вычисляется достаточно быстро). Эффективность данного подхода существенно зависит от удачного выбора хеш-функции.

Задача 3-5-9. Для заданного набора из N неповторяющихся слов (т. е. для словаря) предложите свой вариант хеш-функции с близким к равномерному распределением хеш-значений

$\rho(k)$, $0 \leqslant k < m$. Здесь $\rho(k)$ равно количеству слов в k-м хеш-подмножестве, т. е. слов, для которых $\mathrm{Hash}(x) = k$. Проверьте эффективность, построив график полученной дискретной функции $\rho(k)$, $0 \leqslant k < m$. Сравните результат с известными алгоритмами хеширования:

```
unsigned int H37(const char* str) {
  unsigned int hash = 2139062143;
  while (*str != '\0') {
    hash = 37 * hash + (unsigned int)*str;
    str++;
  }
  return hash;
}
unsigned int Rot13(const char* str) {
  unsigned int hash = 0;
  for (; *str; str++) {
    hash += (unsigned char)(*str);
    hash -= (hash << 13) | (hash >> 19);
  }
  return hash;
}
unsigned int Ly(const char* str) {
  unsigned int hash = 0;
  for (; *str; str++)
    hash = (hash * 1664525) +
           (unsigned char)(*str) + 1013904223;
  return hash;
}
```

Задача 3-5-10. Определим хеш-функцию произвольного слова x по следующей формуле:

$$h(x) = \sum_{i=0}^{n-1} \alpha_i x_i \pmod m,$$

где x_i — код i-й буквы данного слова x, α_i — некоторые заранее заданные весовые коэффициенты, m — число, определяющее диапазон значений хеш-функции, n — максимальная длина слова. Для фиксированного m и исходного словаря из N слов попробуйте экспериментально подобрать набор коэффициентов α_i так, чтобы хеш-функция наиболее равномерно рассеивала слова

по хеш-подмножествам. На данную задачу можно посмотреть как на проблему минимизации по набору коэффициентов $\{\alpha_i\}$ функции дисперсии

$$D(\alpha_1, \ldots, \alpha_n) = \frac{1}{m} \sum_{k=0}^{m-1} \left(\frac{N}{m} - \rho(k) \right)^2$$

для последовательности $\rho(k)$ (ее значение — количество слов в k-м хеш-подмножестве) и попытаться применить алгоритм нахождения минимума функции многих переменных. Если при этом словарные слова содержат только строчные буквы латинского алфавита, то можно сдвинуть диапазон кодовых значений по формуле `x_i = x_i - 'a'`.

Задача 3-5-11. Для заданного словаря исследуйте (см. задачу 3-5-10) эффективность полиномиальной хеш-функции

$$h(x) = (x_0 + px_1 + p^2x_2 + \ldots + p^{n-1}x_{n-1}) \bmod 2^{64}$$

в зависимости от выбора натурального параметра p (здесь x_i — код i-го символа строки x). Например, считая, что h(x) имеет тип `uint64_t`, а основание p взаимно просто с 2^{64}, найдите значения величин $D(1, p, \ldots, p^{n-1})$, $M = N/m$, $\min\limits_k |M - \rho(k)|$, $\max\limits_k |M - \rho(k)|$.

Замечание. Если $p = 1$, то величина

$$g(x) = x_0 + px_1 + p^2x_2 + \ldots + p^{n-1}x_{n-1}$$

равна сумме кодов символов, поэтому хеш-функция $h(x) = g(x) \bmod 2^{64}$ может оказаться малоэффективной. Если p больше всякого x_i, то вычисление g(x), по сути, соответствует переводу числа $[x_{n-1} \ldots x_1 x_0]_p$, записанного в системе счисления с основанием p, в десятичную систему счисления. В этом случае g(x) имеет уникальное значение для каждого слова x.

Задача 3-5-12. Реализуйте простейший алгоритм поиска заданной подстроки x длины n в строке s длины N со сложностью O(nN), основанный на последовательном сравнении строки $[x_0 \ldots x_{n-1}]$ со строками $[s_0 \ldots s_{n-1}], [s_1 \ldots s_n], \ldots, [s_{N-n} \ldots s_{N-1}]$. Затем выберите некоторую полиномиальную хеш-функцию h(x) (см. задачу 3-5-11) и реализуйте алгоритм поиска с предвычислением хеш-значений для $h(x_0 \ldots x_{n-1})$ и $h(s_j \ldots s_{j+n-1})$, $j = 0, 1, \ldots, N - n$. При этом подстроки

$h(x_0 \dots x_{n-1})$ и $h(s_j \dots s_{j+n-1})$ сравниваются только в том случае, если их хеш-коды совпадают.

Указание. В общем случае вычисление хеш-функции от строки длины n осуществляется за $O(n)$ арифметических действий, так как $p^0, \dots, p^{n-1}$ можно найти и заранее сохранить в отдельном массиве. Однако полиномиальность функции $h(x)$ позволяет вычислить $h(s_{j+1} \dots s_{j+n})$ при найденной $h(s_j \dots s_{j+n-1})$ за $O(1)$ действий. Единственная проблема — необходимость деления на p^k над полем 2^{64}; ее можно избежать, определив хеш-функцию в виде

$$h(x_0 \dots x_{n-1}) = (p^{n-1}x_0 + \dots + px_{n-2} + x_{n-1}) \bmod 2^{64}.$$

Задача 3-5-13. Реализуйте хеш-множество элементов некоторого типа `Type` на базе массива списков.

Идеи реализации. Пусть хеш-значение для типа `Type` вычисляется функцией `int Hash (Type x)`, возвращающей значения от 0 до $m - 1$. В рамках языка C++ решение является прямолинейным: строим класс `List1<Type>` — однонаправленный списков элементов типа `Type` (см. задачу 3-2-4) с дополнительной процедурой последовательного поиска требуемого элемента. Далее просто берем массив таких классов. Для языка C решение сводится к реализации массива списков (см. задачу 3-2-13)

Для некоторых типов задач хеш-множество удается эффективно реализовать на основе одного блока памяти, используя так называемый метод последовательных проб. Выделим массив из m элементов типа `Type` (назовем его для определенности `A`) и будем считать, что нулевое значение соответствует отсутствию элемента в множестве (это достаточно естественно, когда `Type` представляет собой некоторый указатель). В начале работы все значения `A[i]` равны нулю. Для заданного элемента `x` вычислим значение хеш-функции `k = h(x)` и разместим `x` в элементе массива `A[k]`. Если элемент `A[k]` уже хранит некоторое ненулевое значение, то ищем ближайшую свободную ячейку массива справа от `A[k]` (т. е. среди `A[k+1]`, `A[k+2]` и т. д.) и в ней размещаем наш элемент `x`. При достижении правой границы `A[m-1]` циклически переходим к началу массива. Поиск заданного элемента `x` производится аналогично: сначала проверяется соответствующий хеш-функции элемент `A[k]`, а затем, при необходимости, элементы `A[k+1]`, `A[k+2]` и т. д., до тех пор пока не будут найдены либо `x`, либо пустое место (в этом случае `x`

не содержится в множестве). Если длина массива m превосходит количество элементов N в рабочем множестве и хеш-функция выбрана удачно, то подобная процедура поиска будет с большой вероятностью завершаться после небольшого количества таких проверок (проб). Отметим, что если на первом шаге элемент `A[k]` оказался занятым, то дальнейший поиск индекса `k(j)`, $j = 1, 2, \ldots$, свободной ячейки `A[k(j)]` может осуществляться, например, по линейному `k(j) = (k + p * j) % m` либо квадратичному `k(j) = (k + p * j) % m` законам, где `p` выбирается взаимно простым с `m`.

Задача 3-5-14. Реализуйте функции добавления и поиска элементов в хеш-множестве по методу проб.

Задача 3-5-15. Разработайте процедуру удаления заданного элемента из хеш-множества, реализованного по методу проб. Учтите, что элементы с одним хеш-значением не обязательно размещаются рядом друг с другом, но между ними не будет пустых ячеек массива.

Задача 3-5-16. Реализуйте работу с множеством слов (текстовых строк) по методу проб. Сами слова размещаются в памяти с помощью функций `malloc`, а указатели на них — в хеш-множестве. Хеш-функцию постройте на основе изложенных ранее идей.

Задача 3-5-17. Пусть m — количество значений хеш-функции (длина базового массива), N — количество элементов множества, с которым мы работаем. Проведите тестирование метода проб, подсчитав среднее количество проб при поиске, добавлении, удалении элемента в зависимости от заполненности базового массива (т. е. от отношения N/m). Рассмотрите различные значения N/m (0,3, 0,5, 0,7, 0,9).

3.5.4. Контейнеры

Напомним, что контейнером называется программная реализация, предназначенная для выделения памяти под размещение некоторого набора данных с возможностью последующего освобождения этой памяти. Повысить эффективность управления памятью по сравнению со стандартными системными средствами удается за счет отказа от универсальности процедур и учета конкретных особенностей решаемых задач. Важным частным случаем является работа с однотипными элементами данных при условии, что их количество и размер a priori известны.

Задача 3-5-18. Реализуйте контейнер для элементов заданного типа `Type` на базе одного заранее выделенного блока памяти.

Решение. Введем тип данных «ячейка контейнера».

```
typedef union U {
  Type element;
  union U *next;
} Cell;
```

Зафиксируем максимальное возможное количество элементов контейнера и создадим массив из подобных ячеек. В начале работы все ячейки считаются свободными и связываются в однонаправленный список с помощью поля `next`. При запросе на выделение памяти берется первая ячейка из списка свободных и ее поле `element` используется для хранения значения размещаемого элемента. При освобождении ячейки она опять включается в начало списка свободных. Считая, что переменные, задающие контейнер, хранятся в структуре

```
typedef struct {
  Cell *cells;
  Cell *top;
} Container;
```

реализуем описанные идеи в виде функций

```
int InitContainer(Container *container,
                                int max_elem);
Type *AllocMem(Container *container);
void FreeMem(Container *container, Type *x);
void FreeContainer(Container *container);
```

предусмотрев отказ из-за недостатка памяти в выделенном блоке. Последняя функция в этом списке освобождает контейнер и возвращает весь выделенный блок в свободную системную память.

```
int InitContainer(Container *container,
                                    int max_elem) {
  int i;
  if (max_elem <= 0) return -1;
  container->cells =
              (Cell *) malloc(max_elem
                                * sizeof(Cell));
```

```
  for (i = 0; i < max_elem - 1; i++) {
    container->cells[i].next =
                          container->cells + i + 1;
  }
  container->cells[max_elem - 1].next = NULL;
  container->top = container->cells;
  return 0;
}
Type *AllocMem(Container *container) {
  Cell *next;
  Type *retval;
  if (container->top == NULL) return NULL;
  next = container->top->next;
  retval = &container->top->element;
  container->top = next;
  return retval;
}
void FreeMem(Container *container, Type *x) {
  Cell *new_top = (Cell*)x;
  new_top->next = container->top;
  container->top = new_top;
}
void FreeContainer(Container *container) {
  if (container->cells) free(container->cells);
  container->cells = NULL;
  container->top = NULL;
}
/* Далее считаем, что
typedef int Type;
число элементов контейнера n = 10;
число циклов работы с контейнером num_trials = 2;
*/
int main(void) {
  Container container;
  int **data;
  int n = 10;
  int num_trials = 2;
  int i, trial;
  if (InitContainer(&container, 10) < 0) {
    printf("Не удалось создать контейнер!\n");
```

```
    return 1;
  }
  if (!(data = (int **) malloc(
                    sizeof(int*) * (n + 1)))) {
    printf("Не удалось выделить память!\n");
    return 2;
  }
  for (trial = 0; trial < num_trials; trial++) {
    for (i = 0; i <= n; i++) {
      data[i] = AllocMem(&container);
      if (data[i]) *data[i] = i;
    }
    for (i = 0; i <= n; i++) {
      printf("i = %d; addr = %p; value = %d\n", i,
          (void*)data[i], data[i] ? *data[i] : -1);
    }
    for (i = 0; i < n; i++) {
      FreeMem(&container, data[i]);
      data[i] = NULL;
    }
  }
  FreeContainer(&container);
  free(data);
  return 0;
}
```

Задача 3-5-19. Что произойдет, если в реализациях предыдущей задачи повторно освободить ранее освобожденный элемент? Устраните возникающие при этом проблемы, добавив к предыдущей реализации битовое множество для идентификации занятых и свободных ячеек в блоке контейнера.

Задача 3-5-20. Реализуйте контейнер для элементов заданного типа `Type` на базе одного заранее выделенного блока памяти с использованием только битового множества свободных ячеек.

Идеи реализации. Для хранения данных выделяется отдельный блок на `max_elem` элементов. Дополнительно создается битовое множество, в котором единицы соответствуют занятым, а нули — свободным ячейкам блока. Свободные блоки (при добавлении данных в множество) отыскиваются

последовательным просмотром битового множества, т. е. процедурой со сложностью O(max_elem).

Задача 3-5-21. Замените в реализациях списков и деревьев задач главы 3 выделение места под элементы из свободной памяти (функциями `malloc` и `free`) на работу с контейнерами, описанными в предыдущих задачах.

Задача 3-5-22. Реализуйте контейнер для размещения в памяти элементов заданного фиксированного размера на основе динамического массива блоков.

Идеи реализации. Для хранения элементов с помощью функции `malloc` выделяется блок заранее оговоренного размера. В каждом блоке помещается фиксированное количество ячеек. В начале блока дополнительно содержится битовое множество, показывающее занятые и свободные места для элементов (каждый бит соответствует одному элементу). Свободное место в блоке определяется просмотром этого битового множества. При полной занятости данного блока выделяется новый блок, при освобождении всех элементов в данном блоке этот блок возвращается в свободную память с помощью функции `free`. При добавлении элемента нужно искать блок со свободными полями. При освобождении элемента нужно искать блок, соответствующий освобождаемому адресу. Простейший подход к реализации этих процедур поиска — связать блоки в однонаправленный список и последовательно просматривать этот список, пока не будет найден блок, включающий искомый адрес (удаление элемента) или имеющий свободное поле (добавление элемента). Другой более эффективный подход состоит в размещении адресов блоков в узлах сбалансированного бинарного дерева (упорядоченность в соответствии со значениями адресов блоков) вместе с двухбитовым полем, показывающим, есть ли в левом и правом поддеревьях блоки со свободными полями.

Задача 3-5-23. Реализуйте контейнер текстовых строк без операции удаления. Контейнер может иметь следующий интерфейс:

```
int InitStrContainer();
char *PutString(char *str);
void FreeStrContainer();
```

Функция `PutString` возвращает указатель на место размещения добавленной строки `str` либо `NULL` при отказе.

Идеи реализации. Для хранения строк с помощью функции `malloc` выделяется блок заранее оговоренного размера. Строки размещаются в блоке последовательно одна за другой (поддерживается указатель на начало свободной области в блоке). Если очередная строка не помещается в остатке блока, то выделяется новый блок и строка размещается в нем целиком (т. е. не режется между блоками). Отдельные блоки связываются в однонаправленный список, необходимый, чтобы вернуть все блоки в свободную память в конце работы.

Задача 3-5-24. Реализуйте контейнер байтовых записей произвольной длины.

Идеи реализации. Записи содержатся в общем блоке памяти. Перед каждой записью хранится длина этой записи и адреса следующей и предыдущей записей, т. е. записи образуют двусвязный список. Место для добавления новой записи в первую очередь ищется после последней размещенной записи. При отсутствии там достаточного места просматривается список и берется первый подходящий свободный промежуток. При добавлении и удалении соответственно корректируются адреса следующих и предыдущих записей в списке.

Ребенок приходит из детского сада:
— Про ноты рассказывали. Ничего не понял.
— Внутренний формат MIDI-файлов знаешь? Вот ноты — это то же самое, только на бумаге.

Глава 4
ПРОЕКТНЫЕ ЗАДАЧИ

В данном разделе предлагаются темы для реализации в минигруппах. Основная цель — научиться делить исходную задачу на отдельные подзадачи, совместно разрабатывать программный код, проводить тестирование, готовить документацию.

На этапе тестирования программы для обработки нештатных ситуаций (например, некорректных входных параметров) полезно использовать прописанный в файле `assert.h` макрос `void assert(int arg)`. При его вызове с нулевым значением переменной `arg` в поток `stderr` выдается некоторое диагностическое сообщение, обычно содержащее имя программы, имя файла, номер строки и имя функции, из которой он был вызван. Далее макрос аварийно останавливает работу программы вызовом ассемблерного кода, содержащего для процессора соответствующую инструкцию `SIGABRT`. Так как `assert()` не рекомендуется оставлять в окончательном варианте библиотеки, то большинство компиляторов автоматически отключают данный макрос при компиляции с ключом оптимизации типа `-o2`, и включают при компиляции с отладочными ключами `-o0 -g`. Отметим, что если в текст программы до оператора `#include <assert.h>` добавить `#define NDEBUG`, то макрос переопределяется следующим образом: `#define assert(condition) ((void)0)`, и, как следствие, вызов `assert()` не производит никаких действий.

При необходимости полного завершения работы программы (например, в некоторой критической ситуации) также может использоваться функция `void exit(int code)`, прототип которой прописан в заголовочном файле `stdlib.h`. В случае штатного окончания рекомендуется задавать `code=0`; любое ненулевое значение сигнализирует о наличии некорректной ситуации в программе. Отметим, что работа данной функции менее категорична: прежде чем завершится программа, будут закрыты все ранее открытые потоки и удалены вре́менные файлы.

Напомним также, что при запуске программы уже определен указатель `stderr` на стандартный поток вывода сообщений об ошибках. На этапе тестирования и отладки программы в данный поток рекомендуется выводить всю информацию о нештатных ситуациях. Например, в отдельную переменную `int key` можно записывать номер нештатной ситации и затем выводить

```
fprintf(stderr,
"File: %s Function: %s Line: %d | ErrCode = %d\n",
                __FILE__, __func__, __LINE__, key);
fflush(stderr);
```

Тогда запуск `./a.exe 2> err_a_exe.txt` обеспечит перенаправление потока сообщений об ошибках в указанный файл. При этом в функции `main` стóит запретить буферизацию данного потока, вызвав `setbuf(stderr, NULL)`.

4.1. Работа файловых систем

Работа файловой системы в любой операционной системе во многом напоминает работу контейнеров: решаются задачи размещения и удаления файлов. Однако нужно принимать во внимание, что созданный файл может изменять размер в процессе своего существования. Файловые системы обычно отводят для хранения каждого файла целое число отдельных блоков некоторой фиксированной длины. Поэтому одной из основных функций файловой системы является определение множества блоков, принадлежащих указанному файлу, а изменение размера файла сводится к добавлению или удалению блока в этом множестве.

Не касаясь проблем, связанных с реальными дисковыми операциями чтения и записи, в этом разделе мы рассмотрим контейнеры, моделирующие работу файловой системы, т. е. позволяющие изменять размер хранимого набора данных и обеспечивающие в некотором смысле доступ к информации, хранящейся в данном наборе. Будем считать, что в нашем распоряжении есть достаточно большой участок оперативной памяти, который мы по аналогии будем называть диском, а отдельный набор данных будем называть файлом. Задача состоит в построении модели файловой системы на базе такого «виртуального диска». Все файловые системы предполагают разбиение диска на несколько отдельных служебных областей, название и конкретная структура которых могут существенно отличаться для различных операционных и файловых систем. Однако в любом случае эти области содержат описание общей структуры диска, информацию, необходимую для доступа к файлам, и собственно блоки файлов. Всюду далее мы будем считать, что блоки всех файлов и пустые блоки расположены в едином массиве, т. е.

набор блоков отдельного файла определяется набором индексов в этом массиве. С каждым файлом связывается его имя, которое и служит для пользователя идентификатором этого файла.

Все приведенные ниже задачи в простейшем варианте предполагают моделирование основных функций файловой системы:

```
/* создать файл: */
int CreateFile(const char *filename);
/* удалить файл: */
int DelFile(const char *filename);
/* добавить блок в конец файла: */
int AddBlock(const char *filename);
/* удалить последний блок файла: */
int FreeBlock(const char *filename);
/* распечатать номера блоков, входящих в файл: */
int CatFile(const char *filename);
/* переименовать файл: */
int RenameFile(const char *oldname,
                              const char *newname);
/* напечатать список имен существующих файлов: */
int ListDir();
```

Возвращаемое значение этих функций — код успешности выполнения указанной операции. Дополнительно можно предложить реализацию функций

```
size_t WriteFile(const char *filename,
          size_t offset, void *src, size_t size);
size_t ReadFile(const char *filename,
          size_t offset, void *dst, size_t size);
```

выполняющих запись или чтение `size` байт с позиции файла, определяемой смещением `offset` байт от его начала. Эти функции возвращают количество реально записанных или прочитанных байтов.

Задача 4-1-1. Реализуйте модельную файловую систему на базе списков файловых блоков и битового множества свободных блоков.

Идеи реализации. Выделим на диске четыре области: Form — описание формата диска, Bit — битовое множество свободных блоков, Dir — каталог файлов, Data — массив блоков файлов.

Содержимое области Form может быть представлено следующими значениями:

```
/* Общий размер диска в байтах: */
size_t DiskSize;
/* Размеры областей на диске: */
size_t FormSize, BitSize;
size_t DirSize, DataSize;
/* Количество блоков в области Data: */
size_t NDataBlocks;
/* Длина одного блока в байтах: */
size_t BlockLength;
```

Область Bit служит для идентификации свободных и занятых блоков в области Data. Она представляет собой битовое множество.

Область Dir является каталогом файлов и содержит массив из `NDataBlocks` структур `DirEntry`:

```
typedef struct {
  char Filename[256];   // имя файла
  size_t Lenth;         // длина в байтах
  size_t FirstBlock;    // номер первого блока
} DirEntry;
```

Область Data представляет собой массив из `NDataBlocks` структур следующего вида:

```
#define BlockLength 4096
typedef struct {
  char data[BlockLength];
  size_t next;
} DataBlock;
```

Блоки одного файла связаны в однонаправленный список по полю `next`, т. е. в переменной `next` хранится номер следующего блока файла (для последнего блока положим -1).

Задача 4-1-2. Измените реализацию файловой системы из предыдущей задачи, заменив битовое множество свободных блоков на список свободных блоков.

Идеи реализации. Свяжем все свободные блоки в однонаправленный список. Таким образом мы получим «файл», состоящий из свободных блоков. Поставим в соответствие этому

файлу отдельную запись `DirEntry`, которую поместим первой в области Dir. Захват и освобождение блоков эффективно выполняется в начале этого «файла» (см. задачу 4-1-1).

Задача 4-1-3. Измените реализацию файловой системы из задачи 4-1-1, выделив ссылки между блоками файлов в отдельную область диска.

Идеи реализации. Выделим на диске четыре области: Form — описание формата диска, Fat — ссылки между блоками файлов, Dir — каталог файлов, Data — массив блоков файлов.

Содержимое области Form может быть представлено следующими значениями:

```
/* Общий размер диска в байтах: */
size_t DiskSize;
/* Размеры областей на диске: */
size_t FormSize, FatSize;
size_t DirSize, DataSize;
/* Количество блоков в области Data: */
size_t NDataBlocks;
/* Длина одного блока в байтах: */
size_t BlockLength;
```

Область Fat представляет собой массив чисел типа `size_t` — ссылок между блоками файлов: если за `k`-м блоком файла следует `m`-й блок, то значение `Fat[k]` равно `m`; если `k` — последний блок файла, то `Fat[k]` равно -1; если `k`-й блок свободен, то `Fat[k]` равно 0. Поиск свободного блока при выполнении операции добавления нового блока к файлу выполняется последовательным просмотром области Fat.

Область Dir имеет тот же вид, что и в задаче 4-1-1. Область Data теперь является просто массивом блоков длины `BlockLength` каждый.

Задача 4-1-4. Модифицируйте реализацию предыдущей задачи, включив все свободные блоки в отдельный «файл» (как в задаче 4-1-2).

Если файловая система поддерживает массив ссылок между блоками файлов, то при доступе к последним блокам придется последовательно просмотреть весь список. В реальных файловых системах это может привести к большому количеству дисковых операций чтения, что снижает эффективность работы.

Рассмотрим другой способ задания множества блоков файла. Поставим в соответствие каждому файлу структуру следующего вида:

```
typedef struct {
  /* статус структуры (занята/свободна): */
  int type;
  /* длина файла в байтах: */
  size_t Length;
  /* номера первых десяти блоков файла: */
  size_t DirectRef[10];
  /* косвенные ссылки на остальные блоки файла: */
  size_t IndirectRef[3];
} INodeBlock;
```

Номера первых десяти блоков файла будут записаны в массиве `DirectRef[10]`. Для файлов большего размера в области Data выделяется специальный блок и его номер сохраняется в `IndirectRef[0]`. В этом блоке последовательно (т. е. в формате массива `size_t DirectRef[]`) записываются номера 11-го, 12-го и т. д. блоков файла. В результате переменная `IndirectRef[0]` содержит номер служебного блока прямых ссылок, являющегося расширенным аналогом массива `size_t DirectRef[10]`. Если такого объема оказывается недостаточно, то реализуется двухуровневая ссылка: в области Data выделяется служебный блок и его номер сохраняется в `IndirectRef[1]`. Далее в этот блок последовательно записываются номера служебных блоков, содержащих прямые ссылки. С помощью `IndirectRef[2]` поддерживается третий уровень косвенных ссылок. Таким образом, элемент `IndirectRef[k]` при $k = 0, 1, 2$ можно рассматривать как корень сильно ветвящегося дерева глубины $k + 1$, в концевых вершинах которого расположены номера блоков файла.

Важными процедурами при реализации файловой системы являются захват свободного блока при увеличении длины файла и освобождение занятого блока при сокращении длины файла. Для реализации этих процедур предложим следующие идеи. Для хранения номеров всех свободных блоков выделим один специальный блок (обозначим его **FBS** — free block stack) и будем брать и добавлять номера в этом блоке по принципу стека. Если номера всех свободных блоков не помещаются в **FBS**, то дно этого стека

(первый номер в блоке) есть номер блока, который содержит следующие номера свободных блоков. При этом первый номер в этом следующем блоке есть номер блока, содержащего следующие номера, либо 0, если больше свободных блоков нет. Таким образом, блоки с номерами свободных блоков образуют список.

При запросе на выделение блока, его номер берется из **FBS**. Если **FBS** содержит только один номер, то блок с этим номером переписывается в **FBS** (т. е. **FBS** заполняется номерами свободных блоков), и после этого номер свободного блока можно опять взять из **FBS**. Если при освобождении блока с номером k оказывается, что **FBS** заполнен до конца, то содержимое **FBS** переписывается в блок с номером k, первый элемент в стеке **FBS** полагается равным k и количество элементов в **FBS** полагается равным 1.

Задача 4-1-5. Реализуйте модельную файловую систему на основе описанных выше подходов.

Идеи реализации. Выделим на диске следующие области: Superblock — описание формата диска, NameDir — каталог имен файлов, INode — массив структур типа **INodeBlock**, Data — массив файловых и служебных блоков.

Область Superblock должна содержать данные о размерах всех служебных областей, количестве блоков, размере блока, текущем и максимальном возможном количестве файлов, а также номер выделенного блока **FBS**.

Область NameDir представляет собой массив структур следующего вида:

```
typedef struct {
  char filename[256];   // имя файла
  size_t inode;         // индекс файла
} NameDirEntry;
```

Поле **filename** содержит полное имя файла, а **inode** — номер соответствующей ему структуры типа **INodeBlock** со ссылками на данные.

Область INode представляет собой массив структур типа **INodeBlock**. При этом значение поля **type**, равное 1, означает, что данная структура относится к существующему файлу, а значение 0 означает, что данная структура свободна и может быть использована для создания нового файла.

Такое раздельное хранение имени файла и ссылок на его блоки позволяет иметь несколько имен для одного и того же

файла, так как различные структуры `NameDirEntry` могут иметь одно и то же значение поля `inode`.

Задача 4-1-6. Измените реализацию файловой системы из предыдущей задачи так, чтобы каталог имен NameDir не являлся выделенной областью диска, а был специальным файлом. При этом его структура типа `INodeBlock` может быть размещена первой в области INode.

Задача 4-1-7. Модифицируйте реализации модельных файловых систем так, чтобы при создании каждому файлу можно было установить права доступа на чтение и запись: ReadOnly — файл можно только читать, WriteOnly — в файл можно только писать, NoDelete — файл нельзя удалить. Эти права доступа должны проверяться файловой системой при выполнении файловых операций внешним пользователем. Предусмотрите функции смены прав доступа к файлу.

Задача 4-1-8. Добавьте возможность импорта/экспорта файлов из реальной ОС в реализованную модельную файловую систему, а также механизм редактирования содержимого каждого файла, т. е. реализуйте простейший строковый редактор файлов, после вызова которого (**edit** filename — открыть файл на редактирование) становятся доступными следующие команды:

> **type** n1, n2 — распечатать строки файла с n1 по n2;

> **ed** n1 — заменить строку с номером n1 на вводимый далее текст, ввод текста заканчивается точкой в первой позиции очередной строки;

> **save** — записать файл;

> **quit** — выйти из редактора.

Программиста принимают на работу.
— В этой графе укажите ваш родной язык.
— Это что?
— Ну, которому вас учили с пеленок.
— А-а-а, это Basic.
— Да нет, же. Настоящий язык.
— А-а-а, ну тогда С.

4.2. Словари, базы данных

Основой алгоритмов решения задач данного раздела является прямой доступ к записям, хранящимся в (двоичном) файле

в соответствии с правилами некоторого формата. Таким образом, основной задачей реализации является поддержание целостности структуры файла (т. е. формата) и эффективное определение позиции требуемой записи в файле. Достаточно простыми примерами подобных задач являются программы проверки орфографии, а также компьютерные словари, в которых происходит поиск и выдача словарной статьи по введенному ключевому слову. Более сложный пример дают базы данных, связывающие файлы с разнообразными записями в комплексную структуру, обеспечивающую работу с данными на основе динамически формируемых запросов. Мы начнем с обсуждения нескольких простых задач.

4.2.1. Проверка орфографии

Задача 4-2-1. Пусть имеется текстовый словарный файл, состоящий из упорядоченных лексикографически слов, хранящихся по одному слову в строке. Требуется предложить новый формат хранения содержимого данного файла, позволяющий быстро считывать словарь в оперативную память компьютера и эффективно проверять правильность написания вводимых с клавиатуры слов.

Идеи реализации. Словарный файл можно хранить в памяти, например, в форме двух динамически создаваемых массивов:

```
int nW; // общее количество слов
int mS; // суммарное количество символов
        // во всех словах без учета '\0'
char *words;
int *points;
words = (char *)malloc((mS + nW) * sizeof(char));
points = (int *)malloc(nW * sizeof(char));
```

При этом `words` указывает на упорядоченные лексикографически и разделенные символами `'\0'` слова: `word1'\0'...wordnW'\0'`. В массиве `points` хранятся индексы `i1, i2,...inW` первых букв слов, т. е. `words[points[i]]` соответствует первой букве `i`-го слова. Такая структура позволяет осуществлять бинарный поиск в оперативной памяти. Далее сохраним на жестком диске в бинарном файле все данные в следующем виде:

```
nW mS word1'\0'... wordnW'\0'i1 i2 ... inW
```

В этом случае первые два целых числа из файла позволят эффективно считать с жесткого диска всю остальную информацию

с помощью функции fread(), воссоздав массивы `words` и `points` в оперативной памяти.

Задача 4-2-2. Добавьте возможность пополнения словаря новыми словами в процессе работы (например, создавая отдельное бинарное дерево для новых слов), а также функцию сохранения расширенного словаря в файл в указанном выше формате.

Задача 4-2-3. Измените формат хранения словаря в соответствии с заменой целочисленного массива `points` на массив структур

```
struct TreeNode {
  int val;  // индекс первой буквы слова
  int left; // индекс первой буквы левого слова
  int right;// индекс первой буквы правого слова
};
struct TreeNode *root =
  (struct TreeNode *)malloc(nW *
                            sizeof(struct TreeNode));
```

Тогда информация о слове, например соответствующем корню, будет записана в `root[0]`. При этом `words[root[0].val]` является первой буквой этого слова, информация о левом потомке этого слова будет храниться в структуре `root[root[0].left]`, о правом — в структуре `root[root[0].right]`.

Задача 4-2-4. Напишите программу поиска в словарном файле слов, удовлетворяющих заданному шаблону. Шаблон задается текстовой строкой, содержащей произвольные печатные символы и управляющие *подстановки*. К базовым подстановкам относятся: «?» (замещается строго один произвольный печатный символ), «*» (замещается любая, возможно пустая, последовательность печатных символов), «[abc123pq]» (замещается строго один символ из указанного набора), «[abg-n123x-zA-N7-9]» (замещается строго один символ из указанного набора с диапазонами). Например, запросу `*[-1-9]*` соответствуют все слова, содержащие либо дефис, либо цифру, а по запросу `*a???f*` надо найти все слова, в которых встречаются буквы «a» и «f», разделенные ровно тремя символами. Усложните задачу, считая, что «[!A-Zpq]» соответствует ровно одному символу, не входящему в указанный набор.

Задача 4-2-5. Напишите программу проверки правописания слов в тексте с возможностью корректировки ошибок в диалоговом режиме. Если слово в словарном файле отсутствует, то

для него находятся все «близкие», т. е. отличающиеся на одно исправление (вставку, удаление, замену одной буквы), слова.

Указание. Например, попробуйте применить алгоритм Вагнера—Фишера.

4.2.2. Толковый (двуязычный) словарь

Задача 4-2-6. Реализуйте множество слов (текстовых строк) с возможностью быстрого поиска, добавления, удаления. Продумайте вопросы оптимизации алгоритмов и методов хранения при выполнении одного или нескольких следующих дополнительных предположений:

а) известна достоверная оценка максимального количества слов в множестве, реальный объем множества близок к этой оценке;

б) количество слов, с которыми придется работать, заранее не известно;

в) в процессе работы не требуется удалять слова, они только накапливаются;

г) скорость поиска гораздо важнее занимаемой памяти;

д) требуется минимизировать память, возможно за счет скорости работы.

Идеи реализации. Во-первых, нужно решить, как размещать в памяти сами слова. Для этого можно использовать функцию `malloc` или некоторую контейнерную реализацию (см. раздел 3.5). Во-вторых, указатели на размещенные слова нужно поместить в некоторую структуру данных (список, дерево, хеш-множество и т. п.). В качестве возможных реализаций можно предложить следующие:

а) упорядоченный массив указателей с бинарным поиском и вставкой;

б) упорядоченный динамический массив указателей с бинарным поиском и вставкой;

в) дерево поиска;

г) сбалансированное дерево поиска;

д) хеш-множество на базе массива списков;

е) хеш-множество с разрешением коллизий методом последовательных проб (см. задачу 3-5-14);

ж) дерево символов (см. задачу 3-3-19).

Для каждой из реализаций оцените (теоретически и экспериментально) трудоемкость операций поиска, добавления, удаления. Оцените процент накладных расходов памяти относительно полезной загрузки памяти (т. е. суммарного количества байтов-символов во всех словах).

При реализации словаря, во-первых, следует разработать и зафиксировать формат словарных файлов, с тем чтобы программа могла работать с разными словарями при условии, что словарные файлы подчинены одному и тому же формату.

Во-вторых, нужно принять решение о том, какие данные в процессе работы должны размещаться в оперативной памяти, а какие будут по мере необходимости читаться с диска.

В-третьих, нужно понять, как организовать работу для обеспечения наибольшей эффективности при заданных ограничениях на память.

В-четвертых, нужно определить, как должен выглядеть пользовательский интерфейс программы, кем и как должны формироваться или модифицироваться словарные файлы.

Словарные файлы

Каждый элемент толкового (или двуязычного) словаря состоит из двух частей: ключевого слова и словарной статьи. Формат словарного файла должен обеспечивать программе удобный доступ и к словам (для поиска), и к статьям (для их выдачи пользователю). Будем считать, что словарный файл имеет следующий формат:

<заголовок>
<блок слов>
<блок статей>
<блок слов>
<блок статей>
...
...

Пусть заголовок имеет длину 64 байта и содержит данные о количественных характеристиках словаря:

Смещение от начала	Длина в байтах	Содержимое
0	8	идентификатор версии словаря
8	4	количество блоков слов/статей
12	4	длина одного блока слов в байтах
16	4	смещение от начала файла до первого блока слов
20	4	общее количество слов/статей
24	4	максимальная длина словарной статьи
28	4	максимальное количество слов в одном блоке
32	4	количество удаленных слов
36	4	количество удаленных статей
40	24	резерв (заполнение нулями)

Замечание. Добавление новых слов и статей легко производить в конце словарного файла. Удаление слов уже не так просто, поэтому в процессе работы целесообразно удаляемые слова только помечать, а чистку и перестройку словарного файла выполнять отдельной программой. В связи с этим мы добавили в заголовок соответствующие поля.

Для блока слов примем формат

<заголовок>
<словарная ссылка>
<словарная ссылка>
...

где заголовок имеет длину 16 байт и содержит следующее:

4 байта	признак блока слов (символы "blk#")
4 байта	абсолютное смещение от начала файла для следующего блока слов (в последнем блоке записывается нулевое смещение)
4 байта	количество слов в данном блоке
4 байта	резерв (заполнение нулями)

Словарная ссылка состоит из четырехбайтового поля задающего смещение словарной статьи данного слова от начала файла (для удаленного слова это 0), за которым идет само словарное слово с завершающим нулем.

Блок статей состоит из последовательности словарных статей. При этом каждая словарная статья начинается с четырехбайтового поля — длины данной словарной статьи, далее идет словарная статья с завершающим нулем. Первая строка статьи есть

словарное слово, последующие строки — толкование или перевод. Для удаленной статьи первый байт словарного слова есть 0.

Таким образом, считав блок слов в память, мы имеем возможность считать и соответствующие словарные статьи, поскольку нам известны их смещения относительно начала словарного файла.

Внутреннее представление словаря

Нам понадобится структура

```
typedef struct {
  /* указатель на слово в памяти */
  char *word;
  /* смещение словарной статьи в файле */
  unsigned long offset;
} DictNode;
```

Теперь, читая блоки слов из файла, мы можем заполнить подобные структуры и организовать их хранение в виде упорядоченного массива, дерева и т. п. для удобного и быстрого поиска. Найдя нужное слово, далее по смещению `offset` можно прочитать из файла словарную статью.

Экономия памяти

Для большого словаря, возможно, придется разбивать доступ к словам на несколько этапов. Одно из возможных решений следующее: считаем, что слова в словарном файле упорядочены лексикографически; размещаем в памяти упорядоченное множество, содержащее первые слова из каждого блока слов вместе со смещениями этих блоков. По этой информации определяем блок, в котором находится искомое слово и считываем этот блок слов в память; далее отыскиваем требуемое слово и его словарную статью. При таком способе доступа для каждой словарной статьи требуется два обращения к диску (в варианте с загрузкой всех слов — только одно), а также жесткое требование упорядоченности слов в словарном файле. Однако налицо существенная экономия памяти. Описанный способ несколько напоминает реализацию B-дерева (отметим, что B-деревья представляют собой весьма удобный и гибкий инструмент для подобных задач).

Изменение словарного файла

При добавлении и удалении слов и при изменении словарных статей возникает ряд проблем, связанных с необходимостью под-

держивать соответствие между данными, хранящимися в памяти, и содержимым словарного файла. В частности, при удалении слова и статьи мы не можем свободно использовать освободившиеся части файла и тем более не можем сдвигать записи в файле на другие позиции, поскольку это изменит смещения статей. Возможное решение — при редактировании словаря лишь помечать некоторые записи как удаленные, а чистку и упорядочивание словарного файла поручить другим программам. Такие служебные программы могут выполнять следующие функции:

а) формировать словарный файл на основе текстового файла, размеченного некоторым удобным для человека образом, например так:

```
word: cat
article: кошка
word: dog
article: собака
word: man
article: человек, мужчина
...
```

б) формировать размеченный текстовый файл (в смысле предыдущего пункта) на основе имеющегося словарного файла;

в) объединять два словарных файла в один новый файл с учетом лексикографической упорядоченности слов;

г) чистить словарный файл, т. е. перестраивать его таким образом, чтобы в нем отсутствовали удаленные записи и «дыры».

Замечания по программированию

Для чтения и записи данных в указанное место двоичного файла нужно открывать файл как бинарный (`"rb"`/`"wb"`), использовать функции позиционирования (fseek) и чтения/записи группы байтов (fread, fwrite).

4.2.3. Базы данных

Базой данных принято называть некоторый набор информации, содержащий, кроме собственно данных, внутренние логические связи между единицами информации. В зависимости от способа представления логических связей выделяются следующие

основные модели баз данных: иерархическая, сетевая и реляционная. Приведем основные идеи, характеризующие данные модели.

Иерархическая модель обычно используется, если информация естественно представляется в виде отношения «один ко многим»: каждый элемент базы связан с некоторым количеством элементов следующего уровня иерархии и одним элементом предыдущего уровня. Данная модель хорошо представляется в виде дерева: единицы информации хранятся в вершинах, а связи в ветвях.

Для ускорения работы можно организовать дополнительные связи внутри данного дерева. При этом мы получим новую модель, которую обычно называют сетевой. Такая модель бесспорно представляет более широкие возможности для организации эффективной работы с базой данных, однако сильно усложняет процедуру формирования эффективных запросов, так как при этом необходимо четко представлять топологию отношений. Так как каждый элемент базы в сетевой модели сам ссылается на некоторое количество элементов и произвольное количество элементов может ссылаться на него, то данную модель удобно использовать, если информация естественно представляется в виде отношения «многие ко многим».

Появление реляционной модели связывают с именем Э. Кодда (E. Codd). В 1970 году он опубликовал статью, в которой сформулировал и обосновал основные принципы построения такой модели на базе реляционной алгебры. Набор отношений между фиксированными типами элементов базы данных удобно представлять в виде таблицы. Строкой такой таблицы является некоторая последовательность элементов, каждый из которых представляет собой единицу информации. Логические отношения установлены между элементами одной строки. Набор различных строк и составляет базу данных.

При выборе структуры реляционной базы данных стараются минимизировать избыточность в логических отношениях (выделяется пять уровней нормализованности базы данных).

Модель базы данных «Курс»

Рассмотрим иерархическую модель базы данных на основе информации об учащихся I—VI курсов высшего учебного заведения. Пусть для каждого курса в базе хранятся *номера учебных групп*. Для каждой группы известны *фамилия*, *имя*, *отчество*

всех составляющих ее студентов. Про каждого студента известны его *пол*, *дата рождения*, *адрес*, *средний балл*. Требуется реализовать систему управления базой данных, позволяющую обновлять и редактировать указанную информацию, а также выдавать справки в виде таблиц или списков на основании различных запросов. Будем для простоты предполагать, что вся база может быть загружена в память. Поэтому в этом разделе мы не будем рассматривать форматы файлов с данными базы, считая, что ее можно загрузить из обычного текстового файла с минимальной разметкой.

Внутреннее представление. Основой внутреннего представления будет набор структур: однонаправленный список курсов, однонаправленный список групп каждого курса, однонаправленный список студентов каждой группы. Элементами однонаправленного списка студентов являются структуры

```
typedef struct _Student {
  char *name;
  char sex;
  Date birth;
  char *address;
  double rating;
  struct _Student *next;
} Student;
```

где

```
typedef struct _Date {
  short year;
  unsigned short month;
  unsigned short day;
} Date;
```

Элементами однонаправленного списка групп каждого курса являются структуры

```
typedef struct _Group {
  int number;
  struct _Group *next;
  Student *first;
} Group;
```

Элементами однонаправленного списка курсов являются структуры

```
typedef struct _Course {
  int number;
  struct _Course* next;
  Group* first;
} Course;
```

В результате мы имеем дерево, корневой уровень которого составляет список курсов, следующий уровень — списки групп, последний уровень — списки студентов.

Инициализирующая процедура должна читать файл с исходной информацией и настраивать базу данных.

Запросы. Для запросов к базе данных будем использовать специальный язык. Опишем ключевые понятия, слова и конструкции этого языка.

Ключевыми являются понятия шаблона, диапазона и числа.

Шаблон — текстовая строка, в которой символы «?» и «*» получают дополнительную трактовку: «?» можно заменить на любой печатный символ, а «*» означает любую последовательность печатных символов.

Диапазон — запись числового отрезка или интервала в привычном математическом виде, например: $[1,5]$, $(-3.6, 12.23)$; здесь символ «*» будет обозначать бесконечность, т. е. запись $(-*,*)$ соответствует всей числовой оси.

Число — запись конкретного целого или вещественного числа.

Перечислим ключевые слова:

`select` отмечает начало задания критериев выборки;

`endselect` отмечает конец задания выборки, необходимо осуществить выборку из базы;

`reselect` отмечает начало задания дополнительных критериев выборки из уже выбранных данных.

Параметры выборки задаются следующим образом:

`name`=*Шаблон* — нужно выбрать фамилии, имена, отчества, подходящие под указанный шаблон;

`group`=*Число* или `group`=*Диапазон* — нужно выбрать номер группы, равный данному числу или попадающий в данный диапазон.

Выборка остальных полей задается аналогично с ключевыми словами `sex`, `birth`, `address`, `rating` и с использованием

шаблонов для текстовых полей и чисел или диапазонов для числовых полей. Если для какого-либо поля критерии не заданы, то при выборке это поле не анализируется.

Приведем несколько примеров запросов.

1. Выбрать всех студентов второго курса со средним баллом от 4,5 и выше.

```
select
group=[200,299]
rating=[4.5,*)
endselect
```

2. Из полученной выборки дополнительно выбрать мужчин 1980 года рождения, фамилии которых начинаются на «И».

```
reselect
sex="м" name="И*" birth="1980.*.*" endselect
```

Задача 4-2-7. Добавьте к языку запросов возможность управления формой выдачи результатов:

`format` отмечает начало задания формата выдачи выборки;

`endformat` отмечает конец задания формата выдачи;

`print` говорит, что нужно вывести выборку в указанном формате.

Формат выдачи определяется указанием полей (тех же ключевых слов, что и при задании выборки), которые нужно вывести на печать через запятую в том порядке, в котором нужно. Например, по запросу

```
format name, rating, address endformat print
```

должен выводиться список студентов из сделанной выборки, содержащий фамилии, средние баллы и адреса.

Задача 4-2-8. Добавьте к языку запросов возможность объединения нескольких значений в правой части критериев запроса (логическая операция «или»). Например,

```
group = 201, [ 207, 208 ], 412
```

означает выборку из групп 201, 207, 208, 412.

Очевидно, что запрос типа «Найти всех студентов 201 группы, у которых средний балл выше 4,5» не потребует много времени, однако поиск информации типа «Найти всех студентов, родившихся 31 декабря» потребует обхода всего дерева.

Для ускорения работы можно организовать дополнительные связи внутри дерева, например создав бинарное дерево поиска по фамилиям студентов всего курса, двунаправленный список указателей на упорядоченные результаты по графе «средний балл» для студентов всего курса и т. п.

Модель базы данных «Студент»

Рассмотрим модель базы данных на основе информации об учащихся высшего учебного заведения. Пусть в нашем распоряжении о каждом студенте имеются следующие данные: *фамилия*, *имя*, *отчество*, *номер учебной группы*, *пол*, *дата рождения*, *адрес*, *средний балл*. Требуется реализовать базу данных, позволяющую обновлять и редактировать указанную информацию, а также выдавать справки в виде таблиц или списков на основании различных запросов. Будем для простоты предполагать, что вся база может быть загружена в память. Поэтому в этом разделе мы не будем рассматривать форматы файлов с данными базы, считая, что ее можно загрузить из обычного текстового файла с минимальной разметкой.

Внутреннее представление. Заметим, что вся рассматриваемая информация о студенте, за исключением номера группы и пола, является сугубо индивидуальной. Одинаковый номер группы и одинаковый пол имеют много студентов. Но в данном случае эти данные занимают мало места, поэтому мы можем пойти на повторное указание этой информации у каждого студента, а не группировать каким либо образом студентов одной группы или одного пола. Таким образом, основой внутреннего представления будет структура

```
typedef struct {
  char *name;
  int group;
  char sex;
  Date birth;
  char *address;
  double rating;
} Info;
```

Наша главная задача — уменьшить время обработки запросов. Поскольку наиболее частыми, по-видимому, бывают запросы по конкретному студенту или по студентам конкретной группы

(курса), то поступим следующим образом. Свяжем структуры типа `Info` в общее бинарное дерево поиска по фамилиям студентов (независимо от номера группы) и дополнительно по каждой группе организуем свой двунаправленный список. Для этого будем использовать структуры следующего вида:

```
typedef struct _Student {
  Info data;
  /* указатели дерева */
  struct _Student *left, *right;
  /* указатели списка */
  struct _Student *next, *prev;
} Student;
```

При этом поиск студента с данной фамилией удобно производить по дереву (по ссылкам `left`, `right`), а обработку группы — через список студентов группы (по ссылкам `next`, `prev`). Заметим, что, найдя данные о студенте по дереву, мы также легко получаем данные о всей его группе благодаря двунаправленному списку.

Инициализирующая процедура должна читать файл с исходной информацией и включать прочитанные данные в базу с правильной расстановкой указателей дерева и списков.

Запросы. Для запросов к базе данных можно использовать рассмотренный в предыдущем разделе язык запросов.

Модель базы данных «Расписание»

Учебное расписание представляет собой прямоугольную таблицу, в которой столбцы соответствуют учебным группам, а строки — дню и времени проведения занятий в рамках одной недели. В каждую клетку этой таблицы заносится наименование учебной дисциплины, фамилия преподавателя и номер аудитории. Требуется разработать программу, позволяющую редактировать расписание и выдавать различные справки о занятости того или иного преподавателя, аудитории, расписании отдельно взятой группы и пр.

Внутреннее представление данных. Данные, используемые в базе, представим в виде следующих структур: *список преподавателей*, *список аудиторий*, *список учебных дисциплин*, *массив номеров групп*, *массив времени занятий*, *матрица расписания*. Списки и массивы используются для хранения

информации об указанных объектах, а матрица — для связывания этой информации в единое целое (расписание). Мы выбрали массивы для хранения номеров групп и времени начала занятий, поскольку эти данные обычно не изменяются или изменяются очень редко, в то время как добавление и замена преподавателей или учебных дисциплин — это обычное явление. Минимальный набор данных, записанных в элементах этих структур может быть представлен следующими типами данных:

```
typedef struct _tli {
  char *name;
  struct _tli *next;
} TeacherListItem;
typedef struct _rli {
  int number;
  struct _rli *next;
} RoomListItem;
typedef struct _sli {
  char *subject;
  struct _sli *next;
} SubjectListItem;
typedef struct {
  int number;
} GroupItem;
typedef struct {
  int day, hour, min;
} TimeItem;
```

Таким образом, мы рассматриваем однонаправленные списки. При желании в эти элементы можно добавить дополнительную информацию, например должность преподавателя, количество мест в аудитории и т. п. Матрицу расписания можно оформить как матрицу элементов типа

```
typedef struct {
  TeacherListItem *teacher;
  RoomListItem *room;
  SubjectListItem *subject;
} MatrixItem;
```

При этом элемент этой матрицы с индексами i, j будет соответствовать j-й группе в массиве групп и i-му времени начала занятий в массиве времен.

Инициализация базы. В качестве начальных данных для заполнения базы можно взять текстовый файл, в котором последовательно перечислены необходимые данные о преподавателях, дисциплинах, аудиториях, группах и временах. На основании этой информации заполняются соответствующие списки и массивы. Далее могут идти фрагменты расписания, на основании которых заполняются некоторые элементы матрицы расписания. При инициализации расписания следует предусмотреть проверку корректности вводимых данных (существуют ли указанные имена, названия, номера в уже заполненных списках преподавателей дисциплин, групп и т. д.).

Работа с базой. Основными функциями базы являются редактирование расписания и получение справок. Эти действия целесообразно производить в интерактивном режиме. Например, можно реализовать простейшее меню с такими пунктами:

— добавить (преподавателя, дисциплину, группу, аудиторию, время);
— удалить (преподавателя, дисциплину, группу, аудиторию, время);
— редактировать (преподавателя, дисциплину, группу, аудиторию, время);
— редактировать клетку расписания;
— вывести расписание группы;
— вывести расписание преподавателя;
— вывести занятость аудитории;
— вывести аудитории, занятые в заданное время;
— вывести аудитории, свободные в заданное время;
— вывести состав преподавателей данной дисциплины.

Задача 4-2-9. Некоторые учебные занятия проводятся один раз в две недели. Модифицируйте этот проект, чтобы поддерживать в расписании и такие занятия.

Задача 4-2-10. При указанном представлении данных составление расписания отдельного преподавателя приводит к просмотру всех элементов матрицы. Организуйте дополнительную связь всех элементов матрицы, относящихся к одному преподавателю, в однонаправленный список, указатель на начало

которого хранится в элементе списка преподавателей. Это можно сделать, например, так:

```
struct _matrix;
typedef struct _tli {
  char *name;
  struct _tli *next;
  struct _matrix *start;
} TeacherListItem;
typedef struct _matrix {
  TeacherListItem *teacher;
  RoomListItem *room;
  SubjectListItem *subject;
  struct _matrix *next;
} MatrixItem;
```

Рассмотрим еще один подход к построению базы данных «Расписание». Учебное расписание в стандартном виде представляет собой прямоугольную таблицу, в каждую клетку которой заносится наименование учебной дисциплины, фамилия преподавателя и номер аудитории. Данную таблицу будем хранить в виде таблицы со следующими столбцами:

```
[room, time, name, discipline, group]
```

Так как каждому столбцу присвоено уникальное имя, то столбцы как единое целое можно переставлять. Информация при этом разрушаться не будет. Строки полученной таблицы принято называть записями, а столбцы — полями. Чтобы можно было идентифицировать записи базы данных, необходимо, чтобы любые две записи отличались хотя бы по одному полю, т. е. вся запись как единое целое должна быть уникальна. Для этого можно ввести дополнительное поле `number`, содержащее порядковый номер записи внутри базы данных.

Представление данных. Таблицу с данными будем хранить в простейшем варианте dbf-формата, работу с которым поддерживают большинство систем управления базами данных. Согласно стандарту содержимое dbf-файла состоит из заголовочной записи (содержащей заголовок и описание структуры каждого поля в записях) и собственно данных. Заголовочная запись имеет следующий формат:

Смещение от начала	Длина в байтах в байтах	Примечание
0	1	номер версии
1	3	дата последнего изменения
4	4	количество записей в файле
8	2	смещение, с которого начинаются записи
10	2	длина записи в байтах
12	20	резервные байты (конец заголовка)
32	$32 \times N$	по 32 байта на описание каждого поля записи
$32 + 32 \times N$	1	признак конца заголовочной записи (0xd)

Если размер типа int — 4 байта, а размер типа short int — 2 байта, то при работе с dbf-файлами заголовок удобно считать в структуру

```
typedef struct {
  char dbf_id; /* (0x03) без memo-полей,
                  (0x83) с memo-полями (в .dbt) */
  char last_update[3]; /* дата:
                          байт - год (две цифры),
                          байт - месяц, байт - день */
  unsigned int last_rec; /* количество
                                    записей в базе */
  unsigned short data_offset; /* смещение,
                         которого начинаются записи */
  unsigned short rec_size; /* размер каждой
              записи: сумма длин полей + 1 байт) */
  char filler[20]; /* пустые байты
                (дополняют структуру до 32 байт) */

} DBF_HEAD;
```

Здесь первый байт предназначен для идентификации dbf-файла. Если первый байт имеет значение 0x83, то dbf-файлу ставится в соответствие dbt-файл со значениями memo-полей (memo-поля имеют произвольные длину и тип и хранятся в битовом формате в отдельном файле). Если значение первого байта равно 0x03, то

файл без memo-полей, в остальных случаях файл не рассматривается как dbf-файл и работа с ним запрещается. Следующее поле длиной 3 байта содержит дату последней корректировки файла. Далее в четырехбайтовом поле в формате беззнакового целого хранится количество записей dbf-файла, включая помеченные для удаления. Используется традиционный для процессоров семейства Intel способ представления числа: младший байт всегда хранится в ячейке с младшим адресом. Следующее поле используется для хранения двухбайтового числа без знака, указывающего смещение в байтах от начала файла до первой записи. Эта информация необходима, поскольку в заголовочной части dbf-файла может содержаться описание различного количества полей. В двухбайтовое поле со смещением 10 заносится длина записи, т. е. одной строки базы данных. Значение хранимого в данном поле числа всегда на единицу больше суммы длин всех полей, поскольку в начале каждой записи имеется еще один байт для индикации удаления записи. Остальные 20 байт, начиная со смещения 12, зарезервированы для внутреннего использования. После заголовка в заголовочной записи размещено описание полей. На описание каждого поля отведено по 32 байта:

Смещение от начала	Длина в байтах в байтах	Примечание
0	11	имя поля (строка ASCII)
11	1	тип поля (C, N, L, D, M)
12	4	смещение в файле от начала записей до данного поля первой записи
16	1	длина поля в байтах
17	1	число знаков после десятичной точки в байтах
18	2	служебные байты
20	1	ID для рабочей области
21	2	служебные байты
23	1	служебный байт
24	8	резервные байты

При работе описание полей удобно считать в массив структур следующего вида:

```
typedef struct {
  char field_name[11]; // имя поля, 10 символов
```

```
  char field_type; /* тип поля: C (0x43),
        D (0x44), L (0x4C), M (0x4D), N (0x4E) */
  char offset[4]; // смещение
  union {
    unsigned short char_len; /* когда тип
                                поля не N; длина */
    struct { // когда тип поля N
      char len; // длина поля
      char dec; // число десятичных знаков
    } num_size;
  } len_info;
  char filler[14]; // дополнение до 32 байт
} FIELD_REC;
```

Первые 11 байт содержат имя поля в формате ASCII-строки. Цепочка символов замыкается нулевым байтом. Если имя поля содержит менее 11 символов, то оставшиеся байты заполняются нулевыми значениями (0x0). Следующий байт содержит ASCII-код типа поля:

Обозначение	Тип поля	Допустимые значения
C	символьный	цепочка ASCII-символов
N	числовой	знак —, цепочка цифр $0, 1, \ldots, 9$
L	логический	T, t, F, f
D	дата	ггггммдд
M	memo-поле	идентификатор записи в dbt-файле

Вслед за типом поля указывается его смещение в памяти (4 байта начиная с 12-го). Для первого поля это 1, для второго — 1 + длина первого поля и т. д.

В байте со смещением 16 хранится длина поля. Как следствие, длина поля не может превышать 256 символов. В действительности такая длина допускается только для полей символьного типа. Для числовых полей указывается количество десятичных знаков, включая десятичную точку.

Длина memo-поля всегда равна 10 байтам. В этом поле хранится в ASCII-формате номер блока dbt-файла, начиная с которого записано собственно содержимое memo-поля. Фиксированную длину имеют поля логического типа (1 байт) и поля типа «дата» (8 байт).

Следующий (семнадцатый) байт содержит количество знаков после десятичной точки для полей числового типа или имеет значение 0x0 для полей других типов. Количество знаков после точки всегда меньше длины поля.

Остальные 14 байтов предназначены для внутренних целей. Итак, в заголовочной записи dbf-файла описание каждого поля занимает 32 байта. Конец описания полей в заголовочной записи помечается кодом 0xd.

После заголовочной записи идут записи данных (т. е. собственно строки таблицы). Напомним, что длина записи указывается в заголовке dbf-файла и равна сумме длин полей плюс единица, так как в начале каждой записи содержится байт, предназначенный для маркирования удаления записи. Запись хранится в виде последовательности ASCII-кодов без разделяющих символов, благодаря чему импорт/экспорт данных осуществляется достаточно просто. Данные в отдельных символьных полях представляются в виде последовательности ASCII-кодов. Если последовательность короче длины поля, оставшиеся байты по умолчанию заполняются пробелами (код 0x20). Целые числа представляются в десятичной системе счисления и хранятся в символьном виде (ASCII-коды цифр). Допустимое количество знаков числа указано в описании поля. Для чисел, имеющих дробную часть, задается количество знаков после десятичной точки, причем сама точка также учитывается при определении длины поля. Если количество знаков в числе меньше длины поля, то ведущие позиции заполняются пробелами (например, « 9.99»). Значения логических полей представляются символами «f» или «t». Дата записывается в соответствующее поле как последовательность символов ASCII в формате «ггммдд».

Для хранения в базе данных информации, длина которой заранее не известна, используются memo-поля. Каждое memo-поле имеет длину 10 байт и в символьном виде хранит номер блока в dbt-файле, с которого начинается соответствующая информация. Подробно структуру классического dbt-файла мы рассматривать не будем. При реализации собственной модельной базы данных для него можно выбрать произвольный формат. Начальные байты memo-поля при необходимости заполняются пробелами. Если значение memo-поля состоит только из пробелов, для него не существует соответствующей текстовой записи в dbf-файле. Таким образом, все поля, независимо от их

типа, могут обрабатываться как тексты, представленные в кодах ASCII. Конец области хранящихся в таблице данных помечается кодом 0x1a (EOF). Следует учитывать, что положением данной отметки управляет не операционная система и она используется для внутренней оптимизации работы с базой данных. Дело в том, что помеченные для удаления записи остаются в файле. Как следствие, при большом количестве логически удаленных записей доступ к действительным записям при последовательном просмотре будет происходить медленно: придется просматривать как помеченные, так и не помеченные для удаления записи. В этом случае логически удаленные записи удобно переместить после метки 0x1a, что ускорит работу с данными, а при необходимости позволит их восстановить соответствующей утилитой.

Задача 4-2-11. Напишите программу заполнения базы данных, т. е. создания dbf-файла из текстового файла, содержащего в виде таблицы необходимые данные о преподавателях, дисциплинах, аудиториях, группах и временах. Например, можно считать, что записи разделены символом `'\'`, а поля разделены символом `'&'`. В начале файла должна присутствовать заголовочная строка, описывающая дальнейший формат таблицы, т. е. позволяющая заполнить поля `last_update`, `last_rec`, `field_name`, `field_type`, `len_info`.

Задача 4-2-12. Напишите программу, распечатывающую содержимое dbf-файла в текстовый файл в виде таблицы.

Задача 4-2-13. Напишите программу для создания новой базы данных и работы с имеющейся в интерактивном режиме. Реализуйте стандартные запросы языка типа miniSQL: `create`, `drop`, `insert`, `delete`, `update`, а также операторы `save` и `open`. Например:

```
create имя_базы_данных
(имя_первого_столбца
тип_первого_столбца
размер_типа,
имя_второго_столбца
тип_второго_столбца
размер_типа,
...
имя_последнего_столбца
тип_последнего_столбца
```

```
размер_типа);
drop имя_столбца from имя_базы_данных;
insert into имя_базы_данных (
имя_столбца, ... , имя_столбца
)
values (значение, ... , значение);
delete from имя_базы_данных
where имя_столбца operator значение
[...[and|or имя_столбца operator значение]...];
update имя_базы_данных
set имя_столбца = значение
[ имя_столбца = значение ],
where имя_столбца operator значение
[...[and|or имя_столбца operator значение]...];
open имя_базы_данных
save имя_базы_данных [as новое_имя ]
```

Здесь **operator** может принимать значения <, >, =, <=, >=, <>, а также допустимо использовать шаблоны, т. е. управляющие подстановки * и ?.

Работа с базой.

Задача 4-2-14. Выберите из dbf-файла записи, удовлетворяющие по заданному полю некоторому шаблону (см. задачу 4-2-4) и создайте из них новый dbf-файл.

Отметим, что реализация, например, запроса «Найти все записи с полем `name ="Efremoff"`» потребует полного перебора базы данных. Если же поддерживать упорядоченность по данному полю, переставив строки таблицы, то аналогичная проблема возникает с другими полями. Поэтому для осуществления быстрого поиска предусматривается создание так называемого индексного файла. В этом файле (его имя обычно совпадает с именем поля, для которого он строится) хранятся интересующий нас отсортированный столбец из базы данных и ссылки на соответствующие записи в исходной базе данных. Данная информация может быть представлена в виде B-дерева (либо бинарного дерева). Допускается создание мультииндексного файла для быстрого поиска по двум и более полям.

Задача 4-2-15. Напишите программу, создающую индексный файл для конкретного поля базы данных, хранящейся в dbf-формате.

Указание. В простейшем варианте в файл записываются отсортированные пары «Содержимое поля — номер записи в базе данных», что позволит осуществлять бинарный поиск по данному полю.

Задача 4-2-16. Для запросов к базе данных реализовать оператор `select`:

```
select имя_столбца [имя_столбца]
from имя_базы_данных
[ where имя_столбца operator значение
[...[ and|or имя_столбца operator значение ]...] ]
[ order_by имя_столбца [desc] ];

select name, discipline, group from Time-table
where name = "Efremoff" and group = 2* or group=3*
order_by group desc;
```

Ключ `desc` означает сортировку по убыванию.

В результате работы оператора `select` создается новая таблица с именем `Current` и ее содержимое отображается на экране.

4.2.4. Справочная система

Пусть требуется разработать и реализовать модельную гипертекстовую систему для выдачи справочной информации. В каждый момент времени при работе программы на экране отображается текущая информационная статья и список ключевых фраз, из которого пользователь может выбирать следующие статьи для просмотра. Решение этой задачи разбивается на три составные части: подготовка рабочих файлов, поиск требуемых информационных записей, интерфейсная часть. Поскольку реализация интерфейсной части сильно зависит от возможностей и квалификации программиста, мы будем считать, что выбор нужной ключевой фразы производится из некоторого меню, исходными данными для которого являются список ключевых фраз и сопутствующая информация, необходимая для поиска. В остальном мы не будем касаться этой части проекта. Рабочие данные для системы — специально подготовленный файл, формат которого обеспечивает эффективный доступ к требуемой информации. Рассмотрим возможный формат рабочего файла

с данными. Будем считать, что файл содержит заголовок и последовательность записей. Приведем формат заголовка.

Смещение от начала файла	Длина поля	Содержимое поля
0	8	идентификатор рабочего файла
8	4	длина заголовка
12	4	количество записей
16	4	максимальная длина записи
20	4	максимальное количество ключевых ссылок в записи
24	8	начало стартовой записи
32	4	длина стартовой записи
36	28	резерв

Приведем формат отдельной записи:

Смещение от начала записи	Длина поля	Содержимое поля
0	4	признак начала записи
4	4	длина данной записи
8	4	количество ссылок в записи (n)
12	8n	список смещений статей
...	...	список ключевых фраз
...	4	признак начала статьи
...	...	информационная статья

Признаки начала записи и начала статьи введены для обеспечения дополнительного контроля при чтении записи во время работы (т. е. действительно ли по указанному смещению находится запись или статья). В качестве таких признаков можно взять наборы символов, которые вряд ли встретятся в другом месте файла.

Прочитав такую запись в память, мы имеем все необходимое для доступа к последующим статьям, на которые указывают ключевые фразы (а именно смещения, определяющие местоположение соответствующих записей).

Следующий этап — подготовка рабочего файла. Рабочий файл практически невозможно создать вручную, нужна специальная программа для его подготовки. Исходные данные для такой программы могут быть сформированы в виде текстового файла с дополнительной разметкой. Например, в справочной системе

по работе с некоторым программным продуктом подобный файл может содержать последовательность записей типа

```
keyword:
    начало работы
    выполнение задания
reference:
    верхнее меню
    сохранение результата
    окно редактирования
    завершение работы
text:
    Для начала работы нажмите кнопку START в верхнем меню.
Выполнив задание, сохраните результат в файле и закройте окно
редактирования. После этого вы можете закончить работу.
```

Здесь слово `keyword` обозначает начало списка ключевых фраз для данной статьи, слово `reference` — начало списка ключевых фраз для последующих статей, `text` — начало текста статьи.

Сложность подготовки состоит в том, что, пока мы не сформируем целиком рабочий файл, мы не сможем определить истинные смещения информационных статей. Другая проблема — проверка корректности исходных данных: может оказаться, что некоторые статьи не имеют ссылок на себя, а некоторые ключевые фразы не имеют ссылок на статьи. Программа подготовки должна обнаружить подобные некорректные ситуации и выдать протокол ошибок, по которому можно подправить исходный текстовый файл с данными. При построении рабочего файла можно вначале не заполнять поля смещений статей, а вести таблицу, где записывать смещения статей, уже занявших свое место в файле, и фиксировать, в какие места файла нужно проставить смещения статей. По завершении этого предварительного форматирования файла можно заполнить поля смещений на основе построенной таблицы.

Процедура поиска и выбора статей не создает особых проблем. На основании информации, прочитанной из заголовка файла, считывается и выводится стартовая статья. Далее запись для каждой очередной статьи содержит всю необходимую информацию для формирования меню ключевых фраз и доступа к последующим статьям.

Задача 4-2-17. Реализуйте описанный проект. Дополнительно реализуйте возможность возвращения к предыдущим статьям (стек). Добавьте в меню переход на стартовую статью и завершение работы.

Задача 4-2-18. Подберите наполнение для данной системы. Интересными приложениями такого типа являются тестирующие программы (вопрос и набор ответов), игровые программы (блуждание в лабиринте). В этих вариантах посещение каждой статьи и выбор правильного ответа дают пользователю определенный набор баллов. При достижении пользователем некоторого уровня программа может перейти на другой набор статей.

4.2.5. Гипертекстовая система

Пусть требуется реализовать программу просмотра гипертекстовых документов, записанных с использованием некоторого подмножества языка HTML (HyperText Markup Language). Данный язык представляет собой совокупность достаточно простых команд (тегов), которые вставляются в исходный текстовый файл и позволяют управлять формой вывода этого документа на экран и осуществлять переходы на различные части этого файла или к другим файлам. Сами команды языка (они могут быть записаны как заглавными, так и строчными буквами) на экране не отображаются. Текст набирается в произвольном текстовом редакторе и сохраняется как обычный ASCII-файл. Определенные команды языка HTML дают возможность включать в исходный текст ссылки на другие документы, превращая исходный документ в гипертекстовый.

Для задания в тексте ссылки на другой файл (или на некоторый фрагмент текущего файла) используется так называемый якорный (anchor) тег:

```
<a href="inf.txt">Дополнительная информация</a>.
```

В этом случае фраза `Дополнительная информация` при выводе на экран выделяется каким-либо образом, и если пользователь выбирает данную ссылку, то программа начинает отображать файл `inf.txt`, расположенный в текущей директории. При необходимости можно обеспечить переход не к началу документа, а к некоторой его части. Так, например, если в файле `docs.txt` содержится строка

```
<a name="label1">Пункт 1.1</a>
```

начиная с которой следует отображать текст, то соответствующая ссылка на данную строку из произвольного файла может выглядеть так:

```
<a href = "docs.txt#label1">См. п. 1.1.</a>
```

Для перехода внутри данного файла ссылка на метку используется без указания имени файла. Так, команда

```
<a href="#label2">Начало сообщений</a>
```

означает переход к строке данного файла, помеченной тегом

```
<a name="label2">Сообщения.</a>
```

Задача 4-2-19. Реализуйте программу для просмотра гипертекстовых файлов с описанной системой ссылок.

Идеи реализации. В простейшем варианте можно ограничиться выводом на экран нескольких строк файла, начиная с выбранной метки. Для вывода последующих строк можно предусмотреть отдельную команду. Напомним, что сами теги гипертекстовой разметки выводить на экран не нужно. Способ выделения ссылок при выводе текста зависит от типа и режима работы монитора. Можно выделять ссылки цветом шрифта, подчеркиванием и т. п. Выбор следующей ссылки можно реализовать через некоторое меню либо непосредственным перемещением курсора (или указателя мыши) на требуемую ссылку.

При подготовке документа в текстовых редакторах для форматирования текста используют символы пробела, табуляции и перехода на новую строку. Программы, обрабатывающие HTML-документы, игнорируют последовательности из нескольких пробелов, заменяя их одним. Переходы на новую строку также заменяются одним пробелом. При этом форматирование текста осуществляется с учетом текущей ширины окна, типа шрифта и встречающихся внутри документа тегов. С одной стороны, это позволяет не заботиться о предварительном форматировании, с другой — внешний вид документа не совпадает с тем, который он имел при подготовке в текстовом редакторе.

Задача 4-2-20. Считая, что в начале HTML-файла прописаны параметры просмотра: длина выходной строки, размер абзацного отступа, шаг табуляции, тип выравнивания текста, добавьте в программу, отображающую HTML-файл, следующие команды.

 (break) — осуществить вынужденный перевод строки; закрывающий тег </br> не требуется (его наличие игнорируется).

** ** (paragraph) — создать новый абзац; в сравнении с **
** абзацы разделяются дополнительно пустыми строками и начинаются с красной строки; можно считать, что если закрывающий тег ** ** отсутствует, то абзац автоматически заканчивается с появлением любого из описанных далее открывающих тегов. Все рассматриваемые далее теги относятся к блочным, т. е. отсутствие закрывающего тега для них формально будем считать недопустимым.

<pre> </pre> (preformated text) — вывести заключенный в теги текст без изменения (как есть).

<h1> </h1> (header 1) — вывести заключенный в теги текст заглавными буквами.

Следующий набор тегов используется при создании ненумерованных и нумерованных списков.

**
** (unordered list) — осуществить дальнейший вывод текста с дополнительным отступом от края на шаг табуляции; каждое новое применение тега должно приводить к увеличению отступа еще на один шаг; закрывающий тег **
** отменяет действие последнего тега **<ul>**.

<ol> (ordered list) — выполнить форматирование, аналогичное тегу **<ul>**.

**
** (list item) **
** — отформатировать заключенный в теги текст как очередной пункт списка, т. е. выдать текст, начав с новой строки; в начале пункта поставить при использовании после **<ul>** определенный символ (например, звездочку), а при использовании после **<ol>** порядковый номер пункта.

4.2.6. Игры со словами

В задачах данного раздела считается, что имеется словарный файл, содержащий лексикографически упорядоченное множество различных слов. Слова хранятся по одному в строке с первой позиции. За основу построения такого файла можно взять, например, электронный словарь А. Лебедева. При решении конкретной задачи, возможно, потребуется построить новый словарный файл с наиболее подходящей структурой, обеспечивающей эффективное выполнение программы.

Задача 4-2-21. Напишите программу, создающую матрицу кроссворда. Выбирая слова из словаря, запишите их с пересечениями так, чтобы каждое слово имело не менее двух пересечений с другими словами и вся матрица помещалась

в некоторый прямоугольник размера $m \times n$. Также можно потребовать построение симметричного кроссворда.

Задача 4-2-22. Пусть имеется прямоугольная матрица размера $m \times n$, в каждой ячейке которой хранится некоторая буква. Нужно найти все слова, записанные в данной матрице. Слова могут начинаться с произвольной позиции и должны читаться по правилу: вправо — вверх (вниз) — вправо — ....

Задача 4-2-23. Напишите игру «Слово по букве». Играющий и программа дописывают в конец текущей последовательности очередную букву так, чтобы она оставалась началом некоторого слова. Проигрывает тот, кто напишет последнюю букву слова. В качестве усложнения можно разрешить дописывать букву как в конец, так и в начало имеющегося слова.

Задача 4-2-24. Напишите программу «Все слова». С клавиатуры вводится некоторое множество букв (строка). Требуется напечатать все слова, которые можно составить из имеющихся букв.

Идеи реализации. Из n букв можно составить $n + n(n-1) + \ldots + n!$ различных «слов». Если каждое проверять на наличие его в словаре, то время работы программы при больших n будет слишком велико. По данному словарю предлагается построить новый словарь, в котором каждое слово составляется из упорядоченных по алфавиту букв соответствующего слова исходного словаря и ссылки на оригинальное слово в исходном словаре. Если после упорядочивания букв два слова совпадут, то они добавляются в новый словарь как два одинаковых слова, но с разными ссылками на исходный словарь. Новый словарь в памяти хранится в виде массива списков. Внутри каждого списка слова упорядочены так, что каждое предыдущее слово содержит все буквы последующего.

Задача 4-2-25. Напишите программу, которая находит в словаре все тройки слов `Слово1`, `Слово2`, `Слово3`, такие что `Слово2` является концом `Слова1` и началом `Слова3`.

Задача 4-2-26. Найдите все слова, имеющие данное окончание.

Задача 4-2-27. Реализуйте игру «Квадрат 5×5». В начале игры на средней линии квадрата 5×5 написано некоторое слово. Программа и человек по очереди пишут букву так, чтобы при прочтении по некоторым правилам получилось новое слово. За прочитанное слово играющий получает количество очков, равное

количеству букв в этом слове. Можно зафиксировать, например, следующие правила:

1) начинать читать можно с любого места, но двигаться разрешается только вправо, вверх, вниз;
2) одну и ту же позицию нельзя читать дважды;
3) нельзя использовать как часть нового слова уже выбранные до этого слова;
4) за один ход учитывается только одно слово.

В качестве усложнения можно разрешить чтение с любого места в любом направлении (в том числе и по диагонали), т. е. по цепочке, которую может обойти шахматный король.

Далее будем считать, что слова словарного файла разделены на слоги и в них поставлены ударения: `прог-рам-'ми-ро-ва-ни-е`. Рассмотрим задачу создания квазистихотворного текста. Назовем контрольным рядом некоторую последовательность конечной длины, состоящую из двух допустимых символов **с**, **С**, например: **cccCcccCcccC**. Будем говорить, что набор слов удовлетворяет данному контрольному ряду, если каждому ударному слогу фразы соответствует символ **С**; безударному слогу может соответствовать как символ **с**, так и символ **С**.

Задача 4-2-28. Напишите функцию, формирующую по данному контрольному ряду фразу из слов имеющегося словаря.

Будем говорить, что две строки рифмуются простой рифмой, если их заключительные слоги совпадают.

Задача 4-2-29. Напишите функцию, формирующую по контрольному ряду с последним слогом типа **cCcCcCcC (но)** фразу из слов имеющегося словаря.

Рассмотрим способ задания контрольных рядов. Фиксируем следующие элементарные группы. Двудольные: **сС** — ямб, **Сс** — хорей; трехдольные: **Ссс** — дактиль, **сСс** — амфибрахий, **ссС** — анапест; четырехдольные: **Сссс**, **ссСс** — хорейные, **сСсс**, **сссС** — ямбические. Аналогично строятся пять пятидольных и шесть шестидольных групп. Главным структурным признаком квазистихотворных строк будем считать повторение (обычно 2—6 раз) элементарной группы. При этом некоторое количество первых и последних символов **с**, **С** контрольного ряда может быть отброшено. При чтении стиха в этом месте естественно возникает пауза в необходимое количество тактов. Будем говорить, что (квази)стихотворные строки образуют (квази)стихотворный текст, ес-

ли они рифмуются между собой. Отметим, что для получения «разумного» текста при подборе слов стоит учитывать части речи.

Задача 4-2-30. Написать функцию, формирующую по заданному контрольному ряду вида

Вариант 1	Вариант 2
cCcCcCcCc (ой)	**cCcCcCcCcCcCc (мо)**
cCcCcCcC (ой)	**cCcCcCcCcCcCc (ма)**
cCcCcCcCc (ой)	**cCcCcCcCcCcC (ан)**
cCcCcCcC (ой)	**cCcCcCcCcCcC (ан)**

(квази)стихотворный текст, используя слова из имеющегося словаря.

Воспитательница в детском саду: «*А теперь давайте повторим слова, которые вы должны забыть*».

4.3. Задачи на преобразование файлов

В данном разделе содержатся типичные задачи, возникающие при обработке текстовых данных. Для решения большинства из них имеются как стандартные консольные утилиты, так и готовые процедуры в специализированных пакетах программ. По сути, речь всегда идет о преобразовании информации, записанной в файле, из одной формы представления в другую: форматировании текстовых файлов, перекодировке, архивации и т. д.

Готовые исполняемые программы могут быть реализованы в следующих вариантах:

1) исходный текст берется из стандартного потока ввода (`stdin`), а результат выводится в стандартный поток вывода (`stdout`);
2) в командной строке задается имя входного файла с исходным текстом и имя файла для сохранения преобразованных данных;
3) в командной строке задается только имя входного файла и его содержимое перезаписывается в новом формате.

4.3.1. Перекодировки, фильтры, преобразования текстов

Фильтрами обычно называют программы, выполняющие преобразования последовательности байтов, поступающих со стандартного входа (`stdin`), и направляющие результат в стандартный вывод (`stdout`). К фильтрам примыкают

перекодировщики — программы, заменяющие значения байтов во входном потоке в соответствии с заданными формулами или таблицами, например программы для преобразования кириллических букв из одной кодировки в другую.

Задача 4-3-1. Напишите программы, выполняющие преобразование текстовых файлов из одной кодировки в другую. Назовите эти программы win2alt, alt2koi, win2koi и т. п. (предполагается, что программа win2alt производит преобразование файла с кодировкой Windows-1251 в файл с альтернативной кодировкой (CP866), программа alt2koi — файла с альтернативной кодировкой в файл с кодировкой KOI8-R и т. д.).

Идеи реализации. Для каждого преобразования следует создать массив (скажем, с именем `code`) из 256 целых чисел, где i-й элемент содержит код i-го символа в новой кодировке. Тогда преобразование текста файла, определяемого указателем `in_file`, выводимого в файл по указателю `out_file`, фактически сводится к простому циклу с проверкой на окончание входного файла:

```
while ((c = fgetc(in_file)) != EOF) {
  fputc(code[c], out_file);
}
```

Организуйте программу следующим образом:

1) имя входного файла задается в виде параметра командной строки;
2) программа создает временный файл, в который записывается перекодированное содержимое исходного файла;
3) исходный файл удаляется;
4) временный файл переименовывается и получает имя исходного.

Как результат выполнения программы получится файл с тем же именем, но другой кодировкой.

Задача 4-3-2. Реализуйте функцию, осуществляющую транслитерацию с кириллицы на латиницу, т. е. замену в тексте букв кириллицы буквами или сочетаниями букв латинского алфавита. Предусмотрите возможность ее использования для перекодирования комментариев в С-коде.

Задача 4-3-3. В операционных системах MS-DOS и Windows концы строк в текстовых файлах принято кодировать двумя байтами, имеющими шестнадцатеричные значения 0x0D, 0x0A. Эти

байты обычно обозначаются `'\r'` и `'\n'` и называются «возврат каретки» и «новая строка». В операционной системе UNIX концы строк в текстовых файлах кодируются одним байтом 0A. Напишите программы преобразования текстовых файлов из одной формы представления в другую, назовите эти программы dos2unix и unix2dos и организуйте их так, как предложено в задаче 4-3-1.

Задача 4-3-4. Напишите программу, которая заменяет каждый символ табуляции `'\t'` в текстовом файле на 8 пробелов. Напишите программу, выполняющую обратное преобразование.

Задача 4-3-5. Напишите программу, которая заменяет всюду в файле один заданный набор символов на другой заданный набор символов (возможно, другой длины).

Задача 4-3-6. Реализуйте функцию с заголовком

```
size_t clean(char *str);
```

которая заменяет в строке `str` каждую серию подряд идущих пробелов на один пробел. Возвращаемое значение — новая длина строки `str`.

Задача 4-3-7. Реализуйте функцию с заголовком

```
size_t clean_p(char *str, const char *p);
```

которая заменяет в строке `str` каждую серию подряд идущих символов, встречающихся в строке `p`, на один пробел. Возвращаемое значение — новая длина строки `str`.

Задача 4-3-8. Реализуйте функцию замены строчных латинских букв, содержащихся в полученной строке, на прописные.

Задача 4-3-9. Напишите функцию шифрования содержимого текстового файла с помощью шифра Цезаря (замените каждую букву на ее образ, получаемый при циклическом сдвиге алфавита на b позиций). Подготовьте обратную к ней функцию расшифровки (т. е. восстановления исходного текста при известном алгоритме и параметрах шифрования). Отдельно реализуйте функцию дешифровки (взлома) шифра (т. е. восстановления исходного текста при наличии некоторой априорной информации о методе шифрования). В данном случае для этого достаточно определить значение b, например методом грубой силы (т. е. полным перебором всех вариантов).

Указание. Если ограничиться символами ASCII-7 (шифровать только англоязычные тексты), то при $b > 0$ формула пересчета кода печатного символа `in` в новый символ `out` имеет вид

`out = (in - 32 + b) % (126 - 32 + 1) + 32`.
Расшифровка сводится к естественному обращению данной формулы.

Количество всевозможных вариантов b совпадает с мощностью алфавита, т. е. с числом используемых букв (95 в данном случае), поэтому дешифровка методом грубой силы сводится к последовательному перебору $b = 1, 2, \ldots, 94$.

Рассмотрите ситуацию, когда в тексте гарантированно содержится некоторое «заветное слово» `char *sacredWord`. Это позволит остановить перебор b при обнаружении `sacredWord` в дешифрованном тексте.

Задача 4-3-10. Напишите функцию шифрования содержимого текстового файла с помощью шифра Атбаш (замените каждую букву на ее образ, получаемый при обращении порядка следования букв в алфавите). Отметим, что для расшифровки используется та же функция.

Задача 4-3-11. Пусть задан массив `char alp[n]`, содержащий коды допустимых для записи сообщения символов. Напишите функцию шифрования содержимого текстового файла с помощью аффинного шифра: каждая буква `alp[i]` исходного сообщения заменяется на букву `alp[j]`, где `j = (a * i + b) % n`. При этом положительные целочисленные коэффициенты `a`, `b` заданы и `НОД(a,n) = 1`. Отдельно реализуйте функцию расшифровки и предложите метод дешифровки подобного шифра.

Указание. Дешифровка методом грубой силы, т. е. перебором параметров `a` и `b`, может занять слишком много времени. В данном случае качественное ускорение дают модификации типа «угадай слово» и/или «угадай букву». Нужно провести частотный анализ зашифрованного текста: вычислить частоты встречаемости отдельных букв, сочетаний из двух-трех букв, возможно сочетаний из нескольких слов, и сравнить их с соответствующими частотами естественного языка (для осмысленных текстов такие характеристики устойчивы).

Задача 4-3-12. Пусть заданы массив `char alp[n]`, содержащий коды допустимых для записи сообщения символов, и вспомогательный массив `char b[m]`, хранящий величины сдвигов алфавита (см. задачу 4-3-9). Напишите функцию шифрования содержимого текстового файла с помощью шифра Виженера—Цезаря с бегущим ключом: буква `alp[i]`

с порядковым номером k = 0, 1, 2, ... в исходном сообщении заменяется на букву alp[j], где j = (i + b[k%m]) % n. Для удобства элементы массива b можно задать ключевым словом, например char b[] = "SUBWAY";.

Указание. По сути, данный метод является m-алфавитным методом кодирования. Каждый сдвиг порождает свой алфавит, с помощью которого кодируются буквы в цепочке с номером k % m исходного сообщения. В результате одинаковые буквы могут кодироваться разными символами, что размывает частотные характеристики и усложняет дешифровку. Для дешифровки можно провести частотный анализ прореженного текста, беря каждую вторую букву, каждую третью и т. д. Если частотный анализ дает неравномерное распределение букв, то считаем, что длина m сдвига найдена. Далее подбираем сдвиг в каждой цепочке.

Отметим, что если в последовательности сдвигов нет закономерностей и ее длина сравнима с длиной сообщения, то взлом шифра в общем случае является некорректной (неразрешимой) задачей.

4.3.2. Помехоустойчивое кодирование

Рассмотрим простейшие приемы, позволяющие однозначно восстановить пересылаемую (например, по сети) информацию при искажении некоторых битов. Ключевая идея излагаемых подходов заключается в добавлении в передаваемое сообщение специальных контрольных (проверочных) символов.

При решении следующих задач необходимо написать:

— функцию кодирования

```
long int text2code(char *outcode,
  unsigned long int *outbits,
  const char *instr,
  const unsigned long int inbits);
```

модифицирующую (в соответствии с указанным алгоритмом) входную последовательность instr, содержащую inbits бит, в последовательность outcode, содержащую outbits бит,

— обратную к ней функцию декодирования

```
long int code2text(char *outstr,
     unsigned long int *outbits,
     const char *incode,
```

```
    const unsigned long int inbits);
```

— и функцию,

```
int code2errcode(char* str,
  const unsigned long int bits);
```

имитирующую внесение случайных ошибок на этапе передачи информации.

В простейшем варианте функция `code2errcode()` может изменять с некоторой вероятностью p каждый бит входной строки `str` на противоположный.

Задача 4-3-13. (Байтовое кодирование с повторением.) Каждый байт входной строки превращается в последовательность из трех равных ему байт выходной строки. При декодировании функция `code2text()` заменяет каждую последовательность из трех входных байт на один байт выходной строки по правилу: если два байта из трех совпадают, то выдаем совпадающее значение, иначе тройка считается «битой» и возвращаем код ошибки (например, отрицательное число, равное количеству правильно восстановленных символов).

На этапе тестирования работы алгоритма для наглядности можно при декодировании заменять «битые» тройки некоторым фиксированным байтом (например, `*`), считая, что в исходном тексте такой символ отсутствует.

Задача 4-3-14. (Байтовое кодирование с контролем четности.) Будем считать, что младшие семь бит каждого байта входной последовательности содержат семибитный код ASCII-7 передаваемого символа. Тогда старший бит, называемый в этом случае битом четности, можно использовать для контроля правильности каждого передаваемого байта: если сумма первых семи бит является нечетным числом, то функция `text2code()` должна установить старший бит в выходном байте равным единице, иначе — равным нулю. В результате каждый байт закодированной последовательности будет иметь четную сумму битов. При декодировании функция `code2text()` проверяет четность суммы битов каждого входного байта и в случае четной суммы выдает байт со старшим нулевым битом, а в случае нечетной суммы прекращает работу и возвращает код ошибки, например отрицательное число, равное количеству правильно восстановленных байтов. Если же заменять каждый «битый»

байт на некоторый недопустимый символ (например, *), то можно изучить влияние функции `code2errcode()` на весь процесс.

Байтовое кодирование с контролем четности позволяет обнаружить любую одиночную ошибку — подобный подход реально применялся при передаче данных из семибитной ASCII-таблицы (в случае ошибки запрашивалась повторная передача символа).

Задача 4-3-15. При передаче данных по сетевым каналам некоторые системные программы предполагают, что старший бит каждого байта равен 0, и используют его как бит контроля четности. Это может привести к искажению информации, если входные данные такому условию не удовлетворяют. Поэтому разработаны специальные программы uuencode и uudecode для предварительной кодировки и последующей декодировки передаваемой информации. Реализуйте аналоги указанных программ.

Идеи реализации. Каждые три байта входного потока делятся на четыре группы по шесть бит. К каждой такой группе приписываются два старших бита 01, и полученные четыре байта выдаются в выходной поток. Поскольку длина исходного файла не обязательно кратна 3, его можно дополнить одним или двумя байтами и в начале закодированного файла передать общее количество байтов в исходном файле. Обратное преобразование выполняется очевидным образом.

Задача 4-3-16. (Битовое кодирование с повторением.) Каждый бит входной строки превращается в последовательность из n равных ему бит выходной строки. При декодировании функция `code2text()` заменяет каждые n входных бит на один бит выходной строки по следующему правилу: если во входной последовательности содержится больше нулей, чем единиц, то выдаем ноль, иначе — единицу.

Задача 4-3-17. (Битовое кодирование с двойным контролем четности.) Выберем некоторое целое n, разобьем входную строку на последовательности битов длины n^2 и для каждой такой последовательности $a_1, \ldots, a_{n^2}$ построим вспомогательную расширенную матрицу

$$\begin{pmatrix} a_1 & \ldots & a_n & b_1 \\ \ldots & \ldots & \ldots & \ldots \\ a_{n^2-n+1} & \ldots & a_{n^2} & b_n \\ c_1 & \ldots & c_n & d \end{pmatrix}.$$

Здесь добавочные биты $b_1, \ldots, b_n$ выбираются так, чтобы сумма битов в каждой строке матрицы была четной; биты $c_1, \ldots, c_n$ выбираются так, чтобы сумма битов в каждом столбце матрицы была четной. Несложно показать, что бит четности строки c равен биту четности столбца b, поэтому элемент d определяется корректно. Полученная матрица является закодированным аналогом входной последовательности $a_1, \ldots, a_{n^2}$. Покажите, что при декодировании функция `code2text()` может не только обнаружить, но и исправить любую одиночную ошибку, а также обнаружить произвольную двойную ошибку. Данный подход естественным образом модифицируется для входной последовательности, состоящей из $n_1 n_2$ бит. В этом случае выходная последовательности будет иметь размер $(n_1 + 1)(n_2 + 1)$ бит.

4.3.3. Форматирование текстов

Задача 4-3-18. Пусть заданы положительное целое n и набор символов-разделителей. Напишите программу, которая преобразует длинные строки текстового файла в более короткие, содержащие не более n символов. Разрыв строки может выполняться только на символах из заданного набора (например, на пробелах и табуляциях). Если подобное разбиение невозможно, то программа выдает информационное сообщение и строка остается без изменения.

Задача 4-3-19. Назовем абзацем последовательность строк, не содержащую внутри себя пустых строк (т. е. пустые строки разделяют абзацы). Напишите программу, которая собирает каждый абзац текстового файла в одну длинную строку.

Задача 4-3-20. Допустим, что вид абзаца текста определяется тремя параметрами: позицией (т. е. порядковым номером символа) левой границы абзаца, позицией правой границы абзаца, количеством пробелов в начале первой строки абзаца (абзацном отступе). Напишите программу, которая форматирует абзацы текстового файла в соответствии с данными значениями этих трех параметров без разбиения (переноса) слов. Строки абзаца должны быть выровнены по левой и правой границе. Предполагается, что между словами можно оставлять промежутки любой длины, а в случае невозможности заполнить строку выравнивание выполняется по левой границе в ущерб правой.

Задача 4-3-21. Решите предыдущую задачу, если разрешается переносить слова по слогам.

Идеи реализации. Пусть «x» обозначает любую согласную, «o» — любую гласную, «*» — любой набор символов (в том числе и пустой), «+» — любой непустой набор символов, «-» — позицию переноса. Для построения разбиения слова для переноса можно использовать следующие правила.

1. Буквы «ь», «ъ» составляют единое целое с предшествующей согласной.
2. Буква «й» составляет единое целое с предшествующей гласной.
3. Нельзя переносить одну букву.
4. Слово можно разделить по паре гласных, если разбиение имеет вид *xo-o+.
5. Слово можно разделить по паре согласных, если разбиение имеет вид *ox-x*o*.
6. Слово можно разделить, если разбиение имеет вид *xo-xo*.
7. Слово можно разделить, если разбиение имеет вид +o-*o*.

Указанные правила выстроены по приоритету. В частности, правилом 7 следует пользоваться только тогда, когда не удается разбить слово по правилам 4—6.

Задача 4-3-22. Напишите программу, форматирующую абзацы текстового файла в соответствии с заданной разметкой. Под разметкой будем понимать следующее. Если в начале очередной строки стоит запись вида `\{a,b,c}`, то это означает, что все абзацы, начиная со следующей строки, должны форматироваться с левой границей в позиции `a`, правой границей в позиции `b` и красной строкой в `c` пробелов. Аналогичная разметка вида `\local{a,b,c}` относится только к одному следующему абзацу, после которого восстанавливаются предыдущие параметры форматирования.

Задача 4-3-23. Отформатированный абзац с ровной правой границей, но большими промежутками между словами часто выглядит менее красиво, чем абзац с неровной правой границей, но небольшими и равномерно распределенными межсловными промежутками. Придумайте способ вычисления меры «неровности» правой границы x и меры «неравномерности» распределения межсловных промежутков y. Введите коэффициенты штрафов a_x, a_y и реализуйте форматирование абзаца таким образом, чтобы общий штраф $f = a_x x + a_y y$ был минимальным. Посмотрите, как меняется результат форматирования в зависимости от значения коэффициентов штрафа a_x, a_y.

Задача 4-3-24. Пусть каждый печатный символ имеет определенную ширину, выражаемую целым числом (эти числа записаны в отдельном массиве). Модифицируйте программу форматирования (см. задачу 4-3-20) в предположении, что параметры абзаца задаются в единицах измерения ширины символов.

Задача 4-3-25. Рассмотрим задачу форматирования страницы с иллюстрациями. Пусть заданы размеры страницы (количество строк и количество символов в строке) и некоторое количество запрещенных прямоугольных областей (мест для иллюстраций). Требуется реализовать программу форматирования так, чтобы абзацы текста размещались в разрешенных участках страницы и «огибали» запрещенные области. Запрещенные области могут задаваться позициями левой и правой границ по горизонтали и диапазоном номеров строк по вертикали. Текст, не поместившийся на страницу, можно игнорировать.

Задача 4-3-26. Рассмотрим еще один способ выделения места для иллюстраций в тексте. Пусть заданы размеры страницы (количество строк и количество символов в строке) и в тексте имеется разметка размещения иллюстраций следующего вида: `\leftpic{a,b}`, `\midpic{a,b}`, `\rightpic{a,b}`, где `a` — ширина иллюстрации (количество символов по горизонтали), `b` — высота иллюстрации (количество строк по вертикали), `\leftpic` и `\rightpic` говорят о том, что иллюстрация должна быть прижата к левому или к правому полю страницы соответственно, `\midpic` — что иллюстрация должна располагаться строго посередине. Требуется реализовать форматирование текста так, чтобы иллюстрация размещалась целиком на странице и располагалась по возможности ближе к тому месту текста, где была размещена команда разметки этой иллюстрации.

Задача 4-3-27. Пусть даны написанные на языке C/C++ тексты двух функций `f()` и `g()`, правильно решающих некоторую задачу. Требуется оценить подобие текстов и сделать вывод, можно ли считать текст функции `f()` «переформатированным» вариантом текста функции `g()`.

Идеи реализации. При решении данной задачи удобно выделить характерные типы модификаций, которые используются для изменения внешнего вида программы с минимальными усилиями, а затем для каждого из них придумать алгоритм, устраняющий подобные изменения:

а) текст `f()` получен из текста `g()` переформатированием кода и переименованием переменных — исключим из обеих функций разделительные символы, т. е. пробелы, табуляции, символы новых строк, комментарии, а также имена переменных;

б) в тексте `f()` изменен интерфейс — исключим операторы ввода-вывода, строковые константы, а также блоки, отвечающие за описание переменных;

в) в тексте `f()` некоторые вычислительные операторы «расщеплены» на несколько — попытаемся объединить идущие подряд арифметические выражения в одну строку.

Полученные в результате «обезличенные» коды `F()` и `G()` сравним Linux-утилитой `diff` либо некоторым самостоятельно реализованным аналогом.

В качестве иного подхода можно рассмотреть статистический метод. Основная идея заключается в следующем: выделим в тексте ключевые объекты. Для литературного текста это могут быть имена людей, названия местностей, ссылки на других авторов и т. д. При разборе С-функций естественно выделить имена функций и переменных, ключевые слова языка. Каждому объекту присваивается формальное имя и некоторая количественная величина: частота использования для имен переменных и для имен функций, количество операторов цикла и конструкций типа **if–else** и т. д. При сравнении текстов функций `f()` и `g()` запишем порядок использования ключевых объектов. Будем считать, что две функции близки, если соответствующие их текстам последовательности использования формальных имен подобны.

4.3.4. Сжатие и архивация

Программы-архиваторы преобразуют исходные данные в более компактную форму, используя информационную избыточность кодируемой информации. Для этого проводится статистический анализ входной последовательности байтов и повторяющиеся наборы заменяются более короткими кодами. Поскольку алгоритмы сжатия, как правило, ориентированы на работу с байтами, мы будем называть байты входного потока символами.

Групповое кодирование RLE

RLE-алгоритм (Run-Length Encoding) в процессе кодирования заменяет группы одинаковых входных символов на пару

(**количество**, **символ**). Например, строка «aaaabbbbbc» будет закодирована в последовательность 4a5b1c. Сжатие будет эффективным, если во входном потоке много длинных цепочек из одинаковых символов. В реальных алгоритмах применяются различные модификации кодирования RLE, позволяющие сократить накладные расходы при отсутствии групп из повторяющихся символов.

Задача 4-3-28. Реализуйте модельные кодировщик и декодировщик файлов на основе следующего варианта кодирования RLE: каждая повторяющаяся последовательность из одного и более символов заменяется на пару (**число повторений** (четыре байта), **символ** (один байт)). Улучшите алгоритм, выделяя для числа повторений один байт; в этом случае последовательности из более чем 255 повторяющихся символов делятся на части.

Задача 4-3-29. Реализуйте кодировщик и декодировщик файлов на основе следующего двухбайтового варианта кодирования RLE. Для повторяющихся групп длиной от 1 до 127 символов выходным кодом являются два байта: **число повторений**, **символ**. Для групп из n неповторяющихся символов длиной от 1 до 127 байт выходным кодом являются $n + 1$ байт: первый байт содержит **число символов** в формате $128 + n$, далее идет исходная группа из n неповторяющихся символов. Более длинные группы в обоих случаях разбиваются на части, и каждая часть кодируется отдельно. При декодировании считываем первый байт в переменную **unsigned char** `n`. Если старший бит `n` равен нулю, то считываем следующий байт в переменную **unsigned char** `c` и `n` раз дублируем `c` в выходной поток. Если старший бит `n` равен единице, то следующие `n-128` байт входного потока без изменений передаем в выходной поток. Далее процесс повторяется.

Задача 4-3-30. Реализуйте кодировщик и декодировщик файлов на основе следующего трехбайтового варианта кодирования RLE. Выходным кодом для повторяющихся групп из четырех и более символов являются три байта: **флаг**, **количество**, **символ**. Флаг — это некоторое выделенное значение (например, 255), которое редко встречается во входном потоке. Группы неповторяющихся символов передаются на выход как есть, при этом во избежание путаницы сам символ с кодом 255 кодируется тройкой $(255, 1, 255)$.

Задача 4-3-31. Базовый вариант RLE-кодирования хорошо работает для сжатия черно-белых изображений, строки которых представляются длинными последовательностями черных и белых пикселов. Реализуйте RLE-кодировщик и декодировщик битового уровня.

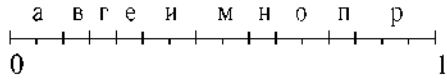
Идеи реализации. В данном случае входным потоком является последовательность битов, кодирующих два возможных цвета пиксела изображения. Выходным кодом является байт, в котором старший бит определяет цвет пиксела, а оставшиеся семь младших бит дают количество повторений.

Арифметическое кодирование

Суть арифметического кодирования состоит в том, что цепочке символов ставится в соответствие некоторая (конечная) двоичная дробь, которая строится на основе информации о частоте появления каждого символа в исходном файле. Рассмотрим идею алгоритма на примере кодирования слова «программирование». Частотный анализ дает следующий результат:

Символ	Количество	Частота
а	2	$0{,}001_2$
в	1	$0{,}0001_2$
г	1	$0{,}0001_2$
е	1	$0{,}0001_2$
и	2	$0{,}001_2$
м	2	$0{,}001_2$
н	1	$0{,}0001_2$
о	2	$0{,}001_2$
п	1	$0{,}0001_2$
р	3	$0{,}0011_2$

Разобьем отрезок $[0, 1]$ на пропорциональные этим частотам части:



Пусть элементы массивов `lb[i]` и `rb[i]` задают границы этих частей для символа с кодом `i`. Опишем алгоритм построения кодирующей дроби.

```
left = 0;
right = 1;
while ((k = NextSym()) != EOF) {
  d = right - left;
  right = left + d * rb[k];
  left = left + d * lb[k];
}
```

По окончании этого цикла в качестве кодирующей дроби можно взять любое число **x**, удовлетворяющее условию **left** $\leqslant$ **x** $<$ $<$ **right**. Кодирование, по сути, представляет собой построение системы вложенных отрезков, где каждый следующий отрезок занимает в предыдущем место, отведенное данной букве в исходном разбиении отрезка $[0, 1]$. Для обозначения конца кодируемой информации можно дополнительно передавать общее число исходных символов либо заканчивать входной поток специальным символом (маркером) конца файла **EOF**. Для слова «программирование» первые несколько отрезков имеют в двоичном представлении следующий вид.

Буква	left	right
п	0,1100	0,1101
р	$0{,}11001101_2$	$0{,}11010000_2$
о	$0{,}110011101111_2$	$0{,}11001111010_2$
г	$0{,}1100111011110001_2$	$0{,}1100111011111100_2$

Окончательные значения границ **left** и **right** соответственно равны

$$0{,}11001110111101101111000111110101110010011111111110001,$$

$$0{,}11001110111101101111000111110101110010011111111110010.$$

В результате исходная 16-байтовая строка сжалась до 51 бита.

Декодирование производится обращением описанного выше процесса.

```
x = <кодирующая дробь>
n = <общее число символов в исходном файле>
while (n--) {
/* найти номер интервала, содержащего x: */
  k = Interval(x);
/* вывести k-й символ: */
```

```
  Output(Symbol(k));
/* длина интервала для k-го символа: */
  d = rb[k] - lb[k]
/* преобразуем дробь для определения
   следующего символа: */
  x = x - lb[k];
  x /= d;
}
```

Заметим, что для выполнения кодирования и декодирования следует сначала определить таблицу частот символов в исходном файле. Таким образом, кодировка требует двух просмотров файла, а для декодирования нужна таблица частот, которую приходится помещать в сжатый файл.

Задача 4-3-32. Реализуйте кодировщик и декодировщик файлов по алгоритму арифметического кодирования.

Идеи реализации. Следует заранее задать ограничение на длину двоичного представления кодирующей дроби (например, 6 байт). Вычисление дроби ведется до тех пор, пока различие между битами двоичного представления границ `left` и `right` находится в пределах этого ограничения. Как только различие между представлениями выйдет из заданных границ, полученная дробь выводится в выходной поток и начинается формирование дроби для следующих символов.

Задача 4-3-33. Реализуйте алгоритм адаптивного арифметического кодирования, не требующий передачи таблицы частот. Он основывается на следующей идее. В начальный момент таблица частот инициализируется значениями 1/256 (т. е. все символы считаются равновероятными). Очередной символ обрабатывается в соответствии с таблицей, после чего таблица корректируется с учетом поступившего символа. Аналогично при декодировании таблица частот корректируется после декодирования каждого выходного символа.

Алгоритм Хаффмана

Алгоритм Хаффмана, иногда его также называют кодированием CCITT (Comité Consultatif International Télégraphique et Téléphonique), заключается в замене каждого символа входного потока на некоторую последовательность битов переменной длины, причем часто встречающиеся символы кодируются более

короткими последовательностями. Для поиска таких замен строится бинарное дерево Хаффмана: в его концевых вершинах располагаются все возможные символы, а ветви, ведущие к этим вершинам, определяют соответствующий код. Для построения дерева нужно иметь таблицу, содержащую для каждого символа количество его появлений в исходном файле. Процесс построения дерева формально описывается следующей схемой.

1. Создаем концевые вершины, содержащие все символы входного алфавита, задаем вес каждой вершины в соответствии с таблицей количеств появлений, объявляем все эти вершины свободными.
2. В множестве свободных вершин находим две вершины (обозначим их A и B), имеющие наименьшие веса, создаем новую родительскую вершину C, присоединяем A и B к C слева и справа, устанавливаем вес C равным сумме весов A и B, исключаем A и B из множества свободных вершин, добавляем C к множеству свободных вершин.
3. Если в множестве свободных вершин более одного элемента, переходим к пункту 2, иначе единственный оставшийся элемент является корнем дерева Хаффмана.

Далее левым ребрам построенного дерева ставится в соответствие число 0, а правым ребрам — 1. Теперь, спускаясь от корня к конкретной концевой вершине и последовательно выписывая нули и единицы, поставленные в соответствие проходимым ребрам, мы получаем код символа, хранящегося в этой вершине. Так строится кодовая таблица соответствий символ—код. Если количество различных символов в файле равно n, то длина кода символа может варьироваться от 1 до $n-1$ в зависимости от частоты появления символов.

Кодирование и декодирование осуществляется заменой каждого символа его кодом и наоборот. При практической реализации биты кодов символов выписываются подряд, а затем делятся на группы по 8 бит (байты), которые и выдаются в выходной поток. При декодировании входной поток рассматривается как последовательность битов, определяющих направления спуска по ветвям дерева Хаффмана. При достижении концевой вершины соответствующий символ выдается в выходной поток, и спуск опять начинается от корня. Так как последний байт при кодировании может оказаться только частично заполненным, то

разумно либо ввести в кодовую таблицу символ конца файла **EOF** (присвоив ему самый длинный код), либо посчитать и сохранить в закодированном файле общее количество битов, требующих декодирования. Отметим, что при кодировании удобно работать с таблицей кодов, а при декодировании — с деревом Хаффмана.

Задача 4-3-34. Реализуйте «демонстрационный» алгоритм кодирования Хаффмана при условии, что кодовая таблица хранится в отдельном файле в текстовом формате **символ—пробел—код** (отдельная строка для каждого символа):

```
a 00
b 01
c 10
d 111
```

Кодированный файл должен представлять собой последовательность символов 0 либо 1. Например, aabdc закодируется последовательностью байтов 00000111110. Отдельно реализуйте соответствующую функцию декодирования.

Задача 4-3-35. Реализуйте алгоритм битового кодирования Хаффмана при условии, что кодовая таблица хранится в отдельном файле в текстовом формате **символ—пробел—код** (см. задачу 4-3-34) и содержит терминальный символ **EOF**, завершающий закодированную последовательность. Отдельно реализуйте соответствующую функцию декодирования.

Указание. Алгоритм кодирования сводится к работе с функциями из задач 1-7-5, 1-7-7. Отметим, что если в начале закодированного файла в формате `uint64_t` хранится общее количество битов закодированной последовательности, то можно не вводить терминальный символ **EOF**.

Задача 4-3-36. Реализуйте функцию построения кодовой таблицы для алгоритма Хаффмана и ее сохранение в отдельном файле в текстовом формате **символ—пробел—код** (см. задачу 4-3-34).

Указание. По заданному входному файлу определяется частота появления каждого символа, затем строится дерево Хаффмана, а затем — искомая кодовая таблица.

Задача 4-3-37. Реализуйте двухпроходный кодировщик файлов по алгоритму Хаффмана: за первый просмотр файла строится кодовая таблица, а за второй — кодируется файл. При этом

кодовая таблица (см. задачу 4-3-35) должна сохраняться в заголовочной части сжатого файла в компактной (битовой) форме.

Адаптивное перестроение дерева

Существует адаптивный вариант алгоритма Хаффмана, который не требует передачи дерева (или частотной таблицы) вместе с закодированными данными. Идея этого алгоритма заключается в адаптивном перестроении дерева с учетом поступивших символов при кодировании и соответствующем преобразовании дерева при декодировании.

Пусть мы имеем некоторое дерево Хаффмана. Пронумеруем вершины этого дерева слева направо и снизу вверх, т. е. нумерация начинается с элементов самого нижнего уровня дерева, потом переходит на предыдущий уровень и т. д. Корень дерева получает максимальный номер.

Будем называть дерево *упорядоченным*, если веса вершин не убывают в порядке построенной нумерации.

Построим начальное дерево Хаффмана, считая, что все символы имеют вес 1. Теперь при появлении нового символа мы должны перестроить дерево так, чтобы сохранить свойство упорядоченности. Во-первых, надо увеличить вес соответствующей концевой вершины и всех родителей по ветви, ведущей к корню. Если после этого дерево остается упорядоченным, то считывается следующий символ. Если же упорядоченность дерева нарушается, т. е. вес некоторой вершины A становится больше, чем вес некоторого правого (по нумерации) соседа, то дерево необходимо перестроить. Для этого из множества правых соседей, вес которых меньше веса A, выбирается сосед с наибольшим номером, и эта вершина и вершина A обмениваются содержимым (вместе с информацией о потомках), т. е. соответствующие ветви дерева «отрезаются» и «переподклеиваются». Затем по ветвям, ведущим к корню от переставленных вершин, корректируются веса. Процесс продолжается, пока дерево не станет упорядоченным. Далее считывается очередной символ.

Задача 4-3-38. Реализуйте кодировщик и декодировщик файлов по адаптивному алгоритму Хаффмана.

Идеи реализации. Обе программы имеют одну и ту же структуру, которая описывается следующей схемой.

1. Инициализировать дерево с равномерным распределением символов.

2. Получить очередной символ (или код) и закодировать (раскодировать) его в соответствии с текущим состоянием дерева Хаффмана.
3. Перестроить дерево по полученному символу.
4. Если есть еще символы (коды) во входном потоке, то перейти к пункту 2, иначе закончить работу.

Задача 4-3-39. Другой способ построения адаптивного алгоритма Хаффмана состоит в инициализации дерева специальным символом **ESC** (начальный код 0, число повторений 1) и символом конца файла **EOF** (начальный код 1, число повторений 1).

Если символ во входном потоке появляется первый раз, то перестройка дерева заключается в добавлении новой концевой вершины с этим символом и с количеством появлений 1. Для этого первая (по нумерации) вершина-лист заменяется на вершину-узел с двумя потомками: новая вершина добавляется слева и становится первой по нумерации, а бывшая первая добавляется справа и становится по нумерации второй. Далее изменяется вес всех родителей по ветви к корню и проводится стандартная перестройка дерева. Если символ поступает повторно, то изменяется вес соответствующей вершины, далее веса всех потомков, а затем, если требуется, проводится перестройка дерева. Схема кодирования имеет следующий вид.

1. Инициализировать дерево символами **ESC** и **EOF**.
2. Получить очередной символ.
3. Если символ уже есть в дереве, то выдать его код, иначе выдать код символа **ESC** и выдать сам символ.
4. Перестроить дерево по полученному символу.
5. Если есть еще символы во входном потоке, то перейти к пункту 2, иначе закончить работу.

Схема декодирования имеет следующий вид.

1. Инициализировать дерево символами **ESC** и **EOF**.
2. Получить очередной код.
3. Если код оказывается кодом символа **ESC**, то следующие 8 бит представляют собой сам символ — выдать этот символ, иначе выдать символ, соответствующий коду.
4. Перестроить дерево по полученному символу.
5. Если встретился символ **EOF**, то закончить работу, иначе перейти к пункту 2.

Алгоритм LZW

Рассмотрим базовый вариант алгоритма Лемпеля—Зива—Велча (A. Lempel, J. Ziv, T. Welch, LZW), ключевые идеи которого используются во многих программах-архиваторах (например, в архиваторе 7Zip). Суть алгоритма заключается в обнаружении во входном потоке повторяющихся цепочек байтов, составлении таблицы обнаруженных цепочек и выдаче в выходной поток кодов (номеров строк таблицы), соответствующих обнаруженным цепочкам. Если исходный файл имеет много повторяющихся последовательностей байтов, то каждая такая последовательность будет заменена на ее порядковый номер в таблице. Важной особенностью алгоритма является его полная обратимость: распаковщик, анализируя в процессе работы сжатые данные, может построить копию исходной таблицы, следовательно, ее не надо передавать в сжатых данных.

Кодирование

Создается таблица, способная вместить достаточно большое количество строк. Первые 256 строк этой таблицы инициализируются всеми возможными односимвольными цепочками (т. е. символами с кодами от 0 до 255). Дальнейший алгоритм выглядит так:

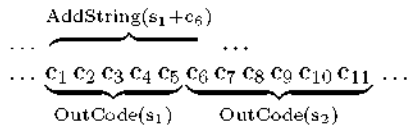
```
s = <пустая цепочка>;
while (есть символы во входном потоке) {
  c = NextSym(); /* взять очередной символ */
  if (InTable(s + c)) { /* цепочка s + c уже есть в
 таблице, удлиняем цепочку прочитанным символом: */
    s = s + c;
  } else { /* цепочки s + c нет в таблице,
        выводим код, соответствующий цепочке s: */
    OutCode(s);
    /* добавляем новую цепочку s + c в таблицу: */
    AddString(s + c);
    /* готовимся к следующей цепочке: */
    s = c;
  }
}
/* выводим код для оставшейся цепочки s: */
OutCode(s);
```

Здесь знак «+» означает конкатенацию цепочек. Алгоритм кодирования каждый раз пытается найти в таблице цепочку, соответствующую читаемой последовательности символов. Если это в какой-то момент не удается, то накопленная к этому моменту цепочка заносится в таблицу под следующим номером.

Замечание. В какой-то момент может наступить переполнение таблицы. В этом случае кодировщик выводит в выходной поток специальный код очистки и таблица цепочек инициализируется заново. Обычно в реальных алгоритмах применяется таблица с 4096 кодами для цепочек. При этом число 256 является кодом очистки (обозначим его **ESC**), а число 257 — кодом конца информации (**EOF**). Эти строки таблицы для цепочек не используются. В данном случае для каждого выходного кода достаточно 12 бит. Поэтому при выводе целесообразно упаковывать каждые два кода в три байта. Для упрощения логики декодера обычно первым кодом сжатых данных является **ESC**, что сразу вызывает инициализацию таблицы цепочек.

Декодирование

Функции `OutCode(s)` и `AddString(s + c)` вызываются в процессе кодирования, если строка `s + c` отсутствует в таблице. Поэтому входной поток, состоящий из символов c_i, структурируется следующим образом:



В результате в данном примере код только что записанной в таблицу строки $s_1 + c_6$ может на следующем шаге появиться в выходном потоке, если только s_2 совпадет с $s_1 + c_6$, и тогда символ c_6 будет равен первому символу строки s_1. Эти замечания позволяют построить формальный алгоритм декодирования, последовательно восстанавливающий таблицу цепочек, идентичную исходной таблице кодирования.

Будем считать, что функция `InitTable()` инициализирует таблицу всевозможными односимвольными цепочками и символами **ESC** и **EOF**; `String(code)` возвращает строку из таблицы, соответствующую коду `code`; `Char(code)` возвращает первый символ такой строки; `NextCode()` извлекает 12-битные

коды из входного потока байтов. Тогда процедура декодирования имеет следующий вид:

```
while ((code = NextCode()) != EOF) {
  /* пока файл не пуст: */
  if (code == ESC) { /* code есть код очистки;
       заново инициализируем таблицу: */
    InitTable();
    code = NextCode();
    if (code == EOF) break;
    /* выводим цепочку, соответствующую
       коду code в таблице цепочек: */
    OutString(code);
    /* запоминаем текущий код для последующей
       модификации таблицы цепочек: */
    old = code;
  } else {
    if (InTable(code)) {/* в таблице есть строка
              для кода code; выводим цепочку: */
      OutString(code);
      /* формируем и добавляем новую цепочку: */
      AddString(String(old) + Char(code));
      old = code;
    } else { /* в таблице нет строки
            для кода code; формируем цепочку: */
      s = String(old) + Char(old);
      /* выводим цепочку:  */
      OutString(s);
      /* и добавляем ее в таблицу: */
      AddString(s);
      old = code;
    }
  }
}
```

Задача 4-3-40. Реализуйте описанные алгоритмы в виде программ упаковки и распаковки файлов. Для представления таблицы цепочек можно использовать следующие структуры:

а) массив указателей на текстовые строки;

б) дерево поиска (сбалансированное) для элементов типа **char** *;

в) дерево символов (см. задачу 3-3-19).

Задача 4-3-41. Улучшите степень сжатия на основе следующих соображений. Таблица цепочек разрастается по мере просмотра входного файла. Следовательно, пока в таблице цепочек заполнено менее 512 строк, кодировщик может выдавать 9-битные коды. При появлении 512-й строки кодировщик переходит на 10-битные коды, после 1024-й строки — на 11-битные и т. д. Декодер также должен анализировать длину своей таблицы в процессе декодирования и соответственно переходить на извлечение более длинных кодов из входного потока.

4.3.5. Работа с bmp-файлами

Наиболее простым форматом хранения цветного изображения является формат bmp (bitmap picture) с 24-битным представлением цвета. В этом случае bmp-файл состоит из двух частей. В первых 54 байтах хранится служебная информация о размерах картинки и структуре самого файла, а далее по строкам записано изображение. При этом первой идет нижняя строка (начало координат находится в левом нижнем углу изображения), а все строки выровнены по 32-битной границе; если ширина картинки не кратна четырем, то дописывается необходимое число фиктивных байт. Цвет каждой точки изображения представлен в формате RGB и кодируется тремя байтами: по одному байту для интенсивности красного, зеленого и синего для каждой точки. Количество различных цветов в этом случае равно $\mathrm{maxColor} = 2^{24}$. Для формирования служебной 54-байтовой таблицы удобно завести следующие переменные с подходящими типоразмерами (считаем, что размер типа **char** — 1 байт, **short** — 2 байта, **int** — 4 байта, а переменные `w`, `h` заданы и содержат ширину и высоту картинки соответственно).

```
char type[2] = {'B', 'M'};
unsigned int fSize = 54 + w * h * 3;
unsigned short reserved1 = 0;
unsigned short reserved2 = 0;
unsigned int offBits = 54;
unsigned int size = 40;
unsigned int width = w;
unsigned int height = h;
unsigned short planes = 1;
unsigned short bitCount = 24;
```

```
unsigned int compression = 0;
unsigned int sizeImage = w * h * 3;
unsigned int xPixelsPreMeter = 0;
unsigned int yPixelsPreMeter = 0;
unsigned int clrUsed = 0;
unsigned int clrImportant = 0;
```

Затем выделить блок памяти `unsigned char header[54]` и последовательно в указанном порядке записать в него содержимое данных переменных:

```
memcpy((void *)header, (void *) type, 2);
memcpy((void *)(header+2), (void *) &fSize,
                                  sizeof(fSize));
.....
```

Для формирования изображения удобно подготовить отдельный массив и заполнить его значениями соответствующих цветов:

```
unsigned char *pict = (unsigned char *)malloc(
                              width * height * 3);
for (i = 0; i < height; i++) {
  for (j = 0; j < width; j++) {
    Color(&r, &g, &b, i, j);
    pict[i * width * 3 + 3 * j + 0] = b;
    pict[i * width * 3 + 3 * j + 1] = g;
    pict[i * width * 3 + 3 * j + 2] = r;
  }
}
```

Здесь мы считаем, что функция `Color()` возвращает тройку `r`, `g`, `b` для (i,j)-й точки и ширина картинки кратна четырем. Если массивы `header` и `pict` сформированы, то далее их содержимое последовательно записывается в файл:

```
out = fopen("Pict.bmp", "wb");
fwrite((void*)header, 1, 54, out);
fwrite((void*)pict, 1, height* width * 3, out);
fclose(out);
```

Замечание. Переменные, используемые для формирования заголовка bmp-файла, можно объединить в отдельную структуру `struct strHeader`, проинициализировать нужными значениями, а затем записать структуру в файл. Однако такой подход будет

работать, только если поля структуры разместятся в памяти непрерывно и размер типа `strHeader` будет равен 54 байта. Отметим, что в общем случае при размещении в памяти ЭВМ в структуру могут добавляться фиктивные байты для оптимизации доступа к полям. За правила выравнивания элементов структуры (а также объединения, класса) отвечает параметр компилятора, значение которого можно локально изменить специальной директивой препроцессора `#pragma pack`:

```
#pragma pack(push, 1)
struct strHeader {
  /* описание полей структуры */
};
#pragma pack(pop)
```

Параметры `push, 1` первой директивы означают, что текущее значение выравнивания упаковки элементов структуры записывается во внутренний стек компилятора, и с данного момента устанавливается однобайтовое выравнивание. Параметр `pop` заключительной директивы обеспечивает выталкивание вершины внутреннего стека компилятора, и считанная величина становится новым значением выравнивания упаковки.

Задача 4-3-42. Пусть имеется текстовый файл `Pict.rgb`, в первой строке которого записаны положительные целые числа height и width (width кратно четырем). Далее содержатся height однотипных строк: в i-й строке через пробел записаны тройки целых чисел со значениями от 0 до 255 $(r_{i,j}, g_{i,j}, b_{i,j})$, $j = 1, \ldots,$ width. Считая, что таким образом закодировано цветное изображение размера height $\times$ width, сохраните его в стандартном bmp-формате. Отдельно напишите функцию декодирования файла `Pict.bmp` в описанный формат.

Задача 4-3-43. Напишите функцию создания картинки размера height $\times$ width, содержащей окружность указанного радиуса, требуемой толщины и заданных цветов для линии и фона.

Задача 4-3-44. Напишите функцию создания картинки размера height $\times$ width, содержащей буквицу «Аз» заданного цвета на заданном фоне.

Указание. В прямоугольном заполненном нулями массиве размера, например, 50×100 задайте единицами контур буквы и примените к нему функцию заливки из задачи 1-9-23. Полученный таким образом массив после преобразования подобия с нуж-

ным коэффициентом будет основой для создаваемой картинки. Отметим, что при значительном масштабировании качество будет ухудшаться, поэтому во избежание подобных проблем обычно используют векторный формат хранения исходного изображения.

— Представляешь, я придумал классный архиватор, позволяющий любой файл сжать до одного байта!
— Так патентовать надо!
— Не принимают. Говорят, что нужен еще и разархиватор. А с этим пока проблема...

4.4. Грамматический разбор и компиляция

Мы будем использовать упрощенное определение формальной грамматики как тройки G = (V,W,F), где V — множество терминальных символов, W — множество нетерминальных (выводимых) символов, F — множество правил вывода. Языком, порожденным грамматикой G, будем называть множество цепочек символов из V и W, образованных в соответствии с правилами вывода F. Для задания формальных грамматик мы будем использовать нормальную форму Бэкуса—Наура (возможно, с незначительными модификациями).

Задача 4-4-1. Постройте формальную грамматику для описания всех возможных идентификаторов языка C.

Решение. Зададим множества символов (терминальных и нетерминальных):

```
V = {"0", "1", ..., "9", "_",
       "A", "B", ..., "Z", "a", "b", ..., "z"},
W = {<идентификатор>, <буква>, <цифра>}.
```

Опишем правила вывода в форме Бэкуса—Наура.

```
<идентификатор> ::= <буква> |
                    <идентификатор> <буква> |
                    <идентификатор> <цифра>
<цифра> ::= "0" | "1" | "2" | "3" | "4" |
            "5" | "6" | "7" | "8" | "9"
<буква> ::= "_" | "A" | ... | "Z" |
            "a" | ... | "z"
```

Задача 4-4-2. Постройте формальную грамматику для описания всех возможных десятичных числовых констант, записанных по правилам языка С.

Решение. Выпишем здесь только правила вывода.

```
<константа> ::= <константа без знака> |
                <знак> <константа без знака>
<знак> ::= "+" | "-"
<константа без знака> ::= <целое> | <вещественное>
<вещественное> ::= <число> | <число> <порядок>
<число> ::= <целое> "." | "." <целое0> |
            <целое> "." <целое0>
<порядок> ::= "E" <целое0> | "E" <знак> <целое0>
<целое0> ::= <цифра> | <цифра> <целое0>
<цифра> ::= "0" | <цифра1>
<цифра1> ::= "1" | "2" | "3" | "4" |
             "5" | "6" | "7" | "8" | "9"
<целое> ::= "0" | <целое1>
<целое1> ::= <цифра1> | <целое1> <цифра>
```

Замечание. Для сокращения записи можно в квадратных скобках указывать необязательный фрагмент конструкции. Например, правило вывода для символа <порядок> запишется так:

```
<порядок> ::= "E" [ <знак> ] <целое0>
```

Задача 4-4-3. Постройте формальную грамматику для описания всех возможных (не только десятичных и числовых) констант, допустимых в языке С.

Задача 4-4-4. Постройте формальную грамматику для описания объявлений переменных, массивов, указателей и массивов указателей в языке С.

Идеи реализации. Будем использовать метасимволы <описание>, <тип>, <имя>, <объект>, <константа> и др. Тогда, например,

```
<описание> ::= <тип> <список объектов>;
<список объектов> ::= <объект> |
                      <объект> "," <список объектов>
<объект> ::= <атом или массив> | "*" <объект>
<атом или массив> ::= <имя> | <имя> "[" "]" |
                            "(" <имя> ")" "[" "]" |
                      "(" "*" <объект> ")" "[" "]" |
```

```
        <атом или массив> "[" <константа> "]" |
    "(" <атом или массив> ")" "[" <константа> "]"
```

Задача 4-4-5. Постройте формальную грамматику для описания прототипов функций языка С.

Задача 4-4-6. Постройте формальную грамматику для описания арифметических выражений, включающих операции сложения, вычитания, умножения, деления, а также унарные + и — и использующих в качестве операндов простые переменные и числовые десятичные константы.

Задача 4-4-7. Добавьте к предыдущей грамматике возможность включить в выражение одномерные массивы. Учтите возможность появления формулы в индексном выражении.

Задача 4-4-8. Добавьте к предыдущей грамматике возможность включить в выражение вызовы функций.

4.4.1. Лексический анализатор, конечные автоматы

Основной задачей лексического анализа является выделение из входного текста определенных групп символов — лексем, которые могут являться терминальными или нетерминальными метасимволами в некоторой формальной грамматике. Типичная частная задача подобного рода — обнаружение в тексте вхождений ключевых слов из заданного набора. Наиболее эффективно подобная задача обнаружения решается с помощью конечного автомата. Не вдаваясь в строгую математическую теорию, отметим, что конечный автомат для обнаружения ключевых слов может быть легко реализован в виде ориентированного графа, в котором вершины соответствуют состояниям автомата, а ребра — переходам между состояниями, выполняемым при получении очередного символа из входного текста. В идейном плане подобная реализация близка к представлению словаря в виде дерева символов (см. задачу 3-3-19), которое мы и возьмем в качестве модели для построения конечного автомата.

Итак, пусть мы имеем некоторое дерево символов, ветви которого соответствуют ключевым словам. Пусть одна из вершин этого дерева является текущей (в начальный момент текущей является корневая вершина). При поступлении на вход очередного символа этот символ ищется среди вершин, являющихся непосредственными потомками текущей. Если поиск оказывается успешным, то текущей вершиной становится та, которая содержит этот символ. В противном случае текущей

становится корневая вершина (вообще говоря, это может быть и другая вершина, если ключевое слово имеет несколько совпадающих друг с другом подстрок, см. задачу 3-5-12 о поиске подстроки). Если текущая вершина соответствует последнему символу в ключевом слове, то фиксируется обнаружение данного ключевого слова, и текущей вершиной становится корень дерева.

Задача 4-4-9. Реализуйте конечный автомат для обнаружения набора слов, заданного в виде массива элементов типа **char** *. На основе этой реализации напишите программу, которая читает текстовый файл и выдает на экран все строки, в которых встречается хотя бы одно слово из заданного набора ключевых слов. Следует предусмотреть возможность задания произвольного набора ключевых слов. Например, эти слова могут считываться из другого текстового файла в начале работы программы.

Отдельные задачи данного раздела о лексическом разборе и анализе программ на языке C, достаточно сложны и в идеальном варианте требуют учета многих тонкостей синтаксиса языка. Поэтому на начальном этапе реализации целесообразно рассмотреть некоторое подмножество синтаксических правил построения языка, чтобы получить работоспособный вариант программы, который затем можно совершенствовать.

Задача 4-4-10. Реализуйте отдельные функции препроцессора текста программы на языке C.

1. Вставьте в текст программы содержимое файла, определяемого директивой **#include**.
2. Обработайте директивы **#ifdef** и **#ifndef** в соответствии с определениями директив **#define**.

Задача 4-4-11. Назовем парой скобок два заданных набора символов. Один их этих наборов будет открывающей скобкой, другой — закрывающей. Примерами подобных скобок могут служить `(` и `)`, `begin` и `end`, `/*` и `*/` и т. п. Целью обработки может быть анализ, сохранение или уничтожение символов, находящихся между двумя парными скобками.

1. Удалите из файла все комментарии в стиле языков C и C++ (т. е. между /* и */, а также между // и концом строки).
2. Проверьте сбалансированность системы вложенных скобок в программе на языке C или C++. Рассмотрите скобки

{, }, скобки [,] в индексных выражениях, учтите все возможные варианты использования скобок (,).

3. Проверьте сбалансированность системы вложенных скобок в файле при условии, что строки, определяющие скобки и правила их взаимного расположения, определены в некотором файле. В качестве примеров можно рассмотреть «скобки» в TEX- или HTML-файле.

Задача 4-4-12. Требуется выделить из текста некоторые фрагменты (лексемы), синтаксис появления которых в тексте определяется известной системой правил. Результатом работы программы должен быть некоторый список таких лексем.

1. Выделите из (арифметического) выражения, записанного по правилам языка C, имена переменных (функций) и символьные представления констант.
2. Выделите из файла с программой на языке C объявления функций и сформируйте строки с прототипами этих функций.
3. Выделите из текста программы на языке C объявления и вызовы функций и сформируйте список перекрестных вызовов (т. е. список функций, вызываемых каждой функцией).
4. Имея текст отдельной функции на языке C, составьте список локальных и глобальных имен, на которые есть ссылки внутри данной функции.

Задача 4-4-13. Реализуйте программу форматирования текста на заданном алгоритмическом языке (например, C). Выходной текст должен удовлетворять правилам хорошего стиля: иметь систему отступов во вложенных конструкциях циклов и разветвлений, промежутки между определениями отдельных функций и т. п.

Задача 4-4-14. Реализуйте программу выделения из HTML-файла всех ссылок на URL (универсальный указатель ресурсов). Форматы HTML-файлов и записей URL описаны во многих руководствах, и здесь мы их приводить не будем.

4.4.2. Построение дерева грамматического разбора

Анализ формальной грамматики, описывающей некоторый язык, позволяет построить дерево разбора, в котором концевые вершины будут соответствовать терминальным метасимволам, а все остальные вершины — нетерминальным метасимволам. В случае арифметического выражения с обычными арифме-

тическими операциями это дерево будет бинарным, причем концевым вершинам можно поставить в соответствие значения переменных или констант, участвующих в выражении, а остальным вершинам — операции и значения результатов этих операций. Обходя дерево снизу вверх и выполняя операции, поставленные в соответствие вершинам, в корне дерева мы получим результат вычисления всего выражения.

Задача 4-4-15. Реализуйте алгоритм построения дерева разбора для арифметического выражения, заданного текстовой строкой. На основе построенного дерева реализуйте функцию вычисления значения выражения при заданных значениях переменных, участвующих в выражении.

Идеи реализации. Определим тип вершины дерева как

```
typedef struct _T {
  char *text;
  double value;
  struct _T *left, *right;
} TreeNode;
```

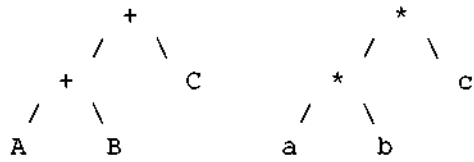
где `text` содержит имя переменной или знак операции, `value` — значение переменной или результат операции.

Будем использоваться фигурные скобки { } для обозначения фрагмента, который может повторяться нуль или более раз. Определим формальную грамматику для выражения следующим образом:

```
<выражение> ::=
  [ <плюс/минус> ] <терм> { <плюс/минус> <терм> }
<терм> ::=
 <множитель> { <умножить/разделить> <множитель> }
<множитель> ::= <переменная> | <константа> |
                "(" <выражение> ")"
<плюс/минус> ::= "+" | "-"
<умножить/разделить> ::= "*" | "/"
<переменная> ::= <идентификатор>
```

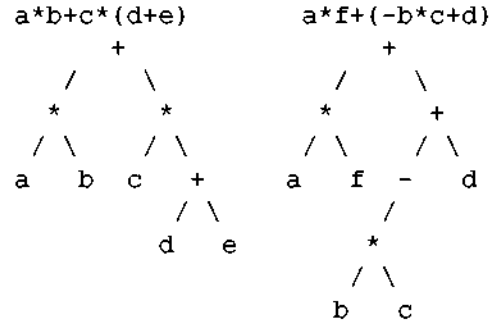
Символы <идентификатор> и <константа> мы определили ранее (см. задачи 4-4-1, 4-4-2). Итак, выражение без знака есть последовательность термов с аддитивными операциями, а терм — последовательность множителей с мультипликативными операциями. Построение дерева реализуется набором рекурсив-

ных процедур, которые строят деревья для выражения без знака, терма и множителя. Например, для выражений `A+B+C` (`A`, `B`, `C` — термы) и `a*b*c` (`a`, `b`, `c` — множители) должны получиться деревья



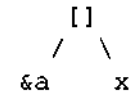
В случае обработки множителя типа «выражение в скобках» строится дерево для этого выражения и его корень присоединяется к вершине, соответствующей данному множителю. Унарные операции плюс и минус реализуются как вершины дерева с одним потомком, расположенные над соответствующим термом.

В качестве примеров приведем деревья грамматического разбора для следующих арифметических выражений:



Задача 4-4-16. Добавьте к предыдущей реализации возможность использования в выражении одномерных массивов.

Идеи реализации. Имя массива определяет базовый адрес, а индекс — смещение относительно этого адреса. Таким образом, операция индексирования `[]` по значению базового адреса и смещения определяет значение элемента. Если обозначить `&a` базовый адрес массива `a`, то выражению `a[x]` можно поставить в соответствие дерево



4.4.3. Применение деревьев грамматического разбора

Деревья грамматического разбора активно применяются в компиляторах алгоритмических языков и других программах, где требуется обработка текстов, записанных по правилам некоторой формальной грамматики (например, браузерах, системах подготовки печатных изданий). В этом разделе мы рассмотрим примеры применения деревьев грамматического разбора для решения нескольких интересных задач.

Задача 4-4-17. Реализуйте интерпретатор модельного алгоритмического языка со следующим синтаксисом:

`var x,y[10]`	объявление переменной `x` и массива `y` на 10 элементов;
`x = <выражение>` `y[x] = <выражение>`	операторы присваивания;
`print x,y[2]`	вывод значений переменных;
`end`	конец программы.

Идеи реализации. Результат обработки инструкций описания переменных заключается в резервировании необходимого количества памяти и распределении этой памяти между заданными переменными. В результате получается таблица распределения памяти, устанавливающая соответствие между именем переменной (или массива) и базовым адресом этой переменной. Обработка оператора присваивания выполняется аналогично задаче 4-4-15, при этом значения переменных пишутся и читаются в соответствии с таблицей распределения памяти.

Если выполнить обход построенного дерева разбора арифметического выражения в соответствии с правилом

«обработка вершины — левое поддерево — правое поддерево»,

то мы получим так называемую префиксную (польскую) запись арифметического выражения. Для выражений

$$a*b+c*(d+e), \quad a*f+(-b*c+d)$$

из задачи 4-4-15 и соответствующих им деревьев будем иметь

$$+*ab*c+de, \quad +*af+\pm*bcd.$$

Во второй формуле символ $\pm$ используется для обозначения унарного минуса, т. е. операции, отвечающей за смену знака аргумента. Префиксная форма обрабатывается справа налево:

как только встречается код операции, соответствующее действие применяется к аргументам (аргументу) справа.

Если же выполнить обход построенных деревьев по правилу «левое поддерево — правое поддерево — обработка вершины», то мы получим так называемую постфиксную форму записи:

$$ab*cde+*+, \quad af*bc+\pm d++.$$

Подобная форма, называемая обратной польской нотацией, активно применяется в реальных системах, так как оказывается очень удобной для вычисления результата работы стекового калькулятора. Так, например, выражение $a*b+c*(d+e)$ дает постфиксный код $ab*cde+*+$. Стековый калькулятор имеет стек, содержащий числа, и может выполнять действия с двумя верхними элементами (одним в случае унарного оператора). Результат операции замещает эти два элемента в стеке. Выпишем состояния стекового калькулятора при просмотре указанного постфиксного кода (все выражения представляют собой вычисленные значения).

```
Номер шага:
  1  2   3   4   5   6   7     8         9
Строка кода
  a  b   *   c   d   e   +     *         +
Состояние стека:
                       e
                   d   d  d+e
     b         c   c   c   c  c*(d+e)
  a  a  a*b a*b a*b a*b a*b   a*b   a*b+c*(d+e)
  -  -  --- --- --- --- --- ------- -----------
```

Имея реализацию стекового калькулятора, мы может разделить процесс интерпретации на два этапа: построение постфиксного кода и интерпретацию кода стековым калькулятором. Этот вывод приводит к следующей задаче.

Задача 4-4-18. Преобразуйте интерпретатор из задачи 4-4-17 так, чтобы он сначала строил некоторый код, аналогичный постфиксному, а затем с помощью стекового калькулятора его интерпретировал. В результате должны получиться «компилятор» и «исполнитель» программы на исходном модельном языке.

Идеи реализации. В рассматриваемом модельном языке присутствуют операции индексирования, чтения из памяти и

записи в память. Поскольку адреса элементов массива тоже приходится вычислять, следует дополнить стековый калькулятор операциями чтения из памяти и записи в память. Для простоты можно считать, что адрес в нашем понимании — это индекс элемента в объединенном массиве всех данных компилируемой программы. Вот одно из возможных решений.

Чтение из памяти. Вершина стека — адрес, в результате операции вершина заменяется на значение по этому адресу.

Запись в память. Вершина стека — значение, под ней — адрес, в результате операции происходит запись значения по указанному адресу и оба элемента удаляются из стека.

Заметим также, что при формировании кода нет необходимости строить дерево разбора целиком. Действительно, если сформирован родитель с двух концевых вершин a, b, то мы можем сразу приписать вершине с кодовую строку abc, а вершины a, b удалить. С идейной точки зрения это означает, что, просматривая арифметическое выражение слева направо, мы сразу выполняем те вычисления, которые можно выполнить, с учетом скобок и приоритетов операций (а также ранее произведенных вычислений).

Задача 4-4-19. Добавьте в модельный алгоритмический язык инструкции условного и безусловного переходов и реализуйте их в компиляторе и исполнителе.

Идеи реализации. Требуемые инструкции можно задать в виде

```
<метка>: <оператор программы>
goto <метка>
if (<выражение>) goto <метка>
```

В систему команд стекового калькулятора можно добавить команду безусловного перехода на команду с заданным порядковым номером и команду перехода на команду с заданным порядковым номером при условии, что текущая вершина стека калькулятора отлична от нуля. Обнаружив в тексте программы метку, следует запомнить в отдельной таблице номер первой команды, соответствующей помеченному оператору. Переход на ранее встречавшуюся метку не вызывает проблем (номера команд уже присутствуют в таблице). Для перехода на метки, которые пока еще не встретились, можно формировать код перехода с неопределенным номером и запоминать в таблице позиции кода этих неопреде-

ленных ссылок. После обработки всего текста все объявленные метки будут зафиксированы в таблице, и мы сможем записать истинные номера команд в неопределенные операторы переходов.

Задача 4-4-20. Добавьте в модельный алгоритмический язык инструкции циклов и реализуйте их в компиляторе и исполнителе.

Идеи реализации. Циклы легко реализуются с помощью команд переходов. Вот один из возможных вариантов введения и интерпретации цикла:

```
while (<выражение>) m1:  if (!<выражение>) goto m2
  <тело цикла>             <тело цикла>
endwhile                 goto m1
                       m2:
```

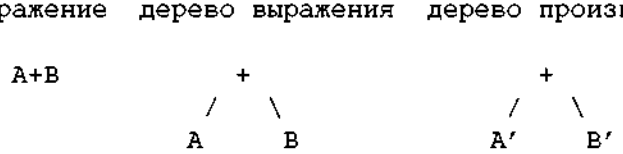
Задача 4-4-21. Добавьте в модельный алгоритмический язык возможность использовать вызовы стандартных функций из некоторого заданного набора (например, элементарных математических функций).

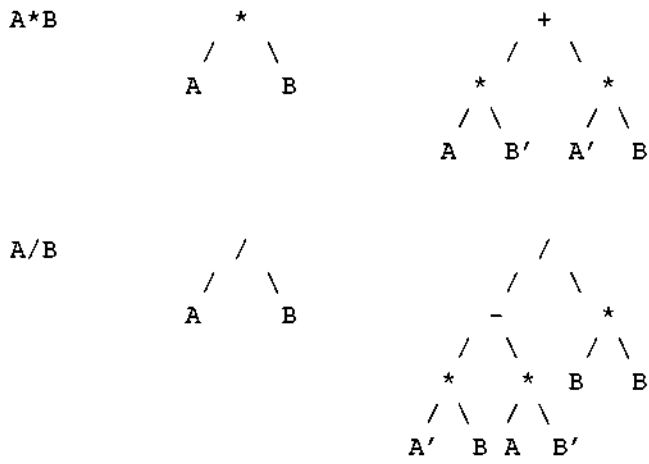
Задача 4-4-22. Реализуйте программу рисования графика функции $y = f(x)$ по заданной формуле для функции $f(x)$.

Идеи реализации. Выражение, задающее функцию $f(x)$, переводится в код стекового калькулятора. Затем этот код многократно используется для вычисления координат точек, принадлежащих графику.

Задача 4-4-23. Реализуйте программу, которая по заданной формуле для функции $f(x)$ формирует выражения для производной $f'(x)$.

Идеи реализации. Построим дерево разбора для выражения функции $f(x)$. Далее строится дерево разбора для производной по правилам дифференцирования. Например, если A и B — деревья для двух выражений, A′ и B′ — деревья для их производных, то имеем следующие правила формирования деревьев:





Обходя дерево производной слева направо и расставляя скобки, где это необходимо, можно получить формулу для производной. Отметим, что этот вариант формулы может оказаться весьма громоздким.

Задача 4-4-24. Пусть заданы формула и ее дерево разбора. Предложите и реализуйте алгоритм для упрощения этой формулы (приведение подобных членов, вычисление константных выражений, вынос общих множителей).

Идеи реализации. Эта задача достаточно сложна. Скажем только, что здесь нужно анализировать структуру дерева разбора, отыскивать в нем подобные поддеревья, находить преобразования дерева, сохраняющие результирующее значение в корне и уменьшающие сложность.

Задача 4-4-25. Реализуйте оптимизирующий компилятор модельного языка.

Идеи реализации. Ограничимся линейными последовательностями операторов присваивания. Результатом грамматического разбора исходного текста является набор деревьев для каждого оператора присваивания. Можно провести анализ этих деревьев с целью

а) константные поддеревья заменить на соответствующие константы;

б) обнаружить одинаковые поддеревья и выделить их в отдельное дерево с сохранением промежуточных результатов в памяти;

в) вынести вычисление выражений, не меняющихся в цикле, из тела цикла.

Если математика для вас — это просто, то информатика будет проще простого (*экспериментально установленный факт*).

Глава 5
ПРИЛОЖЕНИЕ

5.1. Прожиточный минимум

Процесс разработки программы на ЭВМ состоит из следующих стандартных шагов (далее будет дано их детальное описание).

1. С помощью текстового редактора, позволяющего сохранить результат в формате plain text, набираем программу, например в файле `tsk-1-1-1.c`.
2. Компилируем `tsk-1-1-1.c` и получаем файл `tsk-1-1-1.o`. Если компилятор выдает сообщения о синтаксических ошибках или предупреждения, то исправляем код, обычно начиная с первого замечания, и компилируем заново.
3. Собираем (линкуем) `tsk-1-1-1.o` с библиотечными функциями и получаем `tsk-1-1-1.exe`.
4. Запускаем итоговый файл `tsk-1-1-1.exe` и приступаем к тестированию и поиску алгоритмических и программных ошибок.

5.1.1. Выбор рабочего окружения

Если на доступной вам ЭВМ установлена ОС типа Linux со стандартным набором пакетов, то в вашем распоряжении имеются консольный редактор типа vi, редактор для рабочего стола типа Kwrite (или Kate, Gedit, Emacs), позволяющие выполнить первый шаг; компилятор gcc/g++ и утилита make для выполнения второго и третьего шагов; утилита gdb для отладки кода и поиска ошибок; программа построения графиков gnuplot для визуализации полученных результатов. Также с большой вероятностью имеется среда разработки типа Eclipse (или Qt) позволяющая объединить все шаги по набору и отладке программы.

Корпорация Microsoft, начиная с ОС Windows 10, поддерживает дружественные отношения с некоторыми ОС Linux и позволяет с помощью подсистемы WSL провести комфортное развертывание выбранного дистрибутива. Для более ранних версий ОС Windows (а также и для Windows 10) можно установить пакет Cygwin — бесплатно распространяемый эмулятор Linux, требующий минимальных ресурсов ЭВМ и усилий при освоении. Инсталляция Cygwin состоит из следующих шагов

(подробнее см. документацию на сайте): скачиваем с сайта `http://www.cygwin.com` подходящий файл-загрузчик и либо запускаем его с правами администратора (что обычно доставляет меньше хлопот), либо переименовываем, например в `stp64.exe`, и запускаем из командной строки cmd под обычным пользователем: `stp64.exe -- no-admin`. Выбираем директорию для установки (обычно `D:\cygwin64`), режим установки из интернета, ссылку для скачивания и указываем набор интересующих нас пакетов, отметив их галочками. На первом этапе достаточно установить:

Admin: Install;
Base: Install;
Devel: gcc-core, gcc-fortran, gcc-g++, gdb, make, mcpp;
Doc: cygwin-doc, man;
Editors: mc, vim, vim-clang-format, vim-common, vim-doc;
Graphics: gnuplot;
Net: openssh, ssh-pageant.

В результате в `D:\cygwin64` создается набор стандартных поддиректорий ОС Linux. Запустив из-под Windows пакетный файл `D:\cygwin64\Cygwin.bat`, получаем окно-эмулятор ОС Linux и возможность выполнять все команды Linux из установленных пакетов. После первого запуска Cygwin.bat создается домашняя директория `D:\cygwin64\home\YourWindowsName`, в которой стоит создать поддиректорию типа `.\Cprog\BaseTsk` для сохранения и компиляции С-программ соответствующего раздела. Отметим, что редактировать файлы удобнее из ОС Windows стандартным редактором типа Notepad++ (бесплатное ПО, `http://notepad-plus-plus.org`, по ссылке download выбираем Notepad++ zip package, распаковываем в стандартную папку, запускаем `Notepad++.exe`), а компилировать набранные С-файлы и запускать полученный в результате исполняемый файл необходимо из окна Cygwin, перейдя предварительно в соответствующую директорию `BaseTsk`. При этом стоит убедиться, что вы осуществляете компиляцию и редактирование одних и тех же файлов (например, выполнив для этого в окне Cygwin в рабочей директории `./BaseTsk` команду `cat task-1-1-1.c`). Отметим, что создаваемый в результате бинарный файл обычно запускают в среде Cygwin (так как он зависит от библиотеки `cygwin1.dll`). Для запуска в Windows достаточно скопировать файл `cygwin1.dll` в текущую

директорию (но лучше добавить в переменную PATH путь к директории `D:\cygwin64\bin`).

Если работа под ОС Linux (даже в рамках пакета Cygwin) вызывает у вас дискомфорт, то можно ограничиться установкой пакета MinGW (Minimalist GNU for Windows) — набора базовых GNU-инструментов разработки программного обеспечения для создания и запуска приложений непосредственно под Windows.

Использование сред разработки типа Visual Studio Community Edition C/C++, Code::Blocks или Dev-C++, позволяющих объединить все шаги, на наш взгляд, мешает изучению языка C, получению базовых навыков поиска синтаксических и логических ошибок в своей программе и поэтому неразумно на первом этапе обучения. В этом случае процесс компиляции и запуска сводится к заучиванию нужной последовательности кнопок. И, как писали классики современной фантастики Аркадий и Борис Стругацкие в философской повести «Пикник на обочине», получаем пользователя, который «...нажимает красную кнопку — получает банан, нажимает белую — апельсин, но как раздобыть бананы и апельсины без кнопок... не знает. И какое отношение имеют кнопки к бананам и апельсинам... не понимает». Назначение подобных программных продуктов — облегчить жизнь профессионального программиста. А поэтому ими стоит пользоваться тем, кто полностью освоил язык C и хочет сосредоточить все свое внимание либо на ускорении процесса отладки зачетных и экзаменационных задач, либо на разработке больших и сложных проектов.

На данный момент имеется огромное число онлайн-компиляторов C/C++, к примеру `http://ideone.com`, `http://rextester.com`, `https://www.onlinegdb.com`, а также профессиональных облачных IDE, например AWS Cloud9. Однако их использование обычно создает неискушенному пользователю еще больше проблем, чем среда разработки. Поэтому стоит тщательно взвесить все за и против при организации рабочего пространства.

— Давненько я не писал авторучкой. А как у нее поменять раскладку с русской на английскую?

5.1.2. Работа в консольном режиме Linux: shell, gcc, Vim

Минимальный набор консольных команд

`whoami` выдать имя пользователя («Кто я?»)
`pwd` выдать полное имя текущей директории («Где я?»)
`ls` выдать содержимое текущей директории («Что здесь?»)
`ls -a` выдать содержимое, включая скрытые файлы
`ls -l` выдать содержимое с подробной информацией
`clear` очистить экран

`mkdir Cprg` создать директорию с именем `Cprg`
`rmdir Cprg` удалить пустую директорию с именем `Cprg`
`cd ./Cprg` перейти из текущей директории в поддиректорию `Cprg`
`cd ..` перейти в директорию предыдущего уровня
`cd` перейти в домашнюю директорию

`.` условное имя текущей директории
`..` условное имя директории предыдущего уровня
`~` условное имя домашней директории

`cp a.c b.c` копировать файл `a.c` в файл `b.c`
`mv a.c b.c` переименовать файл `a.c` в файл `b.c`
`rm a.c` удалить файл `a.c`
`cat a.c` выдать на экран содержимое файла `a.c`
`less a.c` постранично выдать на экран содержимое файла `a.c` («Пробел» — следующая страница, Enter — следующая строка)
`od a.exe` выдать на экран коды символов файла `a.exe`
`file a.exe` определить и выдать на экран тип файла `a.exe`
`touch f.c` создать пустой файл `f.c`
`du -h` выдать информацию о занятом дисковом пространстве
`df -h` выдать информацию о свободном дисковом пространстве
`free -h` выдать информацию об использовании оперативной памяти

`find . -maxdepth 3 -name "tsk*.c" 2>/dev/null` найти в текущей директории и поддиректориях до указанной глубины 3 все файлы, удовлетворяющие заданному шаблону; сообщения об ошибках (т. е. поток 2) перенаправить в псевдоустройство `/dev/null`, т. е. в пустоту

`gcc a.c -c` скомпилировать файл `a.c` (создается файл `a.o`)
`gcc a.c -c 2>a.err` скомпилировать `a.c`, перенаправив

сообщения об ошибках в файл `a.err`

`gcc a.o -lm -o a.exe` слинковать файл `a.o`, подключив математическую библиотеку (создается `a.exe`)

`./a.exe` запустить файл `a.exe` из текущей директории

`Ctrl+c` аварийно завершить запущенную программу

`Ctrl+z` приостановить выполнение текущей программы, отправив в фоновый (background) режим

`fg` продолжить выполнение последней отправленной в background-режим программы

`ps` выдать список текущих процессов с указанием используемых ресурсов и значениями идентификаторов PID (Process IDentifier)

`top` выдавать список текущих процессов в режиме реального времени (выход по клавише q)

`kill -9 PID` аварийно завершить процесс с указанным PID

`kill -9 -1` аварийно завершить все доступные процессы

`man <команда>` выдать справочную информацию о команде

Большинство командных оболочек (например, `/bin/bash`) поддерживает работу с историей набранных команд, сохраняя команды в отдельном файле, например в `~/.bash_history`. Файл можно отредактировать, а можно отобразить на экране, выполнив команду `history`.

При нажатии клавиши «Стрелка вверх» в консольной строке список последовательно прокручивается от последней команды к первой, «Стрелка вниз» — обратно. При необходимости можно найти набранную ранее команду, нажав комбинацию `Ctrl+r` и набрав подстроку из искомой команды. По умолчанию показывается последняя в списке команда, где присутствует подстрока. Чтобы увидеть более ранние результаты, нужно повторно нажать `Ctrl+r`. Для выхода нажимаем `Ctrl+c`.

— Мне бы самый компактный справочник по основам работы в Linux. Есть у вас такой?
— Да, пожалуйста. Называется команда man.
— Ну нет. Мне нужен бумажный вариант.
— Без проблем: «Linux в кармане». Весит один байт, т. е. всего 2^8 страниц!

Базовые команды текстового редактора vi/Vim

Так как vi (и его продвинутая версия Vim) воспринимают нажатые клавиши через коды вводимых символов, то язык ввода имеет значение. Поэтому далее всегда считаем, что включена английская раскладка клавиатуры.

Редактор имеет два режима: командный режим и режим ввода текста (это необходимо для правильной работы на всех машинах при различных типах удаленных соединений). Информация о включении режима ввода появляется в левом нижнем углу окна редактирования. В командном режиме нажатие любой клавиши воспринимается как команда, а потому используется для перемещения по тексту, поиска и замены, выделения и копирования, вставки и удаления блоков текста. В режиме ввода нажатие любой клавиши обрабатывается как ввод текста. При базовых настройках редактора это может приводить к неожиданным эффектам: после нажатия стрелок в режиме ввода получаем некие символы, а нажатие Backspace выглядит как перемещение курсора без удаления текста. При включении продвинутых настроек редактор ведет себя более предсказуемо, но даже выйти из него на основе общих соображений обычно не удается.

При входе в файл редактор по умолчанию устанавливается в командный режим. Переход в командном режиме в командную строку осуществляется вводом двоеточия. В результате появляется приглашение : в нижней строке окна редактирования. Для выполнения команды необходимо после приглашения : набрать нужную комбинацию символов и нажать клавишу Enter. Например, для выхода из файла без сохранения внесенных изменений набираем `q!` и нажимаем Enter. Для возврата из командной строки в командный режим можно нажать клавиши Enter либо Esc.

Минимальный набор команд редактора Vim

`vim a.c`	из консоли вызвать редактор для редактирования `a.c`, файл открывается в командном режиме
`i`	в редакторе перейти из командного режима в режим редактирования с текущей позиции курсора
Esc	из режима редактирования перейти в командный режим
:	из командного режима перейти в режим командной строки
:Enter	из командной строки вернуться в командный режим
:Esc	из командной строки вернуться в командный режим

`:q!`	выйти из файла без сохранения изменений
`:w`	записать (сохранить) на диск внесенные изменения
`:wq`	записать изменения и выйти из файла
`:w b.c`	сохранить открытый файл на диск как b.c
`vim`	из консоли вызвать редактор для создания нового файла (при сохранении потребуется указать имя), файл открывается в командном режиме

Командная строка редактора Vim

`:q!`	выйти из файла без сохранения изменений
`:w`	сохранить на диск внесенные изменения
`:wq`	сохранить и выйти
`:set number`	включить нумерацию строк
`:set nonumber`	выключить нумерацию строк
`:n`	перейти на строку с номером `n`
`:/words`	найти последовательность `words` (все найденное будет подсвечено)
`:noh`	убрать вызванную поиском подсветку
`:let @/ = ""`	очистить строку поиска
`:split b.c`	открыть в редакторе второе окно с файлом `b.c`
`Ctrl+ww`	перейти из одного открытого окна в другое (если оно есть)
`:help`	открыть в редакторе второе окно с файлом `help`
`:set list`	включить отображение скрытых символов
`:unset list`	выключить отображение скрытых символов
`:e ++enc=utf8`	редактировать файл в кодировке utf8; аналогично выбираются кодировки cp1251, koi8-r, cp866
`:set fileencoding=utf-8`	конвертировать содержимое файла в указанную кодировку
`:`Стрелка вверх/вниз	листать по набранному ранее списку команд

Командный режим редактора Vim

Стрелка	передвинуть курсор в указанном направлении
`h, j, k, l`	перейти влево, вниз, вверх, вправо
`w/W`	перейти в начало следующего слова
`e/E`	перейти в конец следующего слова
`b/B`	перейти в начало предыдущего слова
`PgUp, PgDn`	сместиться на страницу вверх/вниз
`Home, End`	сместиться в начало/конец строки
`x`	удалить текущий символ (над курсором)
`X`	удалить предыдущий символ (перед курсором)

`J`	склеить через пробел текущую и следующую строки
`r<символ>`	заменить символ над курсором на указанный
`dd`	удалить строку
`u`	отменить последнее действие
`.`	повторить последнее действие
`v`	начать посимвольное выделение текста стрелками
`V`	начать построчное выделение текста стрелками
`Ctrl+v`	начать блочное выделение текста
`y`	запомнить выделенный текст в стандартный буфер
`d`	запомнить выделенный текст в стандартный буфер и стереть текст
`p`	вставить содержимое буфера (символы и блок вставятся за курсором, строки добавятся под курсором)
`P`	вставить содержимое буфера до/над курсором
`"ayy`	очистить буфер с именем `a` и запомнить в него текущую строку
`"Ayy`	добавить текущую строку в конец буфера `a`
`"Ap`	вставить содержимое буфера `a` с текущей позиции
`i`	перейти в режим редактирования с текущей позиции курсора
`I`	перейти в режим редактирования с начала строки
`a`	перейти в режим редактирования со следующей позиции
`A`	перейти в режим редактирования с конца текущей строки
`o`	вставить пустую строку за текущей строкой и перейти в режим редактирования
`O`	вставить пустую строку на место текущей строки и перейти в режим редактирования
Esc	перейти из режима редактирования в командный режим

За внешние настройки редактора, такие, как подсветка синтаксиса, отступы, кодировка, отвечает конфигурационный файл `~/.vimrc`, полезно его создать. За основу рекомендуется взять оригинальный файл `/usr/share/vim/vimrc_example.vim`, сохранив его как `~/.vimrc`. Местоположение оригинального файла может отличаться от указанного.

— Последние два года я работаю только в vi. Потому что с тех самых пор не могу из него выйти.

5.1.3. Сборка кода: gcc, make, ar

Процесс создания исполняемого файла в общем случае состоит из двух шагов: компиляции и линковки (которые выполняются одной утилитой gcc). Разберем эти шаги подробнее на примере однофайловой программы `tsk.c`.

Компиляция файла `tsk.c` (создается файл `tsk.o`, последние ключи указаны для дополнительного контроля кода):

```
gcc tsk.c -c -Wall -Wextra -Werror
```

Линковка файла `tsk.o` (создается файл `tsk.exe`, ключ `-lm` указывается при необходимости подключения математической библиотеки):

```
gcc tsk.o -lm -o tsk.exe
```

Запуск полученного кода `tsk.exe`:

```
./tsk.exe
```

Запуск в режиме потокового ввода начальных данных из файла `inputdata`:

```
./tsk.exe < inputdata
```

Запуск с перенаправлением потокового вывода в файл `outputdata`:

```
./tsk.exe > outputdata
```

Запуск с перенаправлением потоков вывода/вывода:

```
./tsk.exe < inputdata > outputdata
```

На этапе отладки программного кода для отслеживания различных проблемных моментов полезно проводить компиляцию как с использованием указанных ключей, так и с дополнительными настройками типа `-fsanitize=undefined`, `-fsanitize=address`, `-fsanitize=bounds`. Однако следует помнить, что установка некоторых флагов компиляции (например, контроля памяти) может существенно замедлить скорость исполнения полученного в результате исполняемого кода.

Отметим, что при минималистичном вызове `gcc tsk.c` осуществляется как компиляция (без использования дополнительных ключей и без сохранения на диске объектного файла `tsk.o`), так и сборка — создание исполняемого файла, обычно с именем

`a.out` (либо `a.exe`). Вызов `gcc tsk.c -o tsk.exe` приводит к сохранению исполняемого файла под указанным именем.

В некоторых случаях бывает удобно настроить утилиту gcc так, чтобы даже при вызове `gcc tsk.c` компиляция и сборка всегда осуществлялись с заранее определенным набором ключей и библиотек. Тогда, добавив ключ `-v`, можно визуализировать реальную картину происходящего: `gcc -v tsk.c`

Замечание. В действительности в цепочке создания исполняемого файла из C-файла можно выделить четыре этапа:

- препроцессирование (выполнение команд препроцессора типа `#include` в C-файле),
- компиляция (преобразование полученного полного кода на C в ассемблерный код),
- ассемблирование (преобразование ассемблерного кода в машинный объектный файл),
- линковка (сборка всех объектных файлов и стандартных библиотек в исполняемый файл).

За каждый этап отвечает свое приложение, но можно поручить весь процесс компилятору gcc и сохранить (при необходимости) результаты промежуточных шагов:

`gcc -E tsk.c -o tsk.i`	препроцессирование и сохранение результата в tsk.i
`gcc -S tsk.c`	сохранение ассемблерного кода в tsk.s
`gcc -c tsk.c`	сохранение машинного кода в tsk.o

Ничто так не ограничивает полет мысли программиста, как компилятор (экспериментально установленный факт).

Утилита make

Разумная лень — двигатель прогресса: существенно легче один раз освоить написание make-файлов и дальше этим пользоваться, чем каждый раз пошагово выполнять процедуру компиляции и линковки, рискуя что-то пропустить. Даже на однофайловой программе при использовании утилиты make процесс пересборки ускоряется в разы.

Рассмотрим типичный пример. Будем считать, что в текущей директории имеются файлы `main.c`, `fun.c`, `com.h`, содержащие соответственно функцию `main`, набор вспомогательных функций

и описание глобальных прототипов (содержимое com.h включено в main.c и fun.c с помощью директивы препроцессора #include "./com.h"). Требуется собрать исполняемый файл m.e из main.c и fun.c, зависящих от com.h. Для этого в текущей директории создадим файл Makefile следующей структуры:

```
m.e: main.o fun.o com.h
<tab> gcc main.o fun.o -o m.e -lm
main.o: main.c com.h
<tab> gcc main.c -c -Wall -Wextra -Werror
fun.o: fun.c com.h
<tab> gcc fun.c -c -Wall -Wextra -Werror
clean:
<tab> rm -f m.e main.o fun.o
```

Здесь <tab> означает символ с кодом 9, т. е. именно табуляцию, и его нельзя заменить пробелами. По сути, в Makefile в формате

```
цель: зависимости
<tab> команда
```

записано пошаговое дерево зависимостей цели-вершины m.e от файлов с исходным кодом main.c, fun.c, com.h и правила перехода по ветвям. При выполнении команды make соответствующая программа ищет в текущей директории файл с именем Makefile и анализирует дерево зависимостей, сравнивая даты изменения каждой вершины дерева и порождающих ее объектов. Если где-то нарушается естественная упорядоченность по времени, то соответствующая ветвь пересобирается. В данном случае изменение com.h приведет к полной пересборке дерева, а изменение main.c потребует выполнения только двух первых команд.

При выполнении make clean содержимое файла Makefile обрабатывается только для цели clean, отвечающей в данном случае за удаление файлов, полученных в результате компиляции. Команда make -f filename означает, что утилита make должна обрабатывать дерево зависимостей из файла с именем filename.

Конечно, для разобранного примера можно обойтись однострочной командой

```
gcc -Wall -Wextra -Werror main.c fun.c -o m.e -lm
```

Можно, впрочем, и реализовать весь код в одном файле или просто в единственной функции **main**. Однако при таком подходе разработка серьезных проектов становится невозможной.

Перепишем исходный make-файл с использованием механизма внутренних переменных, что позволит легко подстраивать его под текущие задачи, а также укажем еще несколько полезных ключей компиляции.

```
CC = gcc
CFLAGS = -Wall -Wextra -Werror -Wpedantic\
                -Wconversion -Wsign-conversion \
                -Wuninitialized -Wunused-variable
OBJS = ./main.o ./fun.o

m.e: $(OBJS)
<tab> gcc $(OBJS) -lm -o m.e
main.o: main.c com.h
<tab> gcc main.c $(CFLAGS) -c
fun.o: fun.c com.h
<tab> gcc fun.c $(CFLAGS) -c
clean:
<tab> rm -f m.e $(OBJS)
```

— Так что делает твоя программа?
— Сейчас запустим и узнаем!

Утилита ar

Для многократного использования собственных функций полезно организовать личную библиотеку. В простейшем случае для создания статической библиотеки в некоторой директории (для определенности в директории с именем **~/CProjects/Lib/Al**) подготавливаем файл (например, **al.h**) с прототипами функций и набор соответствующих obj-файлов (например, **1.o**, **2.o**, **3.o**). Затем в указанной директории выполняем команду **ar -crs libal.a 1.o 2.o 3.o**. В результате создается индексированный архивный файл **libal.a**. Теперь файлы ***.o** могут быть удалены. Пусть в директории **~/CProjects/Lib/Bl** аналогично созданы файлы **bl.h** и **libbl.a** еще одной библиотеки. Поправим **Makefile**

предыдущего раздела, считая, что `main.c` и `fun.c` используют функции из указанных библиотек, а прототипы подключены стандартным образом: `#include<al.h>`, `#include<bl.h>`.

```
CC = gcc
CFLAGS = -O3 -Wall -Wextra -Werror
HDIRS = ~/CProjects
LADIRS = $(HDIRS)/Lib/Al
LBDIRS = $(HDIRS)/Lib/Bl
INCS = -I$(LADIRS) -I$(LBDIRS)
DIRS = -L$(LADIRS) -L$(LBDIRS)
NAMES = -lal -lbl
OBJS = ./main.o ./fun.o

m.e: $(OBJS)
<tab> gcc $(OBJS) -lm $(DIRS) $(NAMES) -o m.e
main.o: main.c com.h
<tab> gcc main.c $(INCS) $(CFLAGS) -c
fun.o: fun.c com.h
<tab> gcc fun.c $(INCS) $(CFLAGS) -c
clean:
<tab> rm -f m.e main.o fun.o
```

Отметим, что на этапе линковки библиотеки указываются после использующих их файлов и в именах библиотек опущены приставки `lib`.

Что общего между магами и программистами — и те и другие шепчут под нос непонятные слова, делают загадочные пассы руками и... в результате не могут внятно объяснить, почему же все это работает.

5.1.4. Отладка кода: gdb

Базовая схема работы с отладчиком GNU Debugger (GDB) выглядит следующим образом. Исходные С-файлы компилируем с ключом `-g` без оптимизации (в результате в `main.o` сохраняется таблица соответствия со строками `main.c`); линкуем и запускаем gdb, указав имя исполняемого файла в качестве параметра; на приглашение (gdb) набираем команду `run`, нажимаем Enter и, если требуется, вводим данные с клавиатуры.

Опишем каждый шаг подробнее. Компиляция файла с ключом отладки `-g` и запретом на оптимизацию кода:

```
gcc main.c -Wall -Wextra -c -O0 -g
```

Сборка:

```
gcc main.o -o main.e
```

Запуск gdb и программного кода под его управлением:

```
gdb ./main.e
(gdb) run
```

В случае «падения» программы на экран выдается информация о типе проблемы и соответствующая строка С-кода. Возможно, для этого после «падения» потребуется выполнить

```
(gdb) where
```

Для отображения С-кода из окрестности текущей строки следует набрать

```
(gdb) list
```

Также можно распечатать содержимое всех локальных переменных:

```
(gdb) info locals
```

либо интересующих нас переменных:

```
(gdb) print i
(gdb) print a[i]
```

Если этого окажется недостаточно, то бывает полезным потребовать выдать стек вызовов функций:

```
(gdb) backtrace
```

и, перемещаясь по хранящимся в стеке фреймам, выяснить (с помощью указанных далее механизмов) причины сбоя. Отметим, что большинство команд gdb допускают одно- и двухбуквенные сокращения.

Стартовые команды gdb

`run`	запустить загруженный исполнимый файл
`quit`	выйти из отладчика
`Ctrl+d`	выйти из отладчика
`help`	обзор команд

Команды для пошаговой отладки

`start`	поставить временную точку останова на main и запустить программу
`step`	выполнить шаг вперед по программному коду
`step 3`	выполнить три шага вперед
`stepi`	выполнить одну инструкцию следующей строки кода
`next`	выполнить строчку кода, в отличие от step, вызов функции отрабатывается за один шаг
`next 5`	выполнить пять строчек
`until 100`	выполнить программу до указанной строчки
`finish`	продолжить выполнение до выхода из текущей функции
`continue`	продолжить выполнение программы
`Ctrl+c`	приостановить работу gdb и получить доступ к его командной строке
`info program`	показать состояние программы

Команды контроля переменных

`info locals`	напечатать содержимое всех локальных переменных
`print mas[i]`	напечатать содержимое указанной переменной
`print &mas[i]`	напечатать адрес переменной
`ptype i`	напечатать тип переменной
`x &i`	вывести содержимое ячейки памяти, расположенной по указанному адресу (количество байтов и формат вывода берутся по умолчанию)
`x/t &i`	вывести содержимое памяти в двоичном формате (другие форматы: o — восьмеричный, x — шестнадцатеричный, c — символьный)
`x/4 &i`	вывести содержимое 4 байт (формат вывода берется по умолчанию)
`x/4t &i`	вывести в двоичном формате содержимое 4 байт (на процессорах x86 байты хранятся в порядке little-endian)
`set var i = 3`	задать новое значение переменной i
`list`	напечатать текущую часть листинга программы
`list 20, 35`	напечатать указанную часть листинга

`list main.c:f1` напечатать листинг функции `f1()` из файла `main.c`

Если в процессе выполнения кода вызывается некоторая функция, то вся необходимая для последующей работы информация (адрес возврата, параметры вызова, содержимое локальных переменных) сохраняется в отдельном блоке, называемом кадром. Кадры последовательно добавляются в так называемый стек вызовов функций. Анализ стека вызовов позволяет проследить все шаги работы программы до точки останова.

`backtrace` напечатать весь стек вызовов функций
`backtrace 5` напечатать указанную часть
`backtrace -10` стека вызовов функций
`up, down` перейти по фреймам стека
`frame 2` перейти к указанному фрейму
`info frame` выдать информацию о текущем фрейме
`info args` показать аргументы в текущем фрейме

Задание дополнительных точек останова (breakpoint)

`break main` определить точку останова в начале функции main
`break main.c:23` в 23-й строке файла main.c
`break main.c:fun` в начале функции fun
`run` запустить программу до первой точки останова
`break 33` определить точку останова в строке 33
`break 33 if i == 12` в строке 33, если переменная i равна 12
`watch i` определить точку останова на изменение переменной
`rwatch i` на считывание переменной
`awatch i` на считывание/изменение переменной
`hbreak 5` определить аппаратную точку останова в 5-й строке
`info break` распечатать листинг имеющихся точек останова
`del 1` удалить точку останова с номером 1
`d` удалить все точки останова
`disable 1` временно отключить точку останова 1
`enable 1` включить точку останова 1

Разберем подробнее работу с gdb на следующем примере некорректной инициализации динамически созданного массива.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int *a = NULL;
  unsigned int n = 1024;
  int i = 1024;
  a = (int *)malloc(n * sizeof(int));
  while (1) {
    i--;
    a[i] = 1000000;
    if (i < 0) break;
  }
  free(a);
  a = NULL;
  return 0;
}
```

Данный код, собранный и запущенный под Linux, «падает» с сообщением `Aborted (core dumped)`. Стандартная последовательность действий позволяет найти проблемную строку:

```
gcc test.c -Wall -Wextra -c -O0 -g
gcc test.o -o test.e
gdb ./test.e
(gdb) run
(gdb) bt
...
#6 0x00000000004011ac in main () at test.c:13
```

Конечно, закомментировав оператор `free(a)`, можно «замылить» ошибку, утверждая, что проблема в библиотечной функции, а с библиотечными функциями должны разбираться разработчики gcc. Однако в данном случае аварийный останов происходит не из-за ошибок в функции `free` — времена вылавливания тривиальных багов в компиляторе давно прошли, — а в результате порчи информации, необходимой для корректного высвобождения памяти. Такую информацию записывает функция `malloc` непосредственно перед блоком выделенной памяти (в данном случае перед нулевой ячейкой `a[0]`).

Следующая последовательность набранных в gdb команд позволяет выявить проблему.

```
(gdb) list // печать С-кода
(gdb) break 7 // задаем точку останова
// на операторе выделения памяти
Breakpoint 1 at 0x401165: file test.c, line 7.
(gdb) run
Breakpoint 1, main () at test.c:7
7 a = (int *)malloc(n * sizeof(int));
(gdb) step
9 i--;
(gdb) print a // получаем адрес начала
                  // выделенного блока
$1 = (int *) 0x17c62a0
(gdb) watch *((int *) 0x17c62a0 - 1)
// задаем точку останова на изменение
// содержимого ячейки перед a[0]
Hardware watchpoint 2: *((int *) 0x17c62a0 - 1)
(gdb) continue
Continuing.
Hardware watchpoint 2: *((int *) 0x17c62a0 - 1)
Old value = 0
New value = 1000000
main () at test.c:11
11 if (i < 0) break;
(gdb) p i
$2 = -1 // теперь понимаем, что в строке 10
// происходит некорректное действие при i = -1
(gdb) quit
```

Подчеркнем, что программа gdb является мощным и неприхотливым инструментом разработчика, позволяющим отслеживать работу программного кода огромного размера, прицепляться к уже запущенным процессам, отлаживать многонитевые программы, контролировать содержимое регистров, а также анализировать файл `core dump` упавшего ранее процесса (если системные настройки обеспечили сохранение такого файла на диске).

Найти ошибки в коде бывает непросто. Особенно если вы уверены, что их там нет.

5.1.5. Визуализация результатов: gnuplot

Умение наглядно представить полученные результаты является необходимым не только на заключительном этапе решения задачи, но и в процессе отладки и верификации программы. Для этих целей прекрасно подходит бесплатно распространяемый пакет **gnuplot**, который можно установить под Windows (обычно просто развернув архивный файл) и который входит в базовый набор всех стандартных версий Linux.

Утилита gnuplot поддерживает консольный и пакетный режимы. В первом случае после запуска gnuplot требуемые команды (инструкции) вводятся непосредственно в командной строке (при запуске оконной реализации их также можно ввести нажатием соответствующих кнопок). В пакетном режиме требуемая последовательность команд заранее набирается в некоторой директории (например, с полным именем **/home/user/numres**) в текстовом файле (например, в **Fname.gpi**, рекомендуемые расширения **.plt**, **.gpi**). Команды набираются либо по одной в строке, либо разделяются символом точка с запятой. Комментарии начинаются с символа **#**. Затем либо в консоли набирается **gnuplot /home/user/numres/Fname.gpi**, либо после запуска gnuplot файл загружается инструкцией **load '/home/user/numres/Fname.gpi'**. Строка с полным именем файла не должна содержать символов национальных алфавитов. При желании после запуска gnuplot можно определить имя текущей директории командой **pwd**, а затем сменить текущую директорию командой **cd '/home/user/numres'** и выполнить **load 'Fname.gpi'**. Также бывает удобно скопировать инструкции из файла в буфер обмена и вставить содержимое буфера в командную строку **gnuplot**. Набранные в текущем сеансе команды сохраняются в буфере и могут быть повторно вызваны нажатием клавиш «Стрелка вверх» и «Стрелка вниз».

Для построения графиков, по сути, имеются две команды: **plot** для работы с функциями одной переменной и **splot** для функций, зависящих от двух переменных. Остальные команды отвечают за стилевую настройку режима визуализации. Приведем скромный минимум инструкций пакета gnuplot без подробных пояснений, так как их назначение становится понятным после первого использования. Отметим, что указанные далее команды набираются после запуска gnuplot, т. е. в gnuplot-консоли.

1. Построение аналитически заданных функций $y = f(x)$.

```
plot sin(2*x)
set xrange[-1.:2.]
set yrange[-1.5:1.5]
replot
reset
set grid
plot sin(2*x) linetype 5 linewidth 3, \
x**3 linetype 2 linewidth 7
unset grid
plot sin(2*x) with points title "Graph"
exit
```

2. Работа с параметрически заданными кривыми.

```
set parametric
plot [0:2*pi] sin(t),cos(t)
# зададим равный масштаб по осям координат:
set size ratio 1
replot
r(t) = 1+cos(t)
plot [0:5*pi] r(t)*cos(t),t*sin(t)
```

3. Опишем процесс визуализации одномерных дискретно заданных функций. Пусть имеется файл `~/numres/a.dat` с двумя столбцами равной длины; в первом содержатся значения дискретной координаты x_i, а во втором — соответствующие им значения $f(x_i)$. Пусть файл `~/numres/b.dat` состоит из нескольких столбцов одинаковой длины, при этом координаты x_i записаны в третьем, а $g(x_i)$ — в четвертом столбце. Для визуализации заданных таким образом табличных функций можно использовать следующие команды:

```
cd '/home/user/numres'
pwd # выдаем имя текущей директории
plot 'a.dat'
plot 'b.dat' using 3:4
plot 'a.dat' with lines, \
'b.dat' using 3:4 with dots
set style data linespoints
plot 'a.dat', 'b.dat' using 3:4
```

4. Для сохранения построенного графика в отдельном файле, например в формате png, eps, gif и т. д., можно поступить следующим образом:

```
# печатаем тип терминала:
show terminal
# обычно в Linux получим: terminal type is x11
# в ОС Windows: terminal type is wxt 0
set size ratio 1
set terminal png; set out 'a-b.png'
plot 'a.dat', 'b.dat' using 3:4
set terminal postscript "Times-Roman" 30 lw 3
set out 'a-b.eps'
replot
# для последующей работы в консоли:
set out
set term x11 # в Linux
```

5. Для построения графиков аналитически заданных функций от двух переменных применяется (возможно, с использованием указанных ранее настроек) команда splot. Рассмотрим следующий пример. Известно, что функция Розенброка $(1-x)^2+100(y-x^2)^2$ является хорошим тестом для алгоритмов поиска экстремумов, так как имеет единственный минимум в точке $(1,1)$, равный 0, и овражистые линии уровня в его окрестности. Продемонстрируем это графически.

```
#отрисовка графика при стандартных настройках:
splot (1-x)**2+100*(y-x**2)**2
#локализация области:
set xrange[-2:2]; set yrange[-0.5:3]; replot
#увеличение числа расчетных точек:
set isosample 50, 75; replot
#отрисовка линий уровня на плоскости:
set contour base; set cntrparam levels 15; replot
#убираем поверхность и ось z:
unset surface; unset ztics; replot
#смена координат легенды и угла обзора:
set key 2.8, 3; set view 0,0; replot
```

Приведем еще несколько команд, полезных для визуализации трехмерных объектов.

```
splot exp(sin(x/2)*cos(y))
set isosample 50, 100; replot
set format z "%.2f"; replot
set hidden; replot
set pm3d; replot
set view map; replot
set contour surf; set key 0,13; replot
```

6. Для таблично заданной функции (в том числе многозначной типа $f(x,y) = \pm\sqrt{x^2+y^2}$) можно создать файл `c.dat` с тремя столбцами, содержащими координаты x_i, y_j и соответствующие значения $f(x_i, y_j)$, а затем выполнить

```
splot 'c.dat'
```

В этом случае поверхность отображается как «облако» несвязанных точек. Если при этом приложение gnuplot запущено в «оконном» режиме, то полученный график можно «зацепить» мышкой, нажимая и удерживая левую кнопку. Это позволит повернуть картинку на требуемый угол, а также приблизить/отдалить изображение. Аналогичный результат можно обеспечить в командном режиме, изменив заданный по умолчанию угол обзора (`60.0,30.0`) направив, например, ось z вниз и в два раза приблизив картинку:

```
set view 300.0,30.0, 2.0
splot 'c.dat'
```

7. Для отображения дискретно заданной функции в виде поверхности можно сформировать файл `d.dat`, содержащий только прямоугольную матрицу значений f_{ij} и выполнить

```
splot 'd.dat' matrix with lines
```

В этом случае f_{ij} отображаются над целочисленной сеткой (i,j), соответствующей индексам элемента f_{ij}. Отметим, что элемент f_{00} располагается в левом нижнему углу экрана, т. е. первая строка файла отображается на экране последней.

8. Для построения таблично заданного двумерного векторного поля следует сохранить в некотором файле `v.dat` четыре столбца: первые два — координаты точек на плоскости, вторые два — длины горизонтальной и вертикальной компонент вектора, а далее, например, выполнить команду

```
plot 'v.dat' using 1:2:3:4 with vectors \
      head filled lt 2
```

9. Если требуется визуализировать динамику зависящей от времени табличной функции одной переменной, то в процессе расчета можно, например, создать файлы с именами a_001, a_002, a_003 одного из указанных выше форматов, соответствующие моментам времени 1, 2, 3. Далее записать, например в файл dyn_a.gpi (вручную, но лучше автоматически в процессе расчета), строки

```
# фиксируем диапазон:
set yrange[-2:5]
# дополнительная интерполяция по узлам:
set style data linespoints
plot 'a_001'; pause 2
plot 'a_002'; pause 2
plot 'a_003'; pause 2
```

и в gnuplot набрать команду

```
load 'dyn_a.gpi'
```

Аналогичный результат обеспечивают набранные в gnuplot команды

```
set yrange[-2:5]; set style data linespoints
do for [i=1:3] {plot 'a_00'.i; pause 2}
```

Для создания и сохранения в текущей директории соответствующего «мультика» в командной строке gnuplot выполняем

```
set term gif animate optimize delay 50 \
loop 1 size 640, 480
set output 'aT.gif'
do for [i=1:3] {plot 'a_00'.i w lp title 'a_00'.i}
set out
set term x11 # в Linux
```

В данном случае кадры файла aT.gif отделены паузой в пятьдесят тактов и установлено однократное повторение полученного изображения. Конечно, технологичнее будет набрать в файле dyn_a_gif.gpi указанные команды и выполнить gnuplot dyn_a_gif.gpi в Linux-консоли.

Построение анимационного файла с двумерным полем скоростей (см. п. 8) по имеющимся данным `v1_dat`,..., `v100_dat` можно реализовать следующим образом:

```
set term gif animate opt delay 50 \
loop 1 size 640, 480
set output 'Tailor2Couette.gif'
do for [i=1:100] { plot 'v'.i.'_dat' u 1:2:3:4 \
with vectors head \
filled lt 2 title 'v'.i.'_dat'}
set out
```

10. Для построения анимационных картинок (см. п. 9) в некоторых случаях более удобным может оказаться следующий прием. По каждому из файлов `a_001`, `a_002`, `a_003` строим (см. п. 4) соответствующую ему gif-картинку: `a_001.gif`, `a_002.gif`, `a_003.gif`. Далее Linux-утилитой convert объединяем все в один анимационный gif:

```
convert -delay 15 -loop 5 *.gif aT.gif
```

В данном случае кадры файла `aT.gif` отделены паузой в пятнадцать тактов и установлено пятикратное повторение полученного изображения. Если значение ключа `-loop` равно нулю, то число повторений картинки не ограничено.

11. Пакет gnuplot активно развивается, а поэтому синтаксис отдельных команд может зависеть от текущей версии. Например, для детального ознакомления с возможностями gnuplot под Windows можно загрузить подготовленные разработчиками пакета командные демофайлы:

```
cd 'D:\Utils\gnuplot\demo'
load 'airfoil.dem'
load 'all.dem'
```

Можно также самостоятельно разобрать синтаксис каждой конструкции, вызвав встроенную команду `help`. Например,

```
help set contour
```

Из апелляции: «Моя ошибочная программа совсем не ошибочная. Просто она решает не совсем ту задачу».

5.2. Частые вопросы и полезные советы

В программистском фольклоре есть много шуток о том, что компьютерная программа на любой стадии готовности, в том числе и в финальной версии, гарантированно содержит ошибку, и, как правило, не одну.

Это связано с тем, что значительная часть времени, затрачиваемого на разработку и реализацию программы, уходит именно на поиск и устранение ошибок. В данном разделе мы собрали несколько типичных ситуаций, с которыми почти наверняка столкнется начинающий программист, и обсуждаем, как с ними можно справиться. Например, как найти и исправить ошибки и как поступать, чтобы уменьшить вероятность их появления.

Раздел построен в форме вопросов и ответов и представляет собой взгляд авторов на стиль и технологию начального программирования в процессе обучения. Поэтому мы постарались представить здесь самые простые методы решения проблем, которые, тем не менее, могут оказаться в некоторых случаях вполне эффективными и полезными и для более профессионального и производственного уровня работы. Советы советами, но документацию никто не отменял. Поэтому мы настоятельно рекомендуем не ограничиваться только приведенными здесь ответами, а прочитать руководства и инструкции по затронутым темам.

Мы начнем с пары глобальных и отчасти философских вопросов, которые традиционно каждый новый учебный год у кого-то да возникают.

Я вообще ничего не понимаю. Что делать и как быть? В качестве ответа на этот вопрос можно привести перефразированное название одной из книг известного американского психолога Д. Карнеги, а именно: надо «успокоиться и начать жить». Действительно, программирование — это сумма некоторых навыков и приемов, которые осваиваются в ходе регулярных тренировок. Это как ходить и говорить — со временем все нормальные дети обучаются и первому, и второму. Вопрос состоит только в том, как сделать процесс обучения более коротким и эффективным. И главное в этом процессе — это регулярность и приложение собственных усилий. Прежде всего надо выяснить, в чем суть проблемы с программированием, чем она вызвана. Часто упоминаются следующие причины: слабая школьная подготовка, непонимание синтаксиса и конструкций алгоритмического

языка, неумение переложить понятный математический алгоритм в последовательность формальных шагов, общая растерянность при работе с компьютером. Как видите, все эти причины связаны с отсутствием опыта, знаний и умений. А самый быстрый путь к обретению опыта — это практика и советы знающих людей. Поэтому обзаведитесь источниками информации: справочными руководствами, знающими товарищами, преподавателями. Беритесь за конкретные задачи (от простых к более сложным), пытайтесь их решить самостоятельно, пытайтесь понять, что не дает добиться результата, задавайте конкретные вопросы, проясняйте все детали данных вам советов, не оставляйте неясных моментов.

Очень часто процесс обучения программированию строится на основе *самостоятельного воспроизведения* предложенных готовых конструкций: Look for the Sources to achieve the new Forces! Поэтому *изучайте* чужой код, старайтесь разобраться, для чего нужна каждая строка чужой программы и почему она записана именно так. Пробуйте доводить чужой код до состояния рабочей программы. Изменяйте его и смотрите, что при этом получатся, как меняется результат. Пытайтесь его модифицировать под свои нужды. Помните, что вам должно быть совершенно понятно назначение каждого оператора и вы должны уметь самостоятельно воспроизвести решение.

Используйте все доступные источники информации. Главное — не откладывать разбор неясных моментов на потом, иначе возрастающие требования к программам потребуют использования сложных конструкций, разобраться в тонкостях которых будет проблематично. Программирование — это как иностранный язык, как общая физическая подготовка. Его нельзя выучить, освоить, набрать «форму» за пару дней, нужна постоянная практика, пусть понемногу, но каждый день. Тогда по прошествии определенного времени все встанет на свои места и сложится, приемы войдут в привычку, стандартные решения станут воплощаться без трудностей, построение программы по понятному алгоритму будет раскладываться на знакомые шаги без особых проблем. При этом рекомендуется постепенно наращивать сложность решаемых задач, не оставляя «за спиной» неосвоенные разделы и приемы.

Не пользуйтесь «копипастом» для минимизации усилий по набору текста для последующей мелкой модификации, даже если оригинальный текст ваш собственный. Как и выдергивание

цитаты, это обычно приносит массу проблем: абсолютно корректный в контексте исходной программы код на новом месте может стать ошибочным. Другая опасность — легко разучиться правильно использовать конструкции и возможности языка. Типичным примером является работа с динамически выделяемой памятью (функциями типа `malloc`, `free`).

Целый месяц (четверть, семестр, год) так делал, а получил незачет (два). Обратите внимание, что итоговый уровень вашего мастерства будет определяться количеством лично вами набранного («копипаст» не в счет) и полностью отлаженного кода. Минимальный входной барьер составляет порядка пяти сотен строк (15—30 задачек), осмысленное программирование начинается после пары тысяч строк, комфортный уровень наступает после пяти-десяти тысяч строк (когда вами реализованы три сотни задач).

Постановка задачи вроде бы понятна, а как писать программу — непонятно. С чего начать? Начать стоит с понимания того, что при написании программы мы вынуждены реализовать алгоритм с помощью тех инструментов и конструкций, которые есть в нашем распоряжении. Говорят, что программа — это алгоритм + данные. Поэтому первый шаг — решить, какие данные и каким образом должны быть представлены в алгоритме. Начальный уровень программирования предполагает использование переменных и массивов определенных базовых типов. Поэтому нужно распланировать, какие переменные и массивы будут представлять исходные данные, какие будут выражать результат, какие потребуются для промежуточных вычислений, откуда эти данные будут браться, куда будет передаваться результат.

Второй шаг — разработка алгоритма. Необходимо сформулировать, к каким преобразованиям данных сводятся отдельные шаги программы, какие формулы позволяют вычислить одни данные по другим, как можно добраться до значений, используемых в вычислениях, какие промежуточные результаты и в какой форме придется получать и сохранять для работы и как это позволит учесть разнообразные условия, присутствующие в логике алгоритма. Алгоритм решения задачи необходимо выбирать самым тщательным образом. Если сейчас нет желания детально продумать каждый шаг, то будьте готовы выделить время на переделку всей программы. К моменту набора

программного кода все должно стать предельно ясным. Если не получается четко изложить алгоритм своими словами, то уж точно не удастся переложить его на C/C++.

Следующий этап — понять, какие шаги можно выполнить имеющимися стандартными процедурами (функциями), а какие придется реализовывать самому. При использовании имеющихся библиотечных процедур нужно разобраться, как их применить, что передать в качестве параметров, как добиться того или иного результата, какие побочные эффекты могут возникнуть (одним словом, следует прочитать инструкцию, user manual).

Ну и последний этап — это реализация и отладка программы, т. е. запись действий с помощью формальных конструкций языка программирования и тестирование программы в рамках имеющейся системы программирования. Для этого потребуется справочная информация (из учебников, с форумов, от знатоков) и умение ее использовать.

Так с чего же все-таки начать? С того что приложить эти общие этапы к каждой самой элементарной задаче и программе, какие будут вам встречаться в процессе обучения. Даже вычисление суммы двух чисел можно разобрать по этой схеме и написать программы разного уровня сложности, надежности, эффективности и назначения.

— Вот чтобы новый язык изучить, советуют фильмы на нем смотреть, сериалы всякие... Дайте что-нибудь на C++!

5.2.1. Стиль написания кода

Какой стиль написания кода лучше использовать? Несмотря на то что компиляторы C/C++ обычно не накладывают ограничений на форму записи строк кода, рекомендуется всегда записывать текст программы с использованием определенных правил форматирования. Это позволяет улучшить читаемость программы и тем самым ускорить поиск отдельных конструкций и частей алгоритма, облегчить понимание их сущности и назначения.

В профессиональных коллективах часто принимают достаточно жесткие и подробные правила форматирования кода, которые должны соблюдаться всеми программистами. Но для начала можно ограничиться следующими базовыми требованиями.

Все блочные конструкции (функции, циклы, `if-else`, `switch` и т. п.) должны быть оформлены по четким правилами соответствия открывающей и закрывающей скобок, а операторы содержимого блока выделены дополнительными отступами. Есть два основных стиля форматирования блоков. В первом варианте открывающая и закрывающая скобки находятся одна под другой, а во втором варианте закрывающая скобка выравнивается на первое слово конструкции. Например:

```
for (i = 0; i < n; i++)
{
    a[i] = 2 * i;
}
if (x > 0)
{
    y = 1;
}
else
{
    y = 2;
}
```

```
for (i = 0; i < n; i++) {
    a[i] = 2 * i;
}
if (x > 0) {
    y = 1;
} else {
    y = 2;
}
```

Профессиональные стандарты рекомендуют применять строго один стиль по всему коду (в некоторых случаях допускается зафиксировать свой стиль для каждой конструкции). Но главное — необходимо сделать так, чтобы каждый блок кода был четко визуально выделен и поиск его начала и конца не вызывал трудностей. Количество пробелов в отступах не так принципиально, но оно должно быть одинаковым по всему тексту. Обычно делают четыре пробела, но некоторые предпочитают восемь пробелов (или табуляцию), хотя это растягивает код в ширину и не всегда хорошо смотрится.

Следующее правило — отделять отдельные смысловые группы кода пустыми строками. Такими группами, например, являются функции, объявления переменных, вычислительные операторы, относящиеся к некоторому смысловому фрагменту алгоритма, операциям ввода или вывода данных и т. п. Часто такие обособленные группы строк кода снабжаются комментариями-заголовками (см. вопрос про комментарии).

Использование пробелов вообще является общепринятым средством для улучшения читаемости программ. Пробелами

часто выделяют арифметические и логические операции в выражениях, операцию присваивания. Пробелы используются для визуального выравнивания нескольких строк кода, стоящих одна под другой, для лучшего их представления в тексте программы.

Еще одно правило, которое может быть полезным, звучит так: «Один оператор — одна строка». На самом деле это правило иногда трудно выполнить (оператор очень длинный), а иногда оно приводит к столбику коротеньких операторов, выводящих единый фрагмент кода за границы экрана. Поэтому подчеркнем еще раз: чувство меры и стремление к главной цели — повышению читаемости программы.

Помните, что скобки не только повышают читабельность кода, но и помогают избежать трудно вылавливаемых ошибок:

```
for (i = 0; i < n; i++); // пустой цикл
a = b + c<<1; // соответствует a = (b + c)<<1;
```

Описывать здесь все разнообразные варианты стилевого оформления кода программы, наверное, нет особого смысла. Изучайте примеры кодов в учебниках, в сети, у своих знакомых и выбирайте те приемы, которые кажутся вам наиболее эффективными для улучшения читаемости программы. Или выполняйте те требования, которые предъявляет заказчик (руководитель рабочей группы, преподаватель). Познакомиться со стандартными стилями форматирования (или просто «причесать» свой текст) можно с помощью специальной программы clang-format. Вызов

```
clang-format -style=Google tsk.c > tsk_G.c
```

запишет отформатированный код вашей программы `tsk.c` в указанный файл `tsk_G.c` в соответствии с правилами стиля Google для C++. Общедоступными являются следующие стили: `LLVM`, `Google`, `Chromium`, `Mozilla`, `WebKit`, `Microsoft`, `GNU`. Для просмотра правил конкретного стиля можно создать файл настроек:

```
clang-format -style=GNU -dump-config > .clang-format
```

Полученный файл `.clang-format` можно посмотреть, отредактировать (не нарушая общей структуры) и далее использовать в работе:

```
clang-format -style=file tsk.c > tsk_GNU.c
```

Для переформатирования исходного файла tsk.c выполняем

```
clang-format -style=file -i tsk.c
```

Как комментировать код? Комментарии используются для разъяснения фрагментов кода, поэтому нет твердых правил, когда их надо добавлять, а когда нет. Иногда говорят, что код ясен сам по себе и не требует пояснений. Иногда, наоборот, комментируют каждый оператор. Поэтому надо просто понять, кому и зачем надо объяснять данный код. То, что очевидно автору программы, может быть совсем не ясно другому человеку, который пытается эту программу прочитать и с ней разобраться. Тем не менее есть несколько общепринятых правил размещения комментариев.

В самом начале файла с программой размещается несколько строк с описанием назначения программы, упоминанием автора, возможными описаниями или ссылками на реализованный алгоритм, указанием версий и истории модификации кода, даты выпуска и т. п. — «шапка программы». Этот комментарий позволяет быстро понять, для чего вообще предназначен данный код и кто несет за него ответственность.

Полезны комментарии при объявлении констант, переменных, функций, существенных для понимания алгоритма, комментарии, отсылающие к описаниям алгоритма в литературе, поясняющие суть отдельных шагов. Иногда эту роль выполняют «говорящие» имена переменных или функций (см. соответствующий вопрос). Наоборот, если переменная имеет локальное вспомогательное значение (например, счетчик цикла), то этот факт разъяснять комментарием не стоит, это просто загромоздит код.

Стоит предварить комментарием группу операторов, выполняющих определенную логическую часть обработки данных, например: «ввод исходных данных», «проверка корректности данных», «решение основного уравнения», «поиск минимального значения», «формирование результата», «вывод результата», «освобождение ресурсов» и т. п. Такие комментарии часто пишутся даже до начала кодирования и являются планом построения алгоритма, который потом реализуется в конкретных операторах. Насколько детальным должно быть такое комментирование — это, как всегда, вопрос сложности и специфики алгоритма и чувства меры программиста.

Комментарии также должны помочь вам вспомнить, о чем была программа, написанная вами некоторое время (день,

месяц, несколько лет) назад, чтобы вы смогли в ней полностью разобраться, свободно применять и модифицировать.

Как выбирать имена для функций и переменных в программе? Хорошим индикатором правильного стиля программирования являются «говорящие» имена объектов программы: имя переменной или функции содержит информацию о назначении объекта. В программистских коллективах применяются разные соглашения, и получить представление об этом можно, читая «профессиональные» коды. Не вдаваясь в описания конкретных стилей, выделим их некоторые характерные черты.

Имена функций принято составлять из слов, характеризующих назначение функции, причем эти слова часто записывают с заглавной буквы, например: `SolveQuadraticEquation()`, `GaussIntegration()`, `DistancePointToPlane()`. В результате имя функции не смешивается в выражении с именем переменной и сразу понятно, какого рода операция выполняется в этом месте кода. Конечно, имена функций не следует делать слишком длинными, можно сокращать слова до разумных пределов. Иначе вместо облегчения восприятия мы получим загромождение кода.

Имена переменных также могут составляться из нескольких слов, описывающих сущность объектов. Но в случае переменных стараются не давать длинных имен, чтобы не загромождать выражения с такими переменными. Если общепринятые описания алгоритмов содержат простые обозначения, то лучше использовать их. Например, сравните

```
double **a // матрица системы уравнений
.....
a[i][j] = a[i][j+1] - a[i+1][j];
```

и

```
double ** matrixOfSystem;
.....
matrixOfSystem[i][j] = matrixOfSystem[i][j+1]
                     - matrixOfSystem[i+1][j];
```

Конечно, если в алгоритме имеются близкие по смыслу переменные (например, две матрицы), то говорящие имена очень удобны. Например, в программе по вычислительной геометрии могут встретиться переменные

```
int numPoints, numSides;
double centerX, centerY;
double length, distToPlane;
```

Обратите внимание, что первая буква имени переменной, в отличие от имени функции, оставляется строчной. Это позволяет легко различать имена переменных и имена функций в сложных выражениях.

При именовании переменных также часто используются префиксы и суффиксы для отдельной спецификации типов или отношений переменных. Например, префикс «p» часто используется для обозначения переменных-указателей. Функция, определяющая минимальный элемент массива и возвращающая индекс этого элемента через параметр-указатель, может иметь следующий прототип:

```
double MinValue(int numElems,
                double *pArray, int *pMinInd);
```

Префиксы и суффиксы в именах могут использоваться для обозначения групп, отношений, сущностей переменных. Так, префикс «n» удобно использовать для обозначения разнообразных количеств. Тогда в двух примерах выше переменные могли бы получить имена `nPoints`, `nSides`, `nElems`. Отметим, что не рекомендуется использовать в качестве префикса для своих нужд символ подчеркивания.

Так что читайте чужие программы, анализируйте примеры именования объектов и формируйте собственный стиль, удобный вам и понятный окружающим.

Зачем писать программный код в нескольких файлах? Когда программный код разрастается, работа с одним файлом может оказаться просто неудобной: «перематывать» текст от начала к концу и обратно будет долго, искать нужное место в коде трудно и т. д. Работа в коллективе также становится невозможной из-за проблем с синхронизацией изменений программы, внесенных разными разработчиками. Поэтому все языки и системы программирования предполагают возможность размещения программы в нескольких файлах. Большие производственные программные продукты могут иметь сотни и тысячи файлов исходного кода. Компиляция и сборка таких проектов с нуля также может занимать много времени, поэтому при их

модификации обычно компилируются только измененные файлы, а для остальных частей используются ранее скомпилированные модули. Таким образом, работа с несколькими файлами является стандартной процедурой в разработке программного обеспечения, и при обучении программированию ее необходимо освоить наряду с другими навыками, даже если в рамках простых учебных задач она кажется лишней и ненужной.

Почему не рекомендуется объявлять массивы в теле отдельных функций? В современных вычислительных системах пользовательской программе предоставляется для работы две области памяти. Одна из этих областей часто называется кучей (heap), другая — системным стеком (stack). В куче размещаются глобальные переменные программы и области, запрашиваемые пользователем с помощью системных вызовов функций типа malloc или операторов `new`. В стеке размещаются локальные переменные, объявленные явно внутри тела функций. Размер стека, предоставляемого пользовательской программе, на порядки меньше размера кучи. При этом стек захватывается каждый раз при вызове очередной функции для размещения ее переменных и освобождается при завершении этой функции. Таким образом, если массив объявлен явно внутри функции, то он размещается в стеке, и если размер массива большой или перед этим в программе были вызваны и еще не завершили работу другие функции, то размера стека может элементарно не хватить для размещения массива. Особенно вероятен такой исход при использовании рекурсивных процедур (большом количестве вложенных незавершенных вызовов). Поэтому объявлять массивы явно внутри функций целесообразно только при их небольших размерах (с учетом возможных многократных вложенных вызовов функций), а для работы с объемными данными всегда запрашивать выделение памяти из кучи с помощью вызовов функции `malloc` или операторов `new`.

Как создать динамический массив? Язык C не поддерживает работу с динамическими массивами (т. е. с массивами, размер которых при необходимости автоматически изменяется во время исполнения программы). Однако в языке C имеются стандартные функции `malloc`, `realloc` и `free` для динамического выделения, перевыделения и освобождения памяти. Отметим, что на их основе эффективно реализовать полноценный

динамический массив не получится, так как многократные вызовы данных функций могут привести к неконтролируемому замедлению скорости работы программы.

Начиная со стандарта C99 в языке также появилась возможность создавать так называемые массивы переменной длины:

```
int n;
scanf("%d", &n);
double mas[n];
```

Однако память для хранения таких массивов выделяется в системном стеке. Как следствие, время жизни и область видимости массива ограничиваются рамками текущей функции (текущего блока операторов), а попытка создать массив большого размера может привести к разрушению стека.

В языке C++ имеется шаблонный класс `std::vector` с функциональностью динамического массива.

Почему не рекомендуется использовать глобальные переменные? Глобальная переменная потому так и называется, что она доступна из всех функций, находящихся в ее области видимости. Это накладывает определенные ограничения на использование имен, совпадающих с именем этой переменной, что может оказаться неудобно для реализации алгоритмов внутри этих функций. Другая потенциальная неприятность — изменение значения такой переменной в результате ошибки в коде функций или неправильной последовательности вызова функций, работающих с этой переменной. Подобные ошибки трудно искать, так как они не локализованы внутри одной функции, а могут проявляться именно при неправильном взаимодействии нескольких функций. Поэтому использование глобальных переменных считается потенциально опасным и рекомендуется его избегать.

Почему не рекомендуется использовать явные константы при написании кода? А как надо? При первоначальном написании кода часто возникает желание побыстрее получить хоть какой-нибудь результат, и некоторые начинающие программисты тогда пишут в циклах, выражениях, параметрах и других позициях явные константы вместо переменных или констант, которые там должны были бы стоять по сути алгоритма. Например, в качестве значения числа π они пишут `3.14`, в качестве числа повторения цикла — явно число 10 и т. д. Программа заработала и выдала результат. После этого начинается доработка программы

до более строгого состояния, под более строгие требования. И вот в этот момент вполне можно и не вспомнить, какие упрощения по записи констант были сделаны на первоначальном этапе. В результате остатки этого первого шага могут сохраниться в программе и в итоге сделать ее неправильной. Кроме этого, поиск констант, которые надо заменить на другие, правильные, может оказаться очень трудным. Например, то же число 10 может в разных местах относиться к разным вещам. Придется приложить дополнительные усилия, чтобы вспомнить, какой смысл вкладывался в это число в данном конкретном месте на этапе первоначальной отладки. И не факт, что эти воспоминания окажутся правильными. Поэтому с самого первого шага реализации программы надо четко определить, за что отвечают разные константы и переменные в алгоритме. Далее надо объявить эти константы с тестовыми значениями как переменные с модификатором `const` либо с помощью конструкции `define`. Далее всюду в программе нужно использовать выбранные имена. Тогда при любой последующей модификации программы достаточно будет изменить значения констант в соответствующих определениях, и вся программа не испортится от ошибочного смешения разных констант.

Почему считается, что итеративные алгоритмы предпочтительней рекурсивных? Прежде всего ознакомьтесь с ответом на вопрос «Почему не рекомендуется объявлять массивы в теле отдельных функций?» Рекурсивный алгоритм в своей реализации сохраняет в памяти (в стеке) промежуточные состояния локальных переменных рекурсивной функции. При большой глубине рекурсии это может привести к переполнению стека и аварийному завершения программы. Кроме того, каждый рекурсивный вызов требует определенных затрат времени на свою организацию, т. е. на размещение параметров вызываемой функции в стеке и на возврат вычисленного значения. Таким образом, при прочих равных, рекурсивный вариант алгоритма может выполняться дольше итеративного и есть риск переполнения стека. Использование рекурсивных алгоритмов оправдано, если код с рекурсией заметно проще итеративного варианта, не используется массивная передача параметров между вызовами и можно заранее оценить глубину рекурсии, чтобы снизить риск аварийного завершения.

Зачем разбивать программу на отдельные функции, ведь она и в таком виде, как у меня, правильно работает? Разбиение большой и сложной задачи на отдельные более простые задачи — это основной принцип не только программирования, но и вообще организации любого производственного процесса. И не только производственного, но и исследовательского, творческого и т. д. Он позволяет выстроить приоритеты и последовательность действий, распределять работу между многими исполнителями, повторно использовать результаты, полученные ранее. Конечно, если вы стремитесь минимизировать «вред» от занятий и программа пишется по принципу «сделал, сдал и забыл», то ее можно писать как угодно (хотя для сложной задачи такой код все равно будет практически невозможно отладить). Если же речь идет об обучении программированию и ваша цель — максимизировать пользу от занятий, то гораздо проще сразу принять к использованию те технологии, которые подтверждают свою эффективность и полезность на всем протяжении развития этой дисциплины. Разбиение программы на отдельные функции — это такое же требование, как разбиение книги на отдельные главы, параграфы, абзацы. Его надо принять как данность и научиться выполнять. Если не освоить этот навык сейчас, все равно придется научиться потом, но уже за счет времени, которое можно было бы потратить на другие, более интересные задачи.

Самое плохое ощущение для программиста — когда вокруг тебя стоят десять человек и все пытаются найти причину проблемы в твоей программе, а ты уже понял, в чем проблема, но боишься сказать, потому что это что-то вопиюще глупое...

5.2.2. Отладка и поиск ошибок

Компилятор выдал 100 500 строк с сообщениями об ошибках. Что делать? Важно понять, о чем говорится в сообщениях об ошибках. Если эти сообщения выдаются на английском языке, то надо найти переводы встречающихся там слов. Обычно в сообщениях указывается номер строки и номер позиции в строке, где, по мнению компилятора, расположено неверное выражение. Это даст возможность определить проблемное место в тексте программы. Отметим, что одна ошибка в тексте может повлечь другие, вторичные ошибки, т. е. если компилятор не

понял какое-то выражение, то с большой вероятностью он не сможет правильно интерпретировать и последующие выражения. Поэтому надо в первую очередь сфокусироваться на самой первой ошибке в списке диагностики. После исправления этой первой ошибки нужно запустить компиляцию заново. Вполне возможно, что теперь список ошибок будет заметно короче.

Бывают ситуации, когда сообщение об ошибке указывает на вроде бы вполне нормальную строку программы, а содержание этого сообщения как-то мало соотносится с содержанием этой строки. Такое возможно, если ошибка была сделана где-то раньше, но в том месте компилятор не увидел ничего незаконного, и эффект от ошибки проявился несколькими строками, или даже десятками строк, позже. Типичной ошибкой такого рода является пропуск точки с запятой в конце оператора, например при описании прототипа функции, а также пропуск открывающей или закрывающей скобки в выражении или блоке. В такой ситуации следует проверить баланс скобок в предшествующих фрагментах кода, а этому может помочь аккуратное форматирование текста (см. вопросы про стиль написания кода). Если разобраться, в чем заключается ошибка, не получается, то попробуйте закомментировать часть предшествующего ошибке программного кода. В некоторых случаях локализовать ошибку удается, поделив ругаемый оператор на несколько.

Почему надо обращать внимание на предупреждения компилятора, они ведь не являются ошибками? Предупреждение выдается, если компилятор встречает формально правильную конструкцию, которая тем не менее может оказаться результатом ошибки программиста. Типичный пример подобной ситуации, например, выражение `if (a = b)`, которое формально не запрещено, но в действительности появилось в результате опечатки в выражении `if (a == b)`, имеющем совершенно другой смысл. Какие конструкции компилятор относит к категории предупреждений, а какие к ошибкам, зависит от настроек, но в любом случае предупреждения нельзя пропускать. Их все следует проанализировать и проверить, не вызваны ли они действительно опечатками.

Компилятор может выдать также предупреждения о ненадежности некоторых конструкций кода. Это означает, что при выполнении программы соответствующие фрагменты могут при-

вести совсем не к тому результату, которого ожидал программист. Такие конструкции также лучше переписать в другой форме, чтобы подобных предупреждений не появлялось. Это сделает работу программы более предсказуемой и, следовательно, более надежной.

Компилятор ругается на неиспользуемую переменную или неиспользуемый параметр функции. Как быть? Если переменная лишняя, то однозначно нужно убрать ее из кода. Если же переменная не используется в данном варианте расчета, то можно воспользоваться следующими приемами:

```
double f(double x, double y){
  .....
  return y + 0.0 * x;
}
```

или

```
double f(double x, double y){
  .....
  (void) x; // или в C++: static_cast <void> (x);
  return y;
}
```

В данном примере компилятор ругался на неиспользуемый параметр `x` функции `f`.

Почему компилятор ругается на описание переменных после исполняемых операторов, т. е. запрещает смешение деклараций и кода? В стандартах ANSI C и C90 локальные переменные можно описывать только в начале блока операторов — области кода, ограниченной фигурными скобками. В стандарте C99 данное ограничение формально снято. Однако пользоваться предоставленной возможностью следует обоснованно, так как хаотичное описание переменных затрудняет анализ кода.

Как убедиться, что программа работает правильно? Если программа скомпилировалась без ошибок, это еще не значит, что она работает правильно. Для проверки правильности работы программу надо протестировать, т. е. запустить на выполнение с различными наборами входных данных и убедиться, что в каждом случае получается правильный или ожидаемый результат. Если на каком-то наборе данных программа перестает работать

или выдает неверный результат, то необходимо найти ошибки и устранить их. Более подробно о тестировании можно прочитать в ответе на вопрос «Что означает «написать тесты» для моей программы?» Заметим, что тестирование программы является необходимым элементом разработки и без достаточного тестирования программа не может считаться готовой к использованию.

Главным признаком правильной программы является ее нормальное завершение при любых (допустимых) входных данных. Как иногда говорят, программа может отказаться решать вашу задачу, если данные некорректны, но она не имеет права «падать» ни при каких условиях.

Программа выдает неверный результат. Что делать? Очень часто данный вопрос формулируется следующим образом: «Моя абсолютно правильная программа выдает ошибку. Кто виноват?» Предлагаются разные варианты ответа: это все баги компилятора, системной библиотеки, ОС, железа... Самое главное в этой ситуации — признать, что ваш программный код содержит ошибку. Возможно, входные данные считываются неправильно либо имеют недопустимые значения. Возможно, какие-то конструкции языка используются некорректно либо ошибочна логика алгоритма. Одним словом, нужно искать ошибку.

Программа работает, но выдает что-то странное. Как искать ошибку? Программисты шутят: «Если программа работает, то она определенно решает некоторую задачу. Остается только выяснить какую». Если есть сомнение в правильности полученного результата, то нужно постараться локализовать место в коде, где допущена ошибка. Скорее всего, неправильный результат вызван опечаткой, которая не повлияла на синтаксическую правильность кода, но изменила суть исполняемых операторов. Искать подобную ошибку, пристально просматривая код, дело абсолютно безнадежное для начинающего программиста (да и для опытного тоже). Психологически автор программы настроен на то, что его код правильный, и поэтому он с большой вероятностью просто не заметит опечатку, скользнув по ней взглядом. Единственным надежным методом является трассировка программы с анализом промежуточных значений. Это можно делать с помощью специальных программ-отладчиков (см., например, раздел 5.1.4 про отладчик gdb) или по-простому с использованием печати промежуточных значений. Этот простой

способ мы здесь вкратце и опишем. Главное — мы должны представлять себе, какие значения промежуточных переменных являются правильными, а какие нет. Нам придется выполнить алгоритм «вручную», чтобы можно было сравнить наши результаты с теми, которые покажет программа. На практике это не всегда необходимо, но в любом случае мы должны понимать, похожи ли вычисленные программой значения на правду и в какой момент они из правильных превратились в неправильные.

Итак, каждая программа имеет исходные данные. Поэтому первое, что надо сделать, — убедиться в правильности считывания/инициализации данных. Проверьте, что при считывании передается именно адрес переменной (в простейшем случае перед именем есть знак `&`); распечатайте введенные данные сразу после их ввода/инициализации — часто ошибка обнаруживается уже на этом этапе.

Далее данные преобразовываются по некоторым формулам. Распечатайте переменные, участвующие в формуле, перед операторами, реализующими эту формулу, и после этих операторов вместе с полученным результатом. Это даст возможность посмотреть, какие значения поступают на вход вычислений и что получается после вычислений.

Если вычисления выполняются с помощью вызовов функций, то распечатайте фактические параметры непосредственно перед вызовом и потом сразу внутри этой функции. То же самое и для результата: распечатайте возвращаемые значения непосредственно перед оператором `return` и сразу после возвращения из функции в месте ее вызова. Это поможет отлавливать ошибки, связанные с неправильной передачей параметров и возвращаемых значений.

Отдельный класс ошибок — опечатки в логических выражениях. Они приводят к тому, что отдельные части кода вступают (или не вступают) в работу не совсем так, как предполагал программист. Поэтому распечатайте значения переменных из логических выражений операторов `if` и из блоков этой условной конструкции; напечатайте параметры циклов и промежуточные значения из тела цикла. Это позволит увидеть, истинным или ложным оказалось условие в процессе вычислений, сколько раз и с какими параметрами выполнился цикл (и выполнился ли он вообще). Все это даст пищу для размышлений и поможет локализовать возможную ошибку. Никогда не доверяйте своему

ощущению, что «ну этот-то фрагмент у меня точно работает правильно». Добавьте операторы промежуточной печати и проверьте, куда заносит вашу программу. Это сэкономит в итоге много времени и сил.

Важное замечание: при печати обязательно заканчивайте строку вывода символом перевода строки `'\n'`. В противном случае вычислительная система не гарантирует своевременного вывода вашей строки (см. ответ на вопрос «Что такое буферизация ввода-вывода и когда я должен ее учитывать?»).

Резюмируем: логика трассировки состоит в прогоне программы с некоторым набором данных и прослеживании результатов промежуточных вычислений, с тем чтобы понять, в каком месте возникает ошибка. Несмотря на свою примитивность, этот метод является весьма эффективным при поиске вычислительных и логических ошибок в программе. Но, конечно, он не заменяет полноценную отладку в дебагере.

Программа «упала». Как искать ошибку? На программистском жаргоне слово «упала» означает аварийное завершение работы программы в ответ на некоторое событие, запрещенное в контексте запуска данной программы на выполнение. Такие события — исключение в операции с плавающей точкой, нарушение сегментации, разрушение или переполнение стека, разрушение структур управления памятью. При поиске ошибки первым делом надо локализовать место ее возникновения. Это можно сделать аналогично трассировке (см. ответ на вопрос «Программа выдает неверный результат. Как искать ошибку?»), причем зачастую достаточно просто распечатывать некоторые словесные метки, чтобы увидеть: вот до этого места программа дошла, а вот эти метки не распечатались, так как программу уже упала. Опять напоминаем, что при печати нужно обязательно заканчивать строку вывода символом новой строки `'\n'`. В противном случае вычислительная система не гарантирует своевременного вывода информации (см. ответ на вопрос «Что такое буферизация ввода-вывода и когда я должен ее учитывать?»).

Обычно существенно быстрее место подобной ошибки удается найти с помощью отладчика gdb (см. раздел 5.1.4). После того как место ошибки локализовано, можно провести трассировку значений, используемых в обнаруженных операторах, и понять причины возникновения ошибки. Наиболее вероятные причины

возникновения каждого из отказов описаны в ответах на соответствующие вопросы. Случается так, что добавление оператора `printf`, запуск под `gdb`, включение строчки комментариев в текст ликвидирует падение (но не ошибку в программе!) и приводит к корректному завершению. В этом случае придется вернуться к исходному коду и попробовать другие варианты.

Программа завершается с исключением в операции с плавающей точкой. Что делать? Исключение в операции с плавающей точкой означает, что была попытка выполнить некорректное действие с вещественными числами. Примерами таких действий являются деление на нуль, вычисление корня или логарифма отрицательного числа и т. п. Другая причина — использование в арифметической операции значений, которые не входят в диапазон представимых чисел. Например, «нечисла» (NaN — not a number) или денормализованного числа. Характерным признаком таких чисел является непомерно большой или непомерно маленький десятичный порядок, который появляется при попытке их распечатать. Детали о представлении вещественных чисел можно узнать из стандарта IEEE 754. Подобные числа могут получиться, например, если переменная была объявлена, но не инициализирована, или при доступе по указателю к области памяти, в которую числа ранее не записывались (например, там находится код операций самой программы, или текстовые строки, или вообще свободный участок памяти.

Таким образом, данное исключение обычно вызвано либо появлением нуля или отрицательного числа, либо использованием неинициализированной переменной, либо ошибочным значением указателя, по которому потом делается попытка взять вещественное значение. В любом случае после локализации места возникновения ошибки нужно распечатать подозрительные переменные и проанализировать их значения. Наиболее быстро и удобно это делается в отладчике gdb.

Программа завершается с ошибкой сегментации. Что делать? Программа выдала все результаты правильно, а потом завершилась с ошибкой сегментации. Что делать? Ошибка сегментации (segmentation fault) возникает, когда программа пытается обратиться к памяти, которая ей не выделена системой. Наиболее частые причины для такой ошибки — это неверный индекс при обращении к элементу массива, что

приводит к выходу за границы этого массива, либо просто обращение по неверно установленному указателю, в частности по нулевому указателю. После локализации места возникновения ошибки нужно распечатать используемые индексы массивов или указатели. Обычно сразу видно, что индекс принимает неправильные значения или имеется нулевой указатель. Далее надо трассировкой определить, где и почему возникли такие значения. Если указатель ненулевой, но есть подозрение, что он неправильный, то имеет смысл распечатать адреса других переменных в окрестности этого фрагмента кода. В ряде случаев можно обнаружить, что эти адреса существенно различаются, что может косвенно свидетельствовать, что изучаемый адрес неправильный. Наиболее быстро и удобно это делать в отладчике gdb.

Другой причиной возникновения ошибки сегментации является разрушение структур, описывающих распределение памяти вашей программы. В этом случае исключение происходит уже в системных процедурах, выгружающих вашу задачу из памяти при ее завершении. В такой ситуации ваша программа как будто заканчивается нормально, выдает все требуемые ответы, но потом возникает нарушение сегментации в строке, соответствующей последней закрывающей фигурной скобке функции `main`.

Проблема может быть вызвана тем, что при захвате памяти с помощью функции `malloc` некоторый участок памяти, непосредственно примыкающий к выделенному вам фрагменту, используется для сохранения параметров выделенного блока (например, его длины). Если в результате ошибки с адресами ваша программа что-то запишет в этот участок, то информация, сохраненная там, будет потеряна. Но обнаружится это только тогда, когда системные процедуры начнут выгружать вашу программу из памяти и пользоваться этой информацией для освобождения памяти. Точно такими же причинами может быть вызвана ошибка сегментации при вызове функции `free`.

В таком случае искать источник ошибки довольно трудно, поскольку он никак не связан с местом ее возникновения, информация о памяти может быть затерта где угодно. Обычно для поиска используют временное отключение фрагментов кода. Можно, например, закомментировать вызовы разных ваших функций и попробовать выполнить программу. Результат, конечно, будет неверным, но ошибка сегментации может пропасть. Тогда можно предположить, что память разрушается

внутри закомментированной функции. Далее комментируем части кода внутри этой функции и смотрим, когда ошибка сегментации сохраняется, а когда исчезает. Так можно локализовать подозрительное место и потом уже более детально трассировать значения переменных в этой области.

Другой подход — использовать возможности gdb для контроля содержимого ячеек на запись.

Программа «зависла». Как искать ошибку? На программистском жаргоне слово «зависла» означает, что программа не выдает никаких результатов, не реагирует на нажатия клавиш и не завершается. Вывести ее из этого состояния иногда можно, нажимая в консоли клавиши `Ctrl+c` («срубить задачу») или `Ctrl+z` (приостановить выполнение задачи, переведя в фоновый режим; для продолжения выполнения нужно набрать `fg`). Если при нажатии клавиши `Enter` программа все же подает какие-то признаки жизни, то проверьте, не ожидает ли она от вас какого-либо ввода. Просмотрите еще раз все операции ввода с клавиатуры в вашей программе и поставьте непосредственно перед ними вывод текстовых приглашений к вводу необходимых значений.

Другая частая причина зависания — вход в бесконечный цикл. Обнаружить такое помогает простая трассировка. Напечатайте из программы текстовые метки позиций, до которых доходит процесс вычисления, и также добавьте печать значений в телах циклов. Вы увидите, до какой метки доходит выполнение вашей программы, а бесконечная серия печати из цикла покажет место ошибки. Далее можно трассировать условие завершения цикла, с тем чтобы определить, по каким причинам оно не выполняется.

Иногда бывает трудно понять проблему с условием цикла, так как строки вывода из цикла мгновенно убегают за край экрана. В этом случае может помочь следующий простой прием, ограничивающий количество проходов цикла путем введения дополнительного счетчика. Например, мы обнаружили бесконечный цикл, который должен был закончиться за 10—20 шагов, но этого не происходит. При попытке распечатать значения по типу

```
for (i = 0; F(i) < G(i); i++) {
  printf("%d %d %d\n", i, F(i), G(i));
  .....
}
```

первые строки убежали за экран, и мы не знаем, какие значения там появлялись, и не можем анализировать работу функций **F** и G. В таком случае добавим в цикл дополнительный оператор:

```
for (i = 0; F(i) < G(i); i++) {
  printf("%d %d %d\n", i, F(i), G(i));
  if (i == 30) exit(1); /* DEBUG code */
  .....
}
```

Теперь наш цикл гарантированно выполнится не более 30 раз и мы сможем увидеть, как меняется значение **i** и что происходит с условием окончания цикла.

При работе в консоли Linux также можно набрать **a.exe|more**, т. е. передать выдаваемые на экран результаты утилите more, которая обеспечит постраничный вывод информации (при просмотре используйте клавиши **Enter** — переход на следующую строку, пробел — переход на следующую страницу, **q** — выход из more, также приводящий к завершению основной программы). Более опасным является вариант **a.exe > tmp.dat**, т. е. перенаправление консольного вывода в файл. В результате можно получить неконтролируемо растущий файл **tmp.dat** и превышение дисковой квоты.

— Ты исправил ошибки в программе?
— В разумных пределах...

5.2.3. Может ли неправильная программа выдавать правильный ответ? Почему тогда она неправильная?

Будем считать, что программа является правильной, если она для произвольных допустимых входных данных за разумное время выдает верный ответ и содержит только строго регламентированные конструкции языка, компиляция которых не зависит от версии и ключей используемого компилятора. Например, использование в расчетах неинициализированных переменных, локальный выход за границу массива в отдельных случаях может и не повлиять на правильность окончательного результата. Следующий менее прозрачный пример разберите самостоятельно.

```
#include<stdio.h>
double M(double sum, int N);
```

```
double M(double sum, int N){
/* M  = (x1 + ... + xN) / N */
  double xi;
  if(scanf("%lf", &xi) != 1){
    return sum / (double)N;
  }
  else{
    M(sum + xi, N + 1);
  }
}
int main(void){
  printf("M = %20.15lf\n", M(0.0, 0));
 return 0;
}
```

В данном случае отсутствие оператора `return` в ветви `else` при gcc-компиляции без дополнительных ключей обычно не портит ответ.

Был глюк, научились повторять — стал баг, задокументировали — теперь фича. Легче изменить спецификацию, чтобы она соответствовала программе, нежели наоборот.

5.2.4. Переносимость

Почему моя программа на одном компьютере (дома) работала, а на другом (в классе) не работает? Моя программа дома компилируется, а здесь, в классе, нет. Что делать? Переносимость программного обеспечения — одна из самых серьезных проблем в программировании. Корень проблемы в том, что существует много различных операционных систем, различной аппаратуры, различных версий вспомогательного программного обеспечения (например, компиляторов).

При возникновении проблем с переносимостью нужно первым делом выяснить, в чем они состоят и чем вызваны. Разберем несколько типичных примеров.

1. Программа использует специфичные встроенные функции или библиотеки конкретной системы программирования, которые присутствуют в одной системе и отсутствуют в другой. Тут надо просто посмотреть, какая функциональность обеспечивается

этими функциями в вашей программе. Возможно, эту функциональность можно реализовать самому; может быть, в другой системе она поддерживается другими библиотеками; может быть, она вообще вам не нужна и попала в вашу программу по инерции при использовании готовых фрагментов чужого кода. Конкретный пример — использование заголовочного файла `conio.h`, присутствующего в Windows, но отсутствующего в Linux. Включение этого файла либо вообще не оправдано, либо может быть заменено использованием аналогичных функций стандартного ввода-вывода.

2. Программа неявно опирается на специфику конкретной системы программирования или конкретных ее настроек. Заполнение свободной памяти зависит от настроек. Иногда свободная память заполняется нулями, иногда значениями NaN, иногда вообще не обрабатывается (остается то, что было раньше). Если ваша программа ориентируется на какое-то конкретное заполнение памяти, то она, скорее всего, не будет переносима, так как в других условиях неинициализированные переменные могут получить другое начальное значение со всеми вытекающими последствиями.

К подобным настройкам относится также реакция системы на различные исключительные ситуации, например на деление на нуль. В одних случаях деление разрешается с результатом Inf (бесконечность) и это значение дальше используется в вычислениях. При других настройках система может вызывать исключение в операции с плавающей точкой и завершать вашу программу. То же самое относится к проверке попытки доступа в чужую память (segmentation fault). Система может остановить программу, а может разрешить ей дальнейшую работу (с непредсказуемыми последствиями). Также могут отличаться и сами способы выделения памяти и контроля за нарушениями. Таким образом, одни и те же дефекты могут быть признаны фатальными или вполне допустимыми.

3. Имеются различия в версиях и настройках компилятора. При описании конструкций языка C иногда встречается фраза «результат не определен». Это означает, что стандарт не задает жестких правил трактовки данной конструкции и конкретная реализация остается на совести разработчика компилятора. Обычно это относится к «двусмысленным» выражениям типа `i = i+++++i`, когда операции инкремента могут быть

выполнены в разной последовательности. Есть и другие примеры. В частности, включение различных режимов оптимизации кода при компиляции может вызывать разные побочные эффекты при выполнении. Мораль здесь простая: чем проще и однозначней записан код программы, тем меньше будет потенциальных проблем с переносимостью.

4. Имеются различия в кодировках. Хорошо известно отличие в обозначении конца строки в системах Unix и Windows. В первой конец строки обозначается одним байтом `'\n'` (код 0x10), во второй — двумя байтами `'\r''\n'` (коды 0x13, 0x10). Если программа при чтении или формировании текстового файла ориентируется только на одно из этих представлений, то при переходе на другую систему возможна неправильная работа. Другая аналогичная особенность связана с кодировками букв. Хорошо известна кодировка ASCII для букв латинского алфавита, цифр, знаков препинания и некоторых специальных символов (спецификация ASCII фиксирует только символы с кодами от 0 до 127). Для букв других алфавитов, например русского, применяются иные кодировки с разными принципами построения. Например, кодировка CP1251 (Windows-1251) сохраняет алфавитный порядок символов русского языка, а кодировка KOI8-R использует алфавитный порядок, созвучный с латинским алфавитом (а, б, ц, д, е, ф,...). Эти кодировки отводят один байт на символ, а кодировка UTF-8 использует два байта для кодировки русских букв и один байт для кодировки латинских. Перечисление можно продолжить, но ясно, что различие в кодировках способно вызвать проблемы с переносимостью программ, обрабатывающих текстовые файлы.

Есть и другие, более тонкие проблемы с переносимостью, например порядок байтов при хранении целых чисел, но здесь мы обсуждать это не будем.

Из апелляции: «Считаю, что в извечном споре, какой должен быть индекс у первого элемента массива, 0 или 1, мой компромиссный ответ 0.5 был отвергнут без надлежащего изучения».

5.2.5. Полезные советы

Как измерить время работы всей программы или ее отдельного фрагмента? Один из самых простых способов — использовать функцию `clock`, возвращающую количество

тактов, отработанных процессором с начала выполнения программы. Не вдаваясь в подробности, проиллюстрируем, как замерить время работы некоторого фрагмента программы.

```
#include <time.h>
.....
clock_t t1, t2;
double seconds;
t1 = clock();
/* здесь фрагмент для замера времени */
t2 = clock();
seconds = ((double)(t2 - t1))/CLOCKS_PER_SEC;
```

Что означает «написать тесты» для моей программы? Тест программы означает ее запуск на наборе специально подготовленных данных, для которых мы можем с уверенностью отличить правильный ответ от неправильного. Например, если решается уравнение, то для теста мы можем подобрать уравнение с известными значениями корней и по выполнении программы сравнить эти значения с полученным результатом. Одного теста для программы, как правило, недостаточно. Правильный набор тестов должен обеспечивать проверку всех возможных логических ветвей выполнения алгоритма. Таким образом, если в нашей программе проверяется какое-то условие, то должен быть тест, при котором условие выполняется (и программа идет по одному пути), и тест, при котором условие не выполняется (программа идет по другому пути). На разных тестах должны проверяться разные классы входных данных. Обычно в тестах проверяется следующее:

- ввод исходных данных (случаи отсутствия данных, испорченных данных, явно некорректных данных и т. п.);
- правильность вычислений (хорошие данные приводят к правильному ответу);
- разумность реакции на несовместимые с условием задачи входные данные (например, на необходимость распечатать максимальный элемент последовательности, которая оказалась пустой);
- масштабирование данных (что случится, если исходные значения будут очень большими, маленькими, разных порядков);

— масштабирование по объему данных (большие наборы входных данных);
— устойчивость и погрешность вычислений (достигается ли заявленная точность результата);
— скорость работы (соответствует ли работа программы заявленной теоретической трудоемкости алгоритма);
— другие особые ситуации по смыслу задачи.

При реализации программы надо сразу предусматривать тестирование. Например, полезно реализовывать удобный ввод данных из разных источников (файлов), генерацию данных с разными характеристиками (массивов разной длины, диапазона значений и пр.). Стоит добавить печать диагностических сообщений, чтобы видеть, по какому пути пошла работа алгоритма. Само тестирование следует проводить после каждого внесения изменений.

Составление хороших тестов — непростая задача, зачастую даже более объемная, чем сама тестируемая задача. При подборе тестов помните, как определил основную цель процесса тестирования Э. Дейкстра: «Тестирование призвано указывать на наличие ошибок, а не подтверждать правильность работы программы».

Почему не следует сравнивать вещественные числа на равенство или неравенство? Некоторые компиляторы квалифицируют операции сравнения вещественных чисел на равенство как ошибки или выдают предупреждения. Это связано с тем, что результат операции с вещественными числами содержит погрешность, обусловленную формой представления вещественных чисел и вынужденным округлением: бесконечное множество действительных чисел заменяется на некоторое конечно подмножество, хранимое в ЭВМ. В этом смысле можно говорить, что операции с вещественными числами почти всегда выполняются с некоторой погрешностью, причем погрешность конкретного результата зависит не только от используемых арифметических действий, но и от порядка проводимых вычислений. Поэтому при одних и тех же входных данных даже математически эквивалентные формулы могут на выходе дать разный результат в пределах накопленной погрешности вычислений. Вообще говоря, имея математически эквивалентные формулы, мы могли бы ожидать, что результат вычисления по ним получится одинаковым, и поставить

сравнение на равенство в условии для выбора нужной ветви алгоритма. Однако в действительности результат получится разным, и алгоритм пойдет по неверному пути. Таким образом, при обработке вещественных чисел этот факт надо иметь в виду, сравнение чисел на равенство проводить в рамках некоторой погрешности `eps` и вместо `if (a == b)` писать `if (fabs(a-b) < eps)`.

Как правильно вводить данные в программу? Как узнать в программе, правильно выполнилось чтение или нет? Как определить закончился ли файл, т. е. есть ли в нем еще значения для чтения? Ввод данных — это одна из принципиальных точек программы, так как от правильности введенных значений зависит возможность дальнейшего выполнения программы. Вводить данные можно разными способами. Здесь мы остановимся на проблеме контроля за правильностью ввода на примере функций типа `scanf`. В контроле ввода есть два основных вопроса:

1) состоялся ли ввод вообще;
2) правильно ли ввелось значение.

На первый вопрос отвечает значение, возвращаемое функцией `scanf`: оно равно количеству объектов, правильно считанных в соответствии с указанным форматом. К примеру, если мы при вызове функции `scanf` заказывали ввод трех значений, то функция должна вернуть 3, иначе при вводе что-то пошло не так:

```
int a, b;
double x;
if (scanf("%d%d%lf", &a, &b, &x) != 3){
/* ошибка ввода */
}
/* контрольная печать: */
printf ("a = %d b = %d x = %f\n", a, b, x);
```

Со вторым вопросом ситуация более запутанная. Параметрами функции ввода являются адреса мест в памяти, куда надо разместить прочитанные значения. Функция `scanf` не знает, какие в действительности типы данных стоят за этими адресами, она определяет эти типы из анализа спецификаций преобразования, записанных в ее форматной строке. В данном случае это `%d%d%lf`, т. е. `int, int, double`. Функция преобразует прочитанные значения в представления этих типов и записывает их по адресам, указанным в параметрах. Если спецификации преобразования соответствуют типам переменных, то все хорошо.

Если нет, то может оказаться так, что 8 байт `double` будут записаны по адресу, где размещается `int`, и соседние с `int` 4 байта будут испорчены. Аналогично неудачно, если вместо 8-байтового представления `double` по указанному адресу будет записано 4 байта от `int`. При определенных настройках компиляторы проверяют соответствие спецификаций и типов и выдают ошибку при их несовпадении. Иначе нам остается надеяться только на себя и тщательно проверять введенные значения при отладке, например с помощью контрольного вывода сразу после чтения.

При вводе данных из файла рекомендуется руководствоваться теми же правилами. Однако очень часто применяется метод контроля по сообщению от функции `feof`. В ряде случаев это может привести к неверной работе и даже зацикливанию программы. Пусть, например, требуется прочитать последнее целое число из файла. Какой фрагмент следующего кода следует оставить?

```
int a;
FILE *f = fopen("num.txt", "r");
//1: while (!feof(f)) fscanf(f, "%d", &a);
//2: while (fscanf(f, "%d", &a) == 1);
```

Если в файле записаны только числа, то оба фрагмента дадут один и тот же правильный результат. Однако рассмотрим случай, когда в файле в какой-то момент вместо чисел будет записано что-то нечисловое, например строка букв. Тогда первый фрагмент зациклится, а второй прочитает последнее число перед той самой строкой букв. Действительно, как только будет достигнута строка букв, операция чтения не состоится, так как функция не сможет преобразовать буквы в числовое значение. Текущая позиция в файле останется без изменения, ее сдвига к концу файла не произойдет. Далее цикл будет выполняться бесконечно, так как конец файла никогда не будет достигнут. Во втором фрагменте точно так же чтение будет продолжаться до достижения строки букв, но в этот момент цикл завершится, так как функция чтения не вернет требуемую единицу, и мы останемся с последним прочитанным числом. Следует ли этот вариант считать правильным — вопрос философский, но он, по крайней мере, не зацикливается.

Ну а теперь запишем, как прочитать последнее число из файла, если оно действительно может оказаться после такой

строки символов. Попробуйте самостоятельно разобраться, как работает соответствующий фрагмент.

```
int a, c;
FILE *f = fopen("num.txt", "r");
while (!feof(f)) {
  if (fscanf(f, "%d", &a) != 1) {
  c = fgetc(f);
  }
}
```

Как эффективно записывать и считывать большие массивы данных? При работе с большими объемами данных рекомендуется использовать низкоуровневые функции типа `fwrite`, `fread`. Например, так можно обрабатывать множество пар «строка, число», хранящихся в виде массива структур.

```
struct B {
  char name[17];
  int id;
}
const int N = 1000000;

/* Запись в файл: */
FILE *fout = fopen("base.data", "wb");
struct B *dbi = (struct B *) malloc(N *
                    sizeof(struct B));
{ /* заполнение данных в массив структур */}
fwrite((void *)dbi, sizeof(struct B), N, fout);

/* Последующее считывание из файла: */
FILE *fin = fopen("base.data", "rb");
struct B *dbo = (struct B *) malloc(N *
                    sizeof(struct B));
fread((void *)dbo, sizeof(struct B), N, fin);
```

Почему требуют писать `int main(void)`, хотя чаще можно встретить `int main()`? В соответствии со стандартом языка C++ формально данные записи эквивалентны, обе они означают, что функция не имеет параметров. В рамках языка C для `int main(void)` действует аналогичное соглашение, а вот

пустые скобки `int main()` означают, что компилятор не должен контролировать процесс согласования типов входных и принимаемых данных. А это может быть запрещено соответствующими ключами компиляции.

Отметим, что функция `main` является перегружаемой и для нее также поддерживается вариант с параметрами, например

```
int main(int argc, char **argv){
/*  операторы */
}
```

Конкретная реализация может варьироваться в зависимости от системы, например, в macOS параметров может быть четыре.

Почему при вводе чисел с двойной точностью надо писать `%lf`, а при выводе можно писать `%f`? Функции ввода-вывода стандартной библиотеки разбираются с типами переменных по спецификациям преобразования, указанным в форматной строке. В случае ввода эта информация является весьма существенной, так как позволяет определить число байтов и форму представления конкретного значения для записи по указанному адресу (см. также ответ на вопрос «Как правильно вводить данные в программу?»). При выводе действительных чисел функция печати получает значения соответствующих переменных и, в соответствии со стандартом ISO C11, сама приводит передаваемое ей вещественное число типа `float` или `double` всегда к числу типу `double`, которое потом и используется для вывода. Поэтому для вывода спецификация `%lf` неотличима от `%f` и также может применяться, если это не регулируется какими-то дополнительными настройками компилятора.

Что лучше использовать для ввода-вывода — stdio или iostream? Функции стандартной библиотеки ввода-вывода (заголовочный файл `stdio.h`) предоставляют пользователю возможность реализовать практически любой ввод и вывод данных при условии, что он сводится к работе с базовыми типами данных, а именно целыми и вещественными числами, символами и текстовыми строками. Отличительной особенностью этой библиотеки является явное управление форматом представления данных. Пользователь должен указать, как нужно преобразовывать друг в друга символьные и числовые представления значений переменных. Это делается с помощью являющейся аргументом функций ввода и вывода форматной строки и спецификаций преобразова-

ния. С одной стороны, это очень удобно, так как все форматные преобразования находятся на виду и легко правятся и подгоняются под нужную форму. С другой стороны, ошибки в выборе подходящих преобразований (несоответствие типов) приводят к странным результатам при выводе и ошибкам при вводе. Другой положительный момент в использовании этой библиотеки — это возможность непосредственного контроля ввода и вывода: функции возвращают значения, по которым можно сразу определить, насколько успешно выполнилась требуемая операция.

Спецификой языка C++ является возможность конструировать новые типы данных путем объединения разных объектов в более сложные структурные объекты. Для организации ввода-вывода таких объектов в рамках библиотеки языка C пользователю придется добираться до объектов базовых типов, чтобы вывести их значения с использованием спецификаций преобразования. В сложных структурных объектах это может оказаться очень непросто, а в ряде случаев и невозможно, так как точное распределение компонент сложного объекта по базовым типам может быть недоступно пользователю (например, если использовались наследование или шаблоны). Поэтому в C++ была предложена другая идеология, предполагающая, что сам объект отвечает за свой ввод и вывод. Это реализуется переопределением операторов << и >> для каждого конкретного типа. Для базовых типов данных это переопределение выполнено по умолчанию, что позволяет сразу использовать << и >> для простейшего ввода и вывода. Итак, при вводе-выводе с помощью операторов << и >> мы можем не волноваться о соответствии типов переменных спецификациям преобразования (нужные преобразования подставляет сам компилятор), и поэтому исключаются ошибки преобразований, возможные в языке C. С другой стороны, если мы хотим получить форму вывода, отличную от предоставляемой по умолчанию, то должны изменить соответствующие настройки потока ввода-вывода. Это делается вызовом отдельных функций (методов класса) для используемого потока, например методов `width`, `precision` и др. Контроль ошибок здесь также имеет свою специфику. При возникновении ошибки в потоке устанавливается соответствующий флаг и поток блокируется, что выливается просто в игнорирование всех последующих операций ввода-вывода с этим потоком. Таким образом, мы можем просто не заметить, что произошла

ошибка, и программа будет работать дальше с непредсказуемым результатом. Чтобы проверить наличие ошибок, нужно вызывать специальные методы (`is_open`, `good`, `bad`, `fail`, `eof`) и анализировать возвращаемые ими значения. Необходимость строго контролировать, скажем, ввод большого количества числовых данных может вылиться в достаточно громоздкий код.

На вопрос, поставленный в заголовке, каждый может дать свой ответ, исходя из описанных выше свойств и собственного опыта. Стандартная библиотека языка C проще по своей структуре и, следовательно, в изучении. Она позволяет все видеть, проверять и править своими руками. Потоковая библиотека языка C++ берет многие решения на себя и поэтому проще в использовании на начальном этапе. Но она более сложна по структуре и требует серьезного изучения, чтобы настраивать ее под требования разнообразных задач ввода-вывода.

Можно ли вернуться при чтении к началу файла, не закрывая и открывая его повторно? Да, и очень просто. Для этого есть специальная функция `rewind`. Другой способ состоит в использовании вызова `fseek(f, 0, SEEK_SET)`, где `f` — указатель на требуемый файл. Однако для устойчивой и предсказуемой работы данной функции рекомендуется открывать соответствующий файл как бинарный, `fopen(fname, "rb")`, и использовать функцию `fread()` (см. документацию по функциям стандартной библиотеки ввода-вывода).

Можно ли вернуться к предыдущему уже прочитанному числу при последовательном чтении файла? Формально мы можем переставлять текущую позицию в файле с помощью функции `fseek()` (см. документацию). Но эта функция требует указать смещение в байтах. Поэтому ответ на вопрос зависит от того, знаем ли мы, сколько байтов занимала в файле запись последнего прочитанного числа, в каком режиме открыт файл и какую функцию мы используем для чтения. Надежной является связка `ftell()`, `fseek()`, `fread()` для файлов, открытых как бинарные, т. е. для двоичных потоков.

Куда возвращается значение оператора `return` из функции `main`? Оператор `return` возвращает значение из функции `main` тому процессу, который функцию `main` вызвал. Не вдаваясь в технические подробности, можно считать, что при стандартном запуске исполняемого кода из командной строки за

это отвечает загрузчик исполняемых модулей. Интерпретаторы часто сохраняют код выхода последней команды в системной переменной с именем `$?`, а ее содержимое можно посмотреть, набрав в консоли `echo $?`.

Если мы сами пишем код для запуска других программ (например, с помощью функции `exec`), мы можем получать и анализировать это значение. Принято считать, что при штатном завершении программы функция `main` должна возвращать значение 0. Ненулевые значения соответствуют ненормальному завершению. Данное соглашение может действовать при автоматическом тестировании вашей программы, например на олимпиаде или зачете. И если некоторая разумная ветка `main` заканчивается, например, `return 1`, то соответствующий тест может получить статус «упавшая программа».

Результатом алгоритма является несколько чисел, а функция позволяет вернуть только одно значение. Как быть? Действительно, по правилам языков C и C++ функция может вернуть только одно значение, но это значение может быть структурного типа, т. е. представлять собой несколько полей разных типов. Таким способом можно вернуть (при большой необходимости) полноценный массив. Другим вариантом может быть использование параметра типа указателя или ссылки. Вот пример трех реализаций на C++ функции, возвращающей квадрат и куб своего аргумента, с использованием этих трех способов. (Реализаций на C будет только две из-за отсутствия ссылочного типа, и первая будет отличаться чуть более громоздким описанием при работе со структурами.)

```
struct SquareAndCube {
  double x2, x3;
};
SquareAndCube FunOne(double x){
  SquareAndCube sq;
  sq.x2 = x * x;
  sq.x3 = sq.x2 * x;
  return sq;
}
void FunTwo(double x, double * px2, double * px3){
  *px3 = *px2 = x * x;
  *px3 *= x;
```

```
  return;
}
void FunThree(double x, double & x2, double & x3){
  x3 = (x2 = x * x) * x;
  return;
}
```

Что такое буферизация ввода-вывода и когда я должен ее учитывать? Операции ввода-вывода требуют на несколько порядков больше времени, чем вычислительная работа процессора и обращение к оперативной памяти. Поэтому все современные вычислительные системы используют буферизацию ввода-вывода, состоящую в том, что данные при выводе перемещаются в буфер оперативной памяти и затем передаются на внешние устройства (диски, мониторы) независимыми аппаратными устройствами. Сам процессор в это время продолжает выполнение кода пользовательской программы, идущего после команды вызова, или обрабатывает другие задачи системы. При команде ввода пользовательская программа блокируется и запускается независимый процесс копирования данных с внешнего устройства в буфер оперативной памяти, а процессор опять занимается обработкой других задач в соответствии с их приоритетами. Когда буфер ввода данных заполняется, система возобновляет работу пользовательской программы, и та считывает данные из буфера ввода и продолжает свои вычисления. Важным итогом этой схемы является асинхронность работы программы и системы вывода, т. е. выполнение программы может уйти очень далеко от точки вызова функции вывода к тому моменту, когда мы увидим реальный вывод на экране или результат окажется в файле. Более того, система не тратит свои силы и время на вывод небольших объемов информации. В стандартном режиме вывод осуществляется только при накоплении в буфере вывода достаточного количества байтов. Если программа печатает мало, то буфер может не выводиться вплоть до окончания работы программы, хотя вызовы функции вывода могут быть разбросаны по всему коду. Это не приносит особых неудобств, если программа работает штатно. Но если возникает исключительная ситуация аварийного завершения, программа падает, то вполне может быть, что к моменту падения буфер вывода еще не был передан на внешние устройства или, как еще говорят, буфер не был вытолкнут. При аварийном

завершении система освобождает все ресурсы, связанные с задачей, и поэтому мы не увидим информации, которая была к этому моменту накоплена в буфере, но еще не была выведена.

Функция `printf` также имеет свой внутренний буфер, т. е. не сразу передает информацию для печати в ОС (которая затем может буферизировать получаемые данные). Для принудительного выталкивания буфера в библиотеке ввода-вывода имеется функция `int fflush(stdout)`. Аналогичного результата обычно можно достичь, если при форматном выводе функцией `printf` в конце печатаемых данных добавить символ новой строки `'\n'`. Именно поэтому при отладочной печати каждый вывод на экран рекомендуется завершать символом `'\n'`. Отметим, что при необходимости буферизацию можно отключить с помощью функции `setbuf(stdout, NULL)`.

Хороший программист должен учиться на ошибках других программистов. Чтобы потом мог обучить новых программистов тем же ошибкам.

Список литературы

1. *Ахо А. В., Хопкрофт Дж. Э., Ульман Дж. Д.* Структуры данных и алгоритмы. М.; СПб.; Киев: Вильямс, 2001.

2. *Бахвалов Н. С., Корнев А. А., Чижонков Е. В.* Численные методы. Решения задач и упражнения. Изд. 2-е, испр. и доп. М.: Лаборатория знаний, 2016. (Сер. «Классический университетский учебник»).

3. *Валединский В. Д., Корнев А. А.* Методы программирования в примерах и задачах. М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2000.

4. *Валединский В. Д., Пронкин Ю. Н.* Вычислительные системы и программирование. Организация вычислительных систем. М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2006.

5. *Валединский В. Д., Пронкин Ю. Н.* Вычислительные системы и программирование. Системы хранения данных. М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2006.

6. *Керниган Б. У., Ритчи Д. М.* Язык программирования С. М.; СПб.; Киев: Вильямс, 2019.

7. *Кристофер В. В. Дж., Седжвик Р.* Алгоритмы на C++. Анализ структуры данных. Сортировка. Поиск. Алгоритмы на графах. Руководство. М.; СПб.; Киев: Вильямс, 2019.

8. *Sedgewick R., Flajolet P.* An introduction to the analysis of algorithms. 2nd ed. Addison-Wesley, 2013.

9. *Sedgewick R., Wayne K.* Algorithms. 4th ed. Addison-Wesley, 2011. `https://algs4.cs.princeton.edu/home/`.

10. *Столяров А. В.* Программирование. Введение в профессию. Т. 1. Азы программирования. МАКС Пресс, 2016.

11. *Столяров А. В.* Программирование. Введение в профессию. Т. 2. Низкоуровневое программирование. МАКС Пресс, 2016.

12. *Страуструп Б.* Язык программирования C++. М.: Бином, 2017.

Учебное издание

Валединский Владимир Дмитриевич, **Корнев** Андрей Алексеевич

Методы программирования
в задачах и примерах на C/C++

Электронное издание сетевого распространения

Текст публикуется в авторской редакции

Макет утвержден 06.06.2023. Формат 60×90/16. Усл. печ. л. 25,9. Изд. № 12476.

Издательство Московского университета. 119991, Москва, ГСП-1,
Ленинские горы, д. 1, стр. 15 (ул. Академика Хохлова, 11).
Тел.: (495) 939-32-91; e-mail: secretary@msupress.com
Отдел реализации. Тел.: (495) 939-33-23; e-mail: zakaz@msupress.com

Издательство
Московского
университета

ISBN 978-5-19-011886-5