МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

КАФЕДРА «ЭВМ И СИСТЕМЫ»

Л.Н. ГРИСЕВИЧ

КОНСПЕКТ ЛЕКЦИЙ

ПО ДИСЦИПЛИНЕ

«ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ»

для студентов специальностей
1-40 02 01 «Вычислительные машины, системы и сети»
и 1-36 04 02 «Промышленная электроника»

Брест 2014

УДК 004.42
ББК 32.973-01+22.18я7
Г 85

Рецензенты: Матюшков Л.П., к.т.н.,
профессор кафедры ИИТ БрГТУ
Злобин Г.Г., к.т.н., доцент кафедры радиофизики
Львовского национального университета
им. И. Франко

Грисевич Л.Н.

Г85 Основы алгоритмизации и программирования: конспект лекций для студентов специальностей 1-40 02 01 «Вычислительные машины, системы и сети» и 1-36 04 02 «Промышленная электроника». – Брест: Изд-во БрГТУ, 2014. – 232 с.: ил. – 22, табл. – 8.

ISBN 978-985-493-275-0

Конспект лекций содержит теоретический курс программирования на языке Си, включающий темы: история развития средств вычислительной техники, представление об алгоритмах и их записи, представление числовой информации, язык программирования Си, структура простой программы, функции ввода/вывода, операторы языка Си, массивы и указатели в Си, алгоритмы поиска и сортировки, рекурсия, структуры и функции пользователя, файловый ввод/вывод, консольный ввод-вывод, работа с памятью в Си, структуры данных и абстрактные типы данных в Си.

Конспект лекций предназначен для использования студентами специальностей 1-40 02 01 «Вычислительные машины, системы и сети» и 1-36 04 02 «Промышленная электроника» при выполнении лабораторных работ по дисциплине «Основы алгоритмизации и программирования».

УДК 004.42
ББК 32.973-01+22.18я7
Г 85

ISBN 978-985-493-275-0

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ 7

ВВЕДЕНИЕ 10

1 ИСТОРИЯ РАЗВИТИЯ СРЕДСТВ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ 12

1.1 Электронные лампы 13
1.2 Электромеханические вычислительные машины 14
1.3 ЭВМ 1-го поколения (1950-1954) 15
1.4 ЭВМ 2-го поколения (1955-1964) 19
1.5 ЭВМ 3-го поколения (1965-1970) 21
1.6 ЭВМ 4-го поколения 22
1.7 История развития персональных ЭВМ (PC – Personal Computer) 23

2 ПРЕДСТАВЛЕНИЕ ОБ АЛГОРИТМАХ И ИХ ЗАПИСИ 30

2.1 Понятие алгоритма и его свойства 30
2.2 Основные этапы алгоритмизации 32
2.3 Правила оформления схемы алгоритма 34

3 ПРЕДСТАВЛЕНИЕ ЧИСЛОВОЙ ИНФОРМАЦИИ 38

3.1 Системы счисления 38
3.2 Правила перевода чисел 39
3.2.1 Перевод целого числа A в СС с основанием N 39
3.2.2 Перевод правильной десятичной дроби в СС с основанием N 40
3.2.3 Перевод из недесятичной СС в десятичную 41
3.3 Операции над числами в различных СС 42
3.3.1 Математические операции над двоичными числами 42
3.3.2 Операции над шестнадцатиричными числами 43

4 ЯЗЫК ПРОГРАММИРОВАНИЯ СИ 44

4.1 История создания 44
4.2 Достоинства и особенности языка Си 45
4.3 Алфавит языка Си 46
4.4 Идентификаторы 47
4.5 Ключевые (зарезервированные) слова 47
4.6 Типы данных языка Си 48
4.7 Константы и переменные 49
4.8 Данные целого типа 50
4.9 Вещественные типы данных 50
4.10 Символьный тип данных 51
4.11 «Пустой» тип void 52
4.12 Совместимость типов 52

5 СТРУКТУРА ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ СИ 53

5.1 Директивы препроцессора Си 54
5.1.1 Директива #include 55
5.1.2 Директива #define 55
5.1.3 Директива #undef 57

5.2 Комментарии 57
5.3 Операции в языке Си 58
5.3.1 Основные операции языка Си 58
5.3.2 Сокращенные операции языка Си 60
5.3.3 Операция приведения типа 61
5.3.4 Операции адресации «&» и разадресации «*» 62
5.4 Выражения. Приоритет операций 63

6 ФУНКЦИИ ВВОДА-ВЫВОДА В ЯЗЫКЕ СИ 65
6.1 Функция форматного вывода printf() 65
6.2 Функция puts() 66
6.3 Функция putchar() 66
6.4 Функция форматного ввода scanf() 66
6.5 Функция gets() 67
6.6 Функция getchar() 67

7 ОПЕРАТОРЫ ЯЗЫКА СИ 68
7.1 Условные операторы 68
7.1.1 Оператор if 68
7.1.2 Оператор выбора switch 71
7.2 Операторы цикла 74
7.2.1 Оператор while 74
7.2.2 Оператор do...while 75
7.2.3 Оператор for 75
7.2.4 Выбор оператора цикла 76
7.3 Операторы переходов break, continue, return, goto 78
7.4 Пустой оператор 81
7.5 Оператор выражение 81

8 МАССИВЫ И УКАЗАТЕЛИ В СИ 82
8.1 Одномерные массивы 82
8.2 Многомерные массивы 84
8.3 Указатели 85
8.4 Строки 87

9 АЛГОРИТМЫ ПОИСКА И СОРТИРОВКИ 89
9.1 Организация последовательного поиска 89
9.2 Организация бинарного поиска 90
9.3 Основные алгоритмы сортировки массивов 92
9.3.1 Сортировка методом пузырька 92
9.3.2 Шейкерная сортировка 93
9.3.3 Сортировка методом выбора 95
9.3.4 Сортировка вставками 96
9.3.5 Быстрая сортировка 97

10 ФУНКЦИИ ПОЛЬЗОВАТЕЛЯ В СИ 101
10.1 Функция пользователя и ее прототип 101
10.2 Функции с переменным числом аргументов 103
10.3 Командная строка. Параметры функции main() 104

11 РЕКУРСИЯ 106

11.1 Рекурсивные объекты 106
11.2 Рекурсивные процедуры и функции 106
11.3 Косвенная рекурсия 107
11.4 Бесконечная рекурсия 107
11.5 Когда рекурсия не нужна 108
11.6 Рекурсивный поиск 110
11.7 Перебор вариантов 110
11.7.1 Сочетания 110
11.7.2 Перестановки 112

12 СТРУКТУРЫ В СИ 113

12.1 Объявление шаблонов структур 113
12.2 Объявление структур-переменных 114
12.3 Доступ к компонентам структуры 115
12.4 Анонимное определение структуры и оператор определения типа typedef 116
12.5 Объединения 117

13 РАБОТА С ПАМЯТЬЮ В СИ 118

13.1 Статическая память 118
13.2 Стековая, или локальная, память 120
13.3 Динамическая память, или куча 121

14 КОНСОЛЬНЫЙ ВВОД/ВЫВОД 124

14.1 Функции консольного ввода 124
14.2 Функции консольного вывода 128

15 РАБОТА С ФАЙЛАМИ 135

15.1 Файловый ввод/вывод 135
15.2 Открытие файла: функция fopen() 135
15.3 Диагностика ошибок: функция perror() 138
15.4 Закрытие файла: функция fclose() 138
15.5 Файловый ввод/вывод в текстовом режиме 139
15.5.1 Форматный ввод-вывод: функции fscanf() и fprintf() 139
15.5.2 Понятие потока ввода или вывода 142
15.5.3 Функции scanf() и printf() ввода/вывода в стандартные потоки 144
15.5.4 Функции текстового преобразования sscanf() и sprintf() 144
15.6 Файловый ввод/вывод в бинарном режиме 146
15.6.1 Функции бинарного чтения и записи fread() и fwrite() 146
15.6.2 Пример: подсчет числа символов и строк в текстовом файле 149
15.7 Другие полезные функции ввода-вывода 151
15.8 Низкоуровневый ввод/вывод. 155
15.8.1 Открытие, создание, закрытие и удаление 156
15.8.2 Произвольный доступ – lseek() 158

16 СТРУКТУРЫ ДАННЫХ И АБСТРАКТНЫЕ ТИПЫ ДАННЫХ 159

16.1 Функциональные структуры данных 159
16.2 Рекурсивные структуры данных 160

16.3 Теоретико-множественные структуры данных 161
16.4 Абстрактные типы данных 161
16.5 Списковые структуры 162
16.5.1 АТД «Список» 162
16.5.2 Линейные списки 164
16.5.3 Очередь, стек и дек 170
16.5.4 Двусвязные списки 177
16.5.5 Иерархические списки 179
16.5.6 Ассоциативные списки 182
16.6 Деревья 184
16.6.1 АТД «Дерево» 184
16.6.2 Двоичные деревья 186
16.6.3 Деревья двоичного поиска 187
16.6.4 Деревья выражений 195
16.6.5 N-арные деревья 209
16.6.6 Частично упорядоченные деревья 216

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 221

ПРИЛОЖЕНИЯ 222

Математическая библиотека – math.h 222
Стандартная библиотека – stdlib.h 225
Библиотека для работы со строками – string.h 228
Таблица наиболее часто используемых скан-кодов 231

ПРЕДИСЛОВИЕ

Дисциплина «**Основы алгоритмизации и программирования**» является базовой для многих курсов, изучаемых студентами за последующие четыре года обучения. Базируясь на школьных знаниях математики и физики, она в свою очередь служит фундаментом для изучения инженерии аппаратного и программного обеспечения.

В течение года обучаемый знакомится с основными принципами создания и визуализации алгоритмов, парадигмой структурного программирования и реализующим ее универсальным языком программирования. Практические навыки, вырабатываемые курсом, включают алгоритмизацию и программирование инженерных задач, анализ, отладку и тестирование программного кода.

Очевидно, что выбор первого изучаемого языка программирования в значительной степени определяет стиль мышления начинающего программиста, закладывает основы подходов, в рамках которых в дальнейшем предстоит формировать навыки и умения, связанные с проблемно-ориентированными языками программирования, включая высокоуровневые объектные подходы и модели. Каким же должен быть первый язык программирования для изучения будущими инженерами – разработчиками системного программного обеспечения и встраиваемых систем?

В предлагаемом вашему вниманию конспекте лекций по основам алгоритмизации и программирования в качестве такого базового языка избран язык **Си**.

Си – это язык, созданный в свое время для написания **ОС Unix** и доступный в настоящее время практически для любой платформы, от микроконтроллеров до мощных вычислительных кластеров. **Си** на сегодняшний день – основное средство создания ядер операционных систем, драйверов устройств и просто высокопроизводительного программного кода. Кроме того, синтаксис **Си**, благодаря своей структурированности и лаконичности лег в основу множества высокоуровневых языков программирования, как общего назначения, так и специальных: **C++**, **C#**, **Java**, **PHP** – лишь несколько наиболее популярных примеров.

Си элегантен в синтаксисе и позволяет создавать легко читаемые и одновременно очень эффективные программы, легко переносимые на другие аппаратные платформы. Однако этот язык создавался для профессиональных системных программис-

тов и потому нетерпим к неаккуратному коду. Обратной стороной высокой эффективности является большая свобода, предоставленная разработчику, на плечи которого ложится вся ответственность за принятые при разработке программы решения.

В отличие от языков, изначально создававшихся как учебные, **Си** почти не контролирует программиста. Рамки высокоуровневого языка программирования в **Си** иллюзорны, их легко обойти – по невежеству или в силу принятых решений. Концепции типов данных четко очерчены, но при этом совершенно прозрачны, и сквозь них проглядывает аппаратная архитектура, в которой все сводится к машинным словам: последовательностям битов фиксированной длины, представляющим на самом деле данные либо адреса ячеек памяти. **Си** содержит удобные и эффективные средства работы с векторными данными, такими как массивы и строки, но сами абстракции этих типов сведены к режимам адресации, близким по своей природе к реализованным в аппаратуре.

Си доверяет программисту, выполняя любые указанные преобразования над типами или нестандартные сочетания синтаксических конструкций. Так, можно создавать лаконичный и эффективный код – но так же легко совершить ошибку, приводящую к катастрофическим результатам на этапе выполнения программы. Перекладывание ответственности за действия программы на плечи программиста – шаг неизбежный при разработке системных программ, драйверов устройств и программ для высокопроизводительных вычислений. На самом деле этим и обеспечивается быстрота написанного на **Си** кода.

Очевидно, что начинать изучать такой язык программирования лучше на очень простых программах – например таких, при создании которых изучаются основы алгоритмизации и структурного подхода. Практика показывает, что начинающий программист быстро усваивает, как опасно допускать небрежность при написании кода, вырабатывает необходимые навыки и достаточно быстро начинает создавать вполне работоспособный код.

Впоследствии, при изучении средств создания сложных программных систем (например, в рамках объектно-ориентированного подхода) выработанная привычка к аккуратному и вдумчивому написанию текста программы должна положительно сказаться на качестве создаваемых программных текстов и в итоге привести к выходу на рынок труда более качественных инженеров – разработчиков электронно-информационных сис-

тем. Поэтому можно с уверенностью заявлять, что язык **Си** является прекрасным выбором для начинающего системного программиста.

При создании данного конспекта лекций автору удалось достигнуть удачного баланса между глубиной изложения технических подробностей, без которых невозможно качественное овладение программированием на **Си**, и простотой представления материала.

Сложность материала разумно нарастает от главы к главе. При этом разделы, касающиеся особенностей языка программирования, чередуются с элементами алгоритмизации, абстракциями данных, типовыми задачами поиска и сортировки информации, обеспечивая равномерное наращивание квалификации обучаемого в плане овладения как особенностями языка программирования, так и различными алгоритмическими подходами.

Список разделов отражает также особенное внимание, уделенное автором к четкой тематической структуре материала. Первичным при выборе границ разделов является функциональное наполнение, и автора не смущают различия в размерах глав в том случае, когда это оправдано с точки зрения их тематической цельности.

Материал снабжен большим количеством примеров на **Си**, а в отдельных главах первой половины конспекта – более наглядными графическими схемами алгоритмов (например, для лучшего восприятия последовательности передачи управления в разделе 7 или для большей наглядности представления рекурсии в разделе 11). Примеры кода, иллюстрирующие теоретический материал, снабжены достаточно подробными комментариями и пояснениями.

Алгоритмическая часть не переусложнена и не математизирована сверх необходимого, что выгодно отличает данную работу от ряда информационных источников, упомянутых автором в ссылках и послуживших отправной точкой при выборе формы и содержания материала соответствующих тематических разделов. Например, содержимое раздела 16 (динамические списки и деревья) отличается от использованных источников более простыми схемами и менее сложной структурой примеров, при сохранении необходимой глубины изложения материала.

Доцент кафедры ЭВМиС БрГТУ Д.А. Костюк

ВВЕДЕНИЕ

Целью изучения дисциплины является подготовка специалиста, владеющего фундаментальными знаниями и практическими навыками в области основ алгоритмизации и программирования.

Задачами изучаемой дисциплины являются:

- овладение студентами теоретическими основами алгоритмизации и структурного программирования;
- овладение студентами приемами программирования на некотором процедурно-ориентированном языке программирования высокого уровня;
- приобретение студентами практических навыков программирования на некотором процедурно-ориентированном языке программирования высокого уровня, отладки и выполнения на компьютере конкретных задач.

Дисциплина «Основы алгоритмизации и программирования» является базовой для соответствующих дисциплин, изучаемых студентами на последующих курсах.

Для изучения данной дисциплины необходимы знания по физике и математике за курс средней школы, а также базовые знания по отдельным разделам высшей математики: математические ряды, методы нахождения корней уравнения, системы уравнений, численные методы, приближенное вычисление функций.

В результате изучения дисциплины обучаемый должен знать:

- основы алгоритмизации;
- основы структурного программирования программ;
- способы представления алгоритмов;
- процедурно-ориентированный алгоритмический язык программирования высокого уровня

и уметь:

- выполнять алгоритмизацию инженерных задач;
- программировать на процедурно-ориентированном алгоритмическом языке программирования;
- отлаживать и тестировать программы;
- использовать имеющееся программное обеспечение;
- анализировать исходные и выходные данные решаемых задач и формы их представления.

Язык **Си** является прекрасным выбором для начинающего программиста. Он не просто хорош для изучения основ, он также достаточно мощный и сегодня широко используется многими

разработчиками. Чтобы изучать объектно-ориентированные языки, такие как **C++** и **Java**, необходимо изучить основные концепции объектно-ориентированного программирования (**ООП**), такие как полиморфизм, наследование, инкапсуляция и прочие. Для этого необходимо ознакомиться с основными элементами программирования. **Си** – это основа любого современного языка. **C++** и **C#** напрямую основаны на **Си**. **Java** также наследует некоторые правила и синтаксис **Си**. Программы, работающие с **ООП**, написаны на **Си**.

Си может похвастаться непревзойденной эффективностью. Когда дело доходит до скорости выполнения, **Си** по прежнему не имеет себе равных. Многие части **Linux**, **Windows** и **Unix** написаны на **Си**. Если вы хотите программировать в этих системах, либо если вы хотите, чтобы ваши программы работали в них, лучше изучать **Си**.

Драйверы современных гаджетов написаны на **Си**, т.к. именно **Си** предоставляет вам доступ к основным элементам компьютера (например, он дает вам прямой доступ к памяти процессора с помощью указателей).

Большинство игр используют **Си** как основу. Никто не будет заинтересован в играх, если они затрачивают слишком много времени для выполнения команд. **Си** используется для того, чтобы создавать быстрые игры.

Си является языком «среднего» уровня. В основном языки программирования разделяются на два типа: низкого и высокого уровня. Язык **Си** сочетает в себе лучшие качества обоих типов. Используя его, можно написать как драйвер для устройства, так и клиентское приложение.

Си – язык структурных блоков. Это означает, что каждый фрагмент кода написан в отдельном блоке и не связан с кодом в следующем блоке. Благодаря этому, можно облегчить программирование, сведя к минимуму возможность нежелаемых побочных эффектов.

1 ИСТОРИЯ РАЗВИТИЯ СРЕДСТВ ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

1642 г. – начало развития технологий принято считать с Блеза Паскаля, который изобрел устройство, механически выполняющее сложение чисел. Его машина предназначалась для работы с 6-8-разрядными числами и могла только складывать и вычитать, а также имела лучший, чем все до этого, способ фиксации результата. Машина Паскаля имела размеры 36x13x8 сантиметров, этот небольшой латунный ящичек было удобно носить с собой. Инженерные идеи Паскаля оказали огромное влияние на многие другие изобретения в области вычислительной техники.

1672 г. – выдающийся немецкий математик и философ Готфрид Вильгельм Лейбниц высказал идею механического умножения без последовательного сложения. И уже через год он представил машину, которая позволяла механически выполнять четыре арифметических действия, в Парижскую академию. Для установки машины Лейбница требовался специальный стол, так она как имела внушительные размеры: 100x30x20 сантиметров.

1812 г. – английский математик Чарльз Бэббидж начал работать над так называемой разностной машиной, которая должна была вычислять любые функции, в том числе и тригонометрические, а также составлять таблицы. Свою первую разностную машину Бэббидж построил в **1822** г. и рассчитывал на ней таблицу квадратов, таблицу значений функции $y = x^2+x+41$ и ряд других таблиц. Однако из-за нехватки средств эта машина не была закончена и сдана в музей Королевского колледжа в Лондоне, где хранится и по сей день.

1818 г. – уроженец Эльзаса Карл Томас, основатель и директор двух парижских страховых обществ, сконструировал счетную машину, уделив основное внимание технологичности механизма, и назвал ее арифмометром. Уже через три года в мастерских Томаса было изготовлено 16 арифмометров, а затем и еще больше. Таким образом, Томас положил начало счетному машиностроению. Его арифмометры выпускали в течение ста лет, постоянно совершенствуя и меняя время от времени названия.

1828 г. – в России стали приспосабливать к более сложным вычислениям счеты и генерал-майор Ф.М. Свободской выставил на обозрение оригинальный прибор, состоящий из множества счетов, соединенных в общей раме. Основным условием, позволявшим быстро вычислять, было строгое соблюдение небольшого числа единообразных правил. Все операции сводились к действиям сложения и вычитания. Таким образом, прибор воплощал в себе идею алгоритмичности.

1.1 Электронные лампы

1883 г. – Томас Эдисон, пытаясь продлить срок службы лампы с угольной нитью, ввел в ее вакуумный баллон платиновый электрод и пропустил через него положительное напряжение. Заметив, что в вакууме между электродом и нитью протекает ток, он не смог найти никакого объяснения столь необычному явлению. Так американский изобретатель открыл термоэлектронную эмиссию и создал первую в мире электронную лампу.

1904 г. – английский физик Дж.А. Флеминг (1849–1945) создал свой диод – двухэлектродную лампу.

Октябрь 1906 г. – американский инженер Ли де Форест изобрел электронную лампу – усилитель, или аудион, как он ее тогда назвал, имевший третий электрод – сетку. Им был введен принцип, на основе которого строились все дальнейшие электронные лампы, – управление током, протекающим между анодом и катодом, с помощью других вспомогательных элементов.

1910 г. – немецкие инженеры Либен, Рейнс и Штраус сконструировали триод, сетка в котором выполнялась в форме перфорированного листа алюминия и помещалась в центре баллона, а чтобы увеличить эмиссионный ток, они предложили покрыть нить накала слоем окиси бария или кальция.

1911 г. – американский физик Ч.Д. Кулидж предложил применить в качестве покрытия вольфрамовой нити накала окись тория – оксидный катод – и получил вольфрамовую проволоку, которая произвела переворот в ламповой промышленности.

1915 г. – американский физик Ирвинг Ленгмюр сконструировал двухэлектродную лампу – кенотрон, применяемую в качестве выпрямительной лампы в источниках питания.

1916 г. – ламповая промышленность стала выпускать особый тип конструкции ламп – генераторные лампы с водяным охлаждением.

Идея лампы с двумя сетками – тетрода была высказана в **1919** г. немецким физиком Вальтером Шоттки и независимо от него в **1923** г. – американцем Э.У. Халлом, а реализована эта идея англичанином Х.Дж. Раундом во второй полов. **20-х** годов.

1929 г. – голландские ученые Г. Хольст и Б. Теллеген создали электронную лампу с 3-мя сетками – пентод. В **1932** г. был создан гептод, в **1933** г. – гексод и пентагрид, в **1935** г. появились лампы в металлических корпусах. Дальнейшее развитие электронных ламп, улучшение их характеристик и функциональных возможностей привело к созданию на их основе совершенно новых электронных приборов.

1.2 Электромеханические вычислительные машины

В первые десятилетия **XX** века конструкторы обратили внимание на возможность применения в счетных устройствах новых элементов – **электромагнитных реле**.

1941 г. – немецкий инженер Конрад Цузе, построил вычислительное устройство, работающее на таких реле.

1943 г. – американец Говард Эйкен с помощью работ Бэббиджа на основе электромеханических реле смог построить на одном из предприятий фирмы **IBM** легендарный гарвардский «**Марк-1**» (а позднее еще и «**Марк-2**»). «**Марк-1**» имел в длину 15 метров и в высоту 2,5 метра, содержал 800 тысяч деталей, располагал 60 регистрами для констант, 72 запоминающими регистрами для сложения, центральным блоком умножения и деления, мог вычислять элементарные трансцендентные функции. Машина работала с 23-значными десятичными числами и выполняла операции сложения за 0,3 секунды, а умножения – за 3 секунды. Однако Эйкен сделал две ошибки: первая состояла в том, что обе эти машины были скорее электромеханическими, чем электронными; вторая – то, что Эйкен не придерживался той концепции, что программы должны храниться в памяти компьютера как и полученные данные.

Примерно в то же время в Англии начала работать первая вычислительная машина на реле, которая использовалась для расшифровки сообщений, передававшихся немецким кодированным передатчиком. К середине **XX** века потребность в автоматизации вычислений (в том числе для военных нужд – баллистики, криптографии и т.д.) стала настолько велика, что над созданием машин, подобных «**Марк-1**» и «**Марк-2**», работало несколько групп исследователей в разных странах.

Работа по созданию первой электронно-вычислительной машины была начата в **1937** г. в США профессором Джоном Атанасовым, болгарином по происхождению. Эта машина была специализированной и предназначалась для решения задач математической физики. В ходе разработок Атанасов создал и запатентовал первые электронные устройства, которые впоследствии применялись довольно широко в первых компьютерах. Полностью проект Атанасова не был завершен, однако через три десятка лет в результате судебного разбирательства профессора признали родоначальником электронной вычислительной техники.

1.3 ЭВМ 1-го поколения (1950-1954)

Начиная с **1943** г. группа специалистов под руководством Говарда Эйкена, Дж. Моучли и П. Эккерта в США начала конструировать вычислительную машину на основе электронных ламп, а не на электромагнитных реле. Эта машина была названа **ENIAC** (**Electronic Numeral Integrator And Computer**) и работала она в тысячу раз быстрее, чем «**Марк-1**». **ENIAC** содержал 18 тысяч вакуумных ламп, занимал площадь 9x15 метров, весил 30 тонн и потреблял мощность 150 киловатт. **ENIAC** имел и существенный недостаток – управление им осуществлялось с помощью коммутационной панели, у него отсутствовала память, и, для того чтобы задать программу, приходилось в течение нескольких часов или даже дней подсоединять нужным образом провода. Худшим из всех недостатков была ужасающая ненадежность компьютера, так как за день работы успевало выйти из строя около десятка вакуумных ламп.

Чтобы упростить процесс задания программ, Моучли и Эккерт стали конструировать новую машину, которая могла бы хранить программу в своей памяти. В **1945** г. к работе был привлечен знаменитый математик Джон фон Нейман, который подготовил доклад об этой машине. В этом докладе фон Нейман ясно и просто сформулировал общие принципы функционирования универсальных вычислительных устройств, т.е. компьютеров. Первая действующая машина, построенная на вакуумных лампах, официально была введена в эксплуатацию **15 февраля 1946 г.** Эту машину пытались использовать для решения некоторых задач, подготовленных фон Нейманом и связанных с проектом атомной бомбы. Затем она была перевезена на Абердинский полигон, где работала до **1955** г.

ENIAC стал первым представителем 1-го поколения компьютеров. Любая классификация условна, но большинство специалистов согласилось с тем, что различать поколения следует исходя из той элементной базы, на основе которой строятся машины. Таким образом, **первое поколение** представляется **ламповыми машинами**.

Устройство и работа компьютера по «принципу фон Неймана»

Необходимо отметить огромную роль американского математика фон Неймана в становлении техники первого поколения. Нужно было осмыслить сильные и слабые стороны **ENIAC** и дать рекомендации для последующих разработок. В отчете фон Ней-

мана и его коллег Г. Голдстайна и А. Беркса (июнь **1946**) были четко сформулированы **требования к структуре компьютеров**:

• машины на электронных элементах должны работать не в десятичной, а в двоичной системе счисления;

• программа, как и исходные данные, должна размещаться в памяти машины;

• программа, как и числа, должна записываться в двоичном коде;

• трудности физической реализации запоминающего устройства, быстродействие которого соответствует скорости работы логических схем, требуют иерархической организации памяти (то есть выделения оперативной, промежуточной и долговременной памяти);

• арифметическое устройство (процессор) конструируется на основе схем, выполняющих операцию сложения, создание специальных устройств для выполнения других арифметических и иных операций нецелесообразно;

• в машине используется параллельный принцип организации вычислительного процесса (операции над числами производятся одновременно по всем разрядам).

На следующем рисунке показано, каковы должны быть связи между устройствами компьютера согласно принципам фон Неймана (одинарные линии показывают управляющие связи, пунктир – информационные).

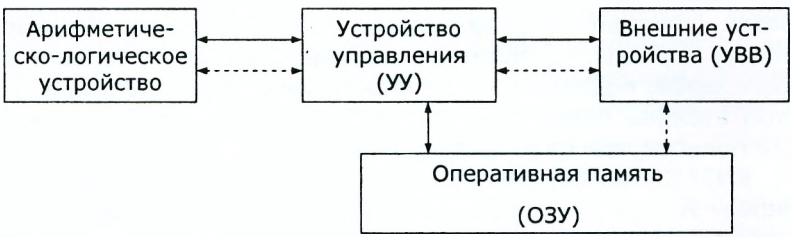


Рисунок 1.1 – Структурная схема вычислительных устройств по архитектуре фон Неймана

Принцип работы:

1) через устройство ввода-вывода (УВВ) программа вводится в оперативное запоминающее устройство (ОЗУ), причем осуществляется согласование с устройством управления (УУ) и УВВ;

2) происходит непосредственное копирование команд из УВВ в ОЗУ;

3) начальные адреса области данных и команд определяются операционной системой (ОС);

4) УУ передает адрес первой команды ОЗУ и считывает команду с ОЗУ;

5) выполняя считывание команды, УУ получает данные из ОЗУ и, при необходимости преобразуя их, отправляет их в арифметико-логическое устройство (АЛУ);

6) полученный результат передается в ОЗУ;

7) ОС выполняет очистку ОЗУ и ждет нового запуска программы.

Практически все рекомендации фон Неймана впоследствии использовались в машинах первых трех поколений, их совокупность получила название «архитектура фон Неймана». Первый компьютер, в котором были воплощены принципы фон Неймана, был построен в **1949** г. английским исследователем Морисом Уилксом. С той поры компьютеры стали гораздо более мощными, но подавляющее большинство из них сделано в соответствии с теми принципами, которые изложил в своем докладе в **1945** г. Джон фон Нейман.

Новые машины первого поколения сменяли друг друга довольно быстро.

1951 г. – заработала первая советская электронная вычислительная машина **МЭСМ**, площадью около 50 квадратных метров. **МЭСМ** имела 2 вида памяти: оперативное запоминающее устройство, в виде 4 панелей высотой в 3 метра и шириной 1 метр и долговременная память в виде магнитного барабана объемом 5000 чисел. Всего в **МЭСМ** было 6000 электронных ламп, а работать с ними можно было только спустя 1,5-2 часа после включения машины. Ввод данных осуществлялся с помощью магнитной ленты, а вывод – цифропечатающим устройством, сопряженным с памятью. **МЭСМ** могла выполнять 50 математических операций в секунду, запоминать в оперативной памяти 31 число и 63 команды (всего было 12 различных команд) и потребляла мощность, равную 25 киловаттам.

1952 г. – на свет появилась американская машина **EDWAC**. Стоит также отметить построенный ранее, в **1949** г., английский компьютер **EDSAC** (**Electronic Delay Storage Automatic Calculator**) – первую машину с хранимой программой.

1952 г. – советские конструкторы ввели в эксплуатацию **БЭСМ** – самую быстродействующую машину в Европе, а в следующем году в **СССР** начала работать «**Стрела**» – первая в Европе серийная машина высокого класса. Среди создателей отечественных машин в первую очередь следует назвать имена С.А. Лебедева, Б.Я. Базилевского, И.С. Брука, Б.И. Рамеева,

В.А. Мельникова, М.А. Карцева, А.Н. Мямлина. В 50-х годах появились и другие ЭВМ: «**Урал**», **М-2**, **М-3**, **БЭСМ-2**, «**Минск-1**», которые воплощали в себе все более прогрессивные инженерные решения.

Проекты и реализация машин «**Марк–1**», **EDSAC** и **EDVAC** в Англии и США, **МЭСМ** в СССР заложили основу для развертывания работ по созданию ЭВМ вакуумноламповой технологии – серийных ЭВМ первого поколения. Разработка первой электронной серийной машины **UNIVAC** (**Universal Automatic Computer**) была начата примерно в **1947** г. Эккертом и Моучли. Первый образец машины (**UNIVAC-1**) был построен для бюро переписи США и пущен в эксплуатацию весной **1951** г. Синхронная, последовательного действия вычислительная машина **UNIVAC-1** создана на базе ЭВМ **ENIAC** и **EDVAC**. Работала она с тактовой частотой 2,25 МГц и содержала около 5000 электронных ламп.

По сравнению с США, СССР и Англией развитие электронной вычислительной техники в Японии, ФРГ и Италии задержалось. Первая японская машина «**Фуджик**» была введена в эксплуатацию в **1956** г., серийное производство ЭВМ в ФРГ началось лишь в **1958** г.

Возможности машин первого поколения были достаточно скромны. Так, быстродействие их по нынешним понятиям было малым: от 100 («**Урал-1**») до 20 000 операций в секунду (**М-20** в **1959** г.). Эти цифры определялись в первую очередь инерционностью вакуумных ламп и несовершенством запоминающих устройств. Объем оперативной памяти был крайне мал – в среднем 2 048 чисел (слов), этого не хватало даже для размещения сложных программ, не говоря уже о данных. Промежуточная память организовывалась на громоздких и тихоходных магнитных барабанах сравнительно небольшой емкости (5 120 слов у **БЭСМ-1**). Медленно работали и печатающие устройства, а также блоки ввода данных. Если же остановиться подробнее на устройствах ввода-вывода, то можно сказать, что с начала появления первых компьютеров выявилось противоречие между высоким быстродействием центральных устройств и низкой скоростью работы внешних устройств. Кроме того, выявилось несовершенство и неудобство этих устройств. Первым носителем данных в компьютерах, как известно, была перфокарта. Затем появились перфорационные бумажные ленты или просто перфоленты. Они пришли из телеграфной техники после того, как в начале **XIX** века отец и сын из Чикаго Чарльз и Говард Крамы изобрели телетайп.

Основные характеристики ЭВМ 1-го поколения:

- строились на электронных лампах;
- производительность – 10-20 тыс. данных в секунду;
- скорость ввода – 20-30 данных в секунду;
- носители информации – перфокарты и перфоленты;
- весили около 30 тонн.

ЭВМ первого поколения, эти жесткие и тихоходные вычислители, были пионерами компьютерной техники. Они довольно быстро сошли со сцены, так как не нашли широкого коммерческого применения из-за ненадежности, высокой стоимости, трудности программирования.

1.4 ЭВМ 2-го поколения (1955-1964)

Элементной базой второго поколения стали полупроводники. Без сомнения, транзисторы можно считать одним из наиболее впечатляющих чудес XX века.

Патент на открытие **транзистора** был выдан в **1948** г. американцам Д. Бардину и У. Браттейну, а через восемь лет они вместе с теоретиком В. Шокли стали лауреатами Нобелевской премии. Скорости переключения уже первых транзисторных элементов оказались в сотни раз выше, чем ламповых, надежность и экономичность – тоже. Впервые стала широко применяться память на ферритовых сердечниках и тонких магнитных пленках, были опробованы индуктивные элементы – параметроны.

1955 г. – в США была введена в эксплуатацию первая бортовая ЭВМ для установки на межконтинентальной ракете «**Атлас**». В машине использовалось 20 тысяч транзисторов и диодов, она потребляла 4 киловатта.

1961 г. – наземные компьютеры «**стретч**» фирмы «**Бэрроуз**» управляли космическими полетами ракет «**Атлас**», а машины фирмы **IBM** контролировали полет астронавта Гордона Купера. Под контролем ЭВМ проходили полеты беспилотных кораблей типа «**Рейнджер**» к Луне в **1964** г., а также корабля «**Маринер**» к Марсу. Аналогичные функции выполняли и советские компьютеры.

1956 г. – фирмой **IBM** были разработаны плавающие магнитные головки на воздушной подушке. Изобретение их позволило создать новый тип памяти – дисковые запоминающие устройства, значимость которых была в полной мере оценена в последующие десятилетия развития вычислительной техники. Первые запоминающие устройства на дисках появились в машинах **IBM-305** и **RAMAC**. Последняя имела пакет, состоявший из 50 ме-

таллических дисков с магнитным покрытием, которые вращались со скоростью 12000 об/мин. На поверхности диска размещалось 100 дорожек для записи данных, по 10000 знаков каждая.

1958 г. – были выпущены первые серийные универсальные ЭВМ на транзисторах одновременно в США, ФРГ и Японии.

1959-1961 гг. – в Советском Союзе были созданы первые безламповые машины «**Сетунь**», «**Раздан**» и «**Раздан-2**». В **60-х** годах советские конструкторы разработали около 30 моделей транзисторных компьютеров, большинство которых стали выпускаться серийно. Наиболее мощный из них – «**Минск-32**» выполнял 65 тысяч операций в секунду. Появились целые семейства машин: «**Урал**», «**Минск**», **БЭСМ**.

Рекордсменом среди ЭВМ второго поколения стала **БЭСМ-6**, имевшая быстродействие около миллиона операций в секунду – одна из самых производительных в мире. Архитектура и многие технические решения в этом компьютере были настолько прогрессивными и опережающими свое время, что он успешно использовался почти до нашего времени.

Специально для автоматизации инженерных расчетов в Институте кибернетики Академии наук УССР под руководством академика В.М. Глушкова были разработаны компьютеры **МИР** (**1966**) и **МИР-2** (**1969**).

Важной особенностью машины **МИР-2** явилось использование телевизионного экрана для визуального контроля информации и светового пера, с помощью которого можно было корректировать данные прямо на экране, а также на ней был реализован язык программирования **АНАЛИТИК**, разработанный в **1968** г. Отличительной чертой языка являются абстрактные типы данных, вычисления в произвольных алгебрах, аналитические преобразования.

Построение таких систем, имевших в своем составе около 100 тыс. переключательных элементов, было бы просто невозможным на основе ламповой техники. Таким образом, второе поколение рождалось в недрах первого, перенимая многие его черты. Однако к середине **60-х** годов бум в области транзисторного производства достиг максимума – произошло насыщение рынка. Дело в том, что сборка электронного оборудования представляла собой весьма трудоемкий и медленный процесс, который плохо поддавался механизации и автоматизации. Таким образом, созрели условия для перехода к новой технологии, которая позволила бы приспособиться к растущей сложности схем путем исключения традиционных соединений между их элементами.

1.5 ЭВМ 3-го поколения (1965-1970)

Приоритет в изобретении интегральных схем, ставших элементной базой ЭВМ третьего поколения, принадлежит американским ученым Д.Килби и Р.Нойсу, сделавшим это открытие независимо друг от друга. Массовый выпуск интегральных схем начался в **1962** г., а в **1964** начал быстро осуществляться переход от дискретных элементов к интегральным. Упоминавшийся выше **ЭНИАК** размерами 9x15 метров в **1971** г. мог бы быть собран на пластине в 1,5 квадратных сантиметра. Началось перевоплощение электроники в микроэлектронику.

Несмотря на успехи интегральной техники и появление мини-ЭВМ, в **60-х** годах продолжали доминировать большие машины. Таким образом, третье поколение компьютеров, зарождаясь внутри второго, постепенно вырастало из него.

Первая массовая серия машин на интегральных элементах стала выпускаться в **1964** г. фирмой **IBM**. Эта серия, известная под названием **IBM-360**, оказала значительное влияние на развитие вычислительной техники второй половины **60-х** годов. Она объединила целое семейство ЭВМ с широким диапазоном производительности, причем совместимых друг с другом. Последнее означало, что машины стало возможно связывать в комплексы, а также без всяких переделок переносить программы, написанные для одной ЭВМ, на любую другую из этой серии. Таким образом, впервые было выявлено коммерчески выгодное требование стандартизации аппаратного и программного обеспечения ЭВМ.

В СССР первой серийной ЭВМ на интегральных схемах была машина «**Наири-3**», появившаяся в **1970** г. Со второй половины **60-х** годов Советский Союз совместно со странами СЭВ приступил к разработке семейства универсальных машин, аналогичного системе **IBM-360**. В **1972** г. началось серийное производство стартовой, наименее мощной модели Единой Системы – ЭВМ **ЕС-1010**, а еще через год – пяти других моделей. Их быстродействие находилось в пределах от десяти тысяч (**ЕС-1010**) до двух миллионов (**ЕС-1060**) операций в секунду.

В рамках третьего поколения в США была построена уникальная машина «**ИЛЛИАК-4**», в составе которой в первоначальном варианте планировалось использовать 256 устройств обработки данных, выполненных на монолитных интегральных схемах. Позднее проект был изменен, из-за довольно высокой стоимости (более 16 миллионов долларов). Число процессоров пришлось сократить до 64, а также перейти к интегральным схемам с малой степенью интеграции. Сокращенный вариант проекта был завершен в **1972** г., номинальное быстродействие

«**ИЛЛИАК-4**» составило 200 милл. операций в секунду. Почти год этот компьютер был рекордсменом в скорости вычислений.

Именно в период развития третьего поколения возникла чрезвычайно мощная индустрия вычислительной техники, которая начала выпускать в больших количествах ЭВМ для массового коммерческого применения. Компьютеры все чаще стали включаться в информационные системы или системы управления производствами. Они выступили в качестве очевидного рычага современной промышленной революции.

1.6 ЭВМ 4-го поколения

Начало **70-х** годов знаменует переход к компьютерам четвертого поколения – на **сверхбольших интегральных схемах** (**СБИС**). Другим признаком ЭВМ нового поколения являются резкие изменения в архитектуре.

Техника четвертого поколения породила качественно новый элемент ЭВМ – **микропроцессор**. В **1971** г. пришли к идее ограничить возможности процессора, заложив в него небольшой набор операций, микропрограммы которых должны быть заранее введены в постоянную память. Оценки показали, что применение постоянного запоминающего устройства в 16 килобит позволит исключить 100-200 обычных интегральных схем. Так возникла идея микропроцессора, который можно реализовать даже на одном кристалле, а программу в его память записать навсегда. В то время в рядовом микропроцессоре уровень интеграции соответствовал плотности, равной примерно 500 транзисторам на один квадратный миллиметр, при этом достигалась очень хорошая надежность.

К середине **70-х** годов положение на компьютерном рынке резко и непредвиденно стало изменяться. Четко выделились две концепции развития ЭВМ. Воплощением первой концепции стали суперкомпьютеры, а второй – персональные ЭВМ.

Из больших компьютеров четвертого поколения на сверхбольших интегральных схемах особенно выделялись американские машины «**Крей-1**» и «**Крей-2**», а также советские модели «**Эльбрус-1**» и «**Эльбрус-2**». Первые их образцы появились примерно в одно и то же время – в **1976** г. Все они относятся к категории суперкомпьютеров, так как имеют предельно достижимые для своего времени характеристики и очень высокую стоимость.

В машинах четвертого поколения сделан отход от архитектуры фон Неймана, которая была ведущим признаком подавляющего большинства всех предыдущих компьютеров.

Многопроцессорные ЭВМ, в связи с громадным быстродействием и особенностями архитектуры, используются для реше-

ния ряда уникальных задач гидродинамики, аэродинамики, долгосрочного прогноза погоды и т.п. Наряду с суперкомпьютерами в состав четвертого поколения входят многие типы мини-ЭВМ, также опирающиеся на элементную базу из сверхбольших интегральных схем.

1.7 История развития персональных ЭВМ (PC – Personal Computer)

Хотя и персональные компьютеры относятся к ЭВМ 4-го поколения, все же возможность их широкого распространения, несмотря на достижения технологии СБИС, оставалась бы весьма небольшой.

В **1970** г. был сделан важный шаг на пути к персональному компьютеру – Маршиан Эдвард Хофф из фирмы **Intel** сконструировал интегральную схему, аналогичную по своим функциям центральному процессору большого компьютера. Так появился **первый микропроцессор Intel 4004**, который был выпущен в продажу в **1971** г. Это был настоящий прорыв, ибо микропроцессор **Intel 4004** размером менее 3 см был производительнее гигантских машин 1-го поколения. Правда, возможности **Intel 4004** были куда скромнее, чем у центрального процессора больших компьютеров того времени, – он работал гораздо медленнее и мог обрабатывать одновременно только 4 бита информации (процессоры больших компьютеров обрабатывали 16 или 32 бита одновременно), но и стоил он в десятки тысяч раз дешевле. Но рост производительности микропроцессоров не заставил себя ждать.

1972 г. – появился 8-битный микропроцессор **Intel 8008**. Размер его регистров соответствовал стандартной единице цифровой информации – байту. Процессор **Intel 8008** являлся простым развитием **Intel 4004**.

1974 г. – был создан гораздо более интересный микропроцессор **Intel 8080**. С самого начала разработки он закладывался как 8-битный чип. У него было более широкое множество микрокоманд (множество микрокоманд **8080** было расширено). Кроме того, это был первый микропроцессор, который мог делить числа. И до конца 70-х годов микропроцессор **Intel 8080** стал стандартом для микрокомпьютерной индустрии.

Несколько инженеров фирмы имели идеи по усовершенствованию **8080**. Они покинули **Intel**, чтобы реализовать их. Ими была организована **Zilog Corporation**, которая подарила миру микропроцессор **Z80**. В действительности **Z80** являлся дальнейшей разработкой микропроцессора **8080**. Было просто увеличено число его команд, что позволило создать и использовать на персональных компьютерах стандартные операционные системы.

И хотя в **1973** г. на рынке и господствовала горстка производителей, в том числе **IBM**, **DEC**, **HEWLETT-PACKARD**, и доходы этих фирм исчислялись миллиардами долларов и основывались главным образом на больших системах (мэйнфреймах) и миникомпьютерах, но до них еще не дошла важность микропроцессоров, и компании не строили планы об использовании этого новшества. Это оставило щелку для мелких предпринимателей, которые незамедлительно разработали новую технологию, радикально изменившую стандарты конструирования и применения компьютеров.

Кроме того, огромную роль в популяризации персональных компьютеров сыграли компьютерные журналы. Такие издания как «**Radio Electronics**» и «**Popular Electronics**» разжигали интерес к потенциалу микрокомпьютеров. По всей территории США возникли клубы любителей. Самым примечательным был компьютерный клуб **Homebrew**, образованный в марте **1975** г. в Менло-Парке (штат Калифорния). В состав его первых членов входили Стив Джобс и Стив Возняк, позднее основавшие компанию **Apple Macintosh**.

Поэтому, когда появился первый микрокомпьютер, на него сразу же возник огромный спрос среди тысяч любителей, интерес которых подпитывался ежемесячно появлявшимися статьями в журналах.

Этим первым микрокомпьютером был «**Altair-8800**», созданный в **1974** г. небольшой компанией в Альбукерке (штат Нью-Мексико). История его создания такова: Эд Робертс, организовавший в **1968** г. компанию **MITS** (**Micro Instrumentation and Telemetry Systems**), занимался производством калькуляторов. В 1973 г. вследствие жесткой конкуренции со стороны **Texas Instruments** он оказался на грани банкротства и вынужден был искать новую нишу на рынке. Робертса заинтересовал микропроцессор **8080**, выпущенный **Intel** в апреле **1974** г., и уверенный в том, что этот микропроцессор может стать основой микрокомпьютера, он сам создал такую машину.

И хотя возможности его были весьма ограничены (оперативная память составляла всего 256 байт, клавиатура и экран отсутствовали), а также имелись серьезные недостатки по эксплуатации, «**Altair-8800**» стал бестселлером. Тысячи любителей, всегда мечтавших о собственном компьютере, безрассудно заказывали практически бесполезную для себя вещь. Так, из маленького американского городка, началось триумфальное шествие персонального компьютера по миру, изменяя жизнь, быт и даже мышление людей.

Позже покупатели сами снабжали этот компьютер дополнительными устройствами: монитором для вывода информации,

клавиатурой, блоками расширения памяти и т.д. Вскоре эти устройства стали выпускаться другими фирмами. В конце **1975** г. Пол Аллен и Билл Гейтс (будущие основатели фирмы **Microsoft**) создали для компьютера «**Альтаир**» интерпретатор языка **Basic** (Бейсик), что позволило пользователям достаточно просто общаться с компьютером и легко писать для него программы. Это также способствовало популярности персональных компьютеров.

Успех **Альтаир-8800** заставил многие фирмы также заняться производством персональных компьютеров. Персональные компьютеры стали продаваться уже в полной комплектации, с клавиатурой и монитором, спрос на них составил десятки, а затем и сотни тысяч штук в год.

В июле **1976** г. в продажу поступил новый компьютер **Apple I**, было собрано в общей сложности порядка 200 экземпляров устройства. Цена компьютера в продаже составляла **666,66** доллара. **Apple I** был полностью собран на монтажной плате, содержащей около 30-ти микросхем, за что и считается многими первым полноценным ПК. Тем не менее, для получения рабочего компьютера, пользователи должны были добавить к нему корпус, источник питания, клавиатуру и монитор.

В **1977** г. был представлен **Apple II**, всего было произведено около 5-6 миллионов экземпляров. В отличие от других машин того времени, **Apple II** выглядел более похожим на офисный инструмент, чем на элемент электронного оборудования. Это был компьютер, который подходил для домашней обстановки, стола менеджера или школьного класса.

В **1979** г. фирма **Intel** выпустила новый микропроцессор **Intel 8086/8088**. Тогда же и появился первый сопроцессор **Intel 8087**. Тактовые частоты, на которых мог работать микропроцессор **Intel-8086/8088**: 4.77, 8 и 10 МГц.

В конце **70-х** годов распространение персональных компьютеров даже привело к некоторому снижению спроса на большие компьютеры и мини-компьютеры (мини-ЭВМ). Это стало предметом серьезного беспокойства фирмы **IBM** – ведущей компании по производству больших компьютеров, и в **1979** г. фирма **IBM** решила попробовать свои силы на рынке персональных компьютеров.

Прежде всего, в качестве основного микропроцессора компьютера был выбран новейший тогда 16-разрядный микропроцессор **Intel 8088**. Его использование позволило значительно увеличить потенциальные возможности компьютера, так как новый микропроцессор позволял работать с 1 Мбайтом памяти, а все имевшиеся тогда компьютеры были ограничены 64 Кбайтами. В компьютере были использованы и другие комплектующие

различных фирм, а его программное обеспечение было поручено разработать небольшой тогда еще фирме **Microsoft**. И таким образом в **1981** г. появилась первая версия операционной системы для компьютера **IBM PC – MS DOS 1.0**. В дальнейшем по мере совершенствования компьютеров **IBM PC** выпускались и новые версии **DOS**, учитывающие новые возможности компьютеров и предоставляющие дополнительные удобства пользователю.

В августе **1981** г. новый компьютер под названием «**IBM Personal Computer**» был официально представлен публике, и вскоре после этого он приобрел большую популярность у пользователей. **IBM PC** имел 64 Кб оперативной памяти, магнитофон для загрузки/сохранения программ и данных, дисковод и встроенную версию языка **BASIC**.

Через один-два года компьютер **IBM PC** занял ведущее место на рынке, вытеснив модели 8-битовых компьютеров.

Фирма **IBM** не сделала свой компьютер единым неразъемным устройством и не стала защищать его конструкцию патентами. Наоборот, она собрала компьютер из независимо изготовленных частей и не стала держать спецификации этих частей и способы их соединения в секрете. Напротив, принципы конструкции **IBM PC** были доступны всем желающим. Этот подход, называемый «принципом открытой архитектуры», обеспечил потрясающий успех компьютеру **IBM PC**, хотя и лишил фирму **IBM** возможности единолично пользоваться плодами этого успеха. Вот как открытость архитектуры **IBM PC** повлияла на развитие персональных компьютеров:

1. Перспективность и популярность **IBM PC** сделала весьма привлекательным производство различных комплектующих и дополнительных устройств для **IBM PC**. Конкуренция между производителями привела к удешевлению комплектующих и устройств.

2. Очень скоро многие фирмы перестали довольствоваться ролью производителей комплектующих для **IBM PC** и начали сами собирать компьютеры, совместимые с **IBM PC**. Поскольку этим фирмам не требовалось нести огромные издержки фирмы **IBM** на исследования и поддержание структуры громадной фирмы, они смогли продавать свои компьютеры значительно дешевле (иногда в 2-3 раза) аналогичных компьютеров фирмы **IBM**.

3. Пользователи получили возможность самостоятельно модернизировать свои компьютеры и оснащать их дополнительными устройствами сотен различных производителей.

Новое поколение микропроцессоров идет на смену предыдущему каждые два года и морально устаревает за 3-4 года.

Микропроцессор вместе с другими устройствами микроэлектроники позволяет создавать довольно экономичные информационные системы. Причина такой популярности микропроцессора состоит в том, что с их появлением отпала необходимость в специальных схемах обработки информации, достаточно запрограммировать ее функцию и ввести в постоянное запоминающее устройство (ПЗУ) микропроцессора.

Через короткий отрезок времени модель **IBM PC** была усовершенствована. Новая модификация получила название «расширенного» **IBM PC/XT** (**Personal Computer/eXTended version**). В данной модификации производители отказались от использования магнитофона в качестве накопителя информации, добавили второй дисковод гибких дисков, а также возможность использования жесткого диска емкостью 10-30 МБ. В настоящее время наличие жесткого диска в ПК **XT** является практически обязательным. Модель базировалась на использовании того же микропроцессора – **Intel 8088**.

В **1982** г. фирма **Intel** выпустила новый микропроцессор **Intel 80286**, который имел 134 тыс. транзисторов и был разработан по 1,5-микронной технологии (микрон – микрометр или мкм). Он мог работать с 16 Мб оперативной памяти на частотах: 8, 12 и 16 МГц. Его принципиальное новшество – защищенный режим и виртуальная память размером до 1 Гб – не нашли массового применения, процессор большей частью использовался как очень быстрый **8088**.

В том же году была выпущена новая модель компьютеров по названием **IBM PC/AT** (**Personal Computer/Advanced Technology** – «ПК усовершенствованной технологии»). В связи с использованием нового микропроцессора с сопроцессором **80287** производительность системы возросла более чем вдвое. Она укомплектована дисководами гибких дисков нового типа (с утроенным объемом хранимой информации), жестким диском от 40 МБ и выше. Шина материнской платы ПК расширена до 16 бит.

Накал конкурентной борьбы заставил разработчиков **IBM** в конечном счете отказаться от принципа «открытой архитектуры». Новое семейство моделей ПК **IBM** получило название **PS/2** (**Personal System 2** – «персональная система/2»). Она абсолютно несовместима с первым поколением на аппаратном уровне, но сохраняет совместимость на уровне программного обеспечения. В модели **PS/2** фирма **IBM** заявила о своем переходе на новую шинную архитектуру – микроканальную (**Micro Channel Architecture, MCA**). Это позволило отгородиться от сторонних производителей, но ограничило потребителей в выборе: все дополнительные устройства для этих ПК выпускала

только сама **IBM**; другие фирмы ее практически не поддерживали. Первые модели семейства **PS/2** использовали микропроцессор **Intel 80286** и фактически копировали ПК **AT**, но на базе иной архитектуры.

В **1985** г. появился **Intel 80386SX** и **Intel 80386DX**. Он открыл класс 32- разрядных процессоров. Микропроцессор **Intel 80386** имел 275 тыс. транзисторов и изготавливался по технологии 1,5 мкм. Адресуемое пространство оперативной памяти увеличилось до 4 Гб вследствие увеличения разрядности процессора с 16 бит до 32 бит. Новый микропроцессор работал на частотах: 16, 20-40 МГц.

Новая модель ПК на базе очередного поколения микропроцессоров **Intel 80386** (ПК **386**) была впервые разработана уже не **IBM**, а фирмой **Compaq**. Этот ПК может работать в реально многозадачном и многопользовательском режиме. С некоторым запозданием **IBM** выпустила компьютер такого класса – новую модель семейства **PS/2**.

В **1987** г. фирма **Microsoft** разработала версию **3.3** (**3.30**) операционной системы **MS DOS**, которая стала фактическим стандартом на последующие 3-4 года.

В **1989** г. **Intel** выпустила новый микропроцессор **80486SX/DX/DX2**, имевшие 1,2 млн транзисторов на кристалле, изготовленному по технологии 1 мкм. От 386-го существенно отличается размещением на кристалле первичного кэша и встроенного математического сопроцессора 80487. Микропроцессоры **80486** по-прежнему могли адресовать до 4 Гб оперативной памяти и работали на частотах: 25, 33, 50 и 66 МГц.

В **1991** г. финским студентом **Линусом Торвальдсом** была начата разработка ядра **ОС Linux**.

В июне **1991** г. **Microsoft** выпускает **MS-DOS 5.0**, который имеет свои особенности: обладает улучшенными интерфейсами меню оболочки, полноэкранным редактором, утилитами на диске и возможностью смены задач. Последующие версии **MS-DOS 6.0**, **MS-DOS 6.21** и **MS-DOS 6.22** кроме стандартного набора программ имеют в своем составе программы для резервного копирования, антивирусную программу и другие усовершенствования в операционной системе.

В **1992** г. появляется процессор **Intel 80486DX4**, который работает на учетверенной частоте внешней шины, что позволило увеличить тактовую частоту процессора до 100 МГц.

Следующие основные даты развития операционных систем фирмы **Microsoft** шли одновременно с развитием аппаратной части персонального компьютера.

6 апреля 1992 г. – выход **Windows 3.1**. В ней исправлено множество ошибок, повышена стабильность, добавлены некоторые новые возможности, в том числе масштабируемые шрифты TrueType. **Windows 3.x** становится самой популярной в США (по числу инсталляций) операционной средой для ПК и остается таковой до **1997** г.

27 октября 1992 г. – выход **Windows for Workgroups 3.1**. В ней интегрируются функции, ориентированные на обслуживание сетевых пользователей и рабочих групп, в том числе доставки электронной почты, совместного использования файлов и принтеров и календарного планирования. Версия 3.1 стала предвестником бума малых локальных сетей, но потерпела коммерческую неудачу, получив обидное прозвище «**Windows for Warehouse**» («**Windows для складов**»).

24 мая 1993 г. – выпуск **Windows NT** (сокращение от **New Technology** – новая технология). Для функционирования первой версии **3.1**, изначально ориентированной на аудиторию взыскательных пользователей и рынок серверов, требуется ПК высокого класса; кроме того, продукт не свободен от шероховатостей. Однако **Windows NT** хорошо принята разработчиками благодаря ее повышенной защищенности, стабильности и развитому **API** – интерфейсу **Win32**, упрощающему составление мощных программ. Проект начинается как **OS/2 3.0**, но в итоге исходный текст продукта был полностью переработан.

8 ноября 1993 г. – выпуск **Windows for Workgroups 3.11**. В ней обеспечена более полная совместимость с **NetWare** и **Windows NT**; кроме того, в архитектуру ОС внесены многие изменения, направленные на повышение производительности и стабильности и позднее нашедшие применение в **Windows 95**. Продукт был гораздо более доброжелательно встречен корпоративной Америкой.

24 августа 1995 г. после многочисленных задержек и без беспрецедентной для программного продукта рекламной шумихи на рынке выходит **Windows 95**. Потеряв голову, в очередях за ней стоят даже люди, не имеющие компьютера. **Windows 95** – самая дружественная пользователю версия **Windows**, для инсталляции которой не требуется предварительно устанавливать **DOS**; ее появление делает ПК более доступным массовому потребителю. Благодаря значительно усовершенствованному интерфейсу наконец-то ликвидировано отставание от платформы **Mac**, и компьютеры **Mac** оказываются окончательно оттесненными в узкую нишу рынка. В **Windows 95** имеется встроенный набор протоколов **TCP/IP**, утилита **Dial-Up Net-working** и допускается использование длинных имен файлов.

2 ПРЕДСТАВЛЕНИЕ ОБ АЛГОРИТМАХ И ИХ ЗАПИСИ

2.1 Понятие алгоритма и его свойства

Понятие алгоритма возникло задолго до появления вычислительных машин. На протяжении многих веков люди пользовались этим понятием интуитивно. Арабский математик **IX века Мухаммед ибн Муса Аль-Хорезми** впервые выдвинул идею о том, что решение любой поставленной математической и философской задачи может быть оформлено в виде последовательности механически выполняемых правил, т.е. может быть алгоритмизировано.

Согласно интуитивному определению, **алгоритм** – это строгая и четкая конечная система правил решения некоторого класса задач, определяющая процесс преобразования исходных данных в искомый результат.

В рамках данного определения понятие алгоритма отождествлялось с понятием метода вычисления. Различные вычислительные алгоритмы веками формулировались и успешно применялись на практике, поэтому понятие метода вычислений считалось изначально ясным и потребности в изучении самого этого понятия не возникало.

Формальные определения алгоритма появились в **30-х - 40-х годах XX века**. Можно выделить **три основных типа** универсальных алгоритмических моделей, различающихся исходным пониманием, что такое алгоритм.

Первый тип связывает понятие алгоритма с наиболее традиционными понятиями математики – вычислениями и числовыми функциями. Рекурсивные функции (т.е. функции, вызывающие сами себя) являются исторически первой формализацией понятия алгоритма данного типа. Эта модель основана на **функциональном подходе** и рассматривает понятие алгоритма с точки зрения того, что можно вычислить с его помощью.

Второй тип основан на представлении алгоритма как некоторого детерминированного устройства, способного выполнять в каждый отдельный момент некоторые примитивные операции, или инструкции. Такое представление не оставляет сомнений в однозначности алгоритма и элементарности его шагов. Основной теоретической моделью этого типа, созданной в **30-х годах**, является **машина Тьюринга**, которая представляет собой автоматную модель, в основе которой лежит анализ процесса выполнения алгоритма как совокупности набора инструкций.

Третий тип алгоритмических моделей – это преобразования слов в произвольных алфавитах, в которых элементар-

ными операциями являются подстановки, т.е. замены части слова (подслова) другим словом. Преимущество этого типа состоит в его максимальной абстрактности и возможности применить понятие алгоритма к объектам произвольной природы.

Модели **второго** и **третьего типа** довольно близки и отличаются в основном эвристическими подходами.

Примером алгоритма из школьной арифметики является **алгоритм Евклида** проверки того, является ли заданное число простым. В качестве примера алгоритма, хоть и с некоторыми оговорками, можно рассматривать и рецепт из поваренной книги. Рецепт всегда конечен (иначе исполнитель умрет с голода), но инструкции в рецепте не всегда точны («возьмите небольшую кастрюлю», «добавьте сахар по вкусу» и т.д.), поэтому и желаемое качество результата не всегда достигается.

На практике алгоритмом или программой часто называют набор точных инструкций для выполнения конкретных действий, а сами языки таких инструкций часто называют алгоритмическими языками или языками программирования. При этом под языком программирования обычно понимают язык инструкций для реально существующего вычислительного устройства – компьютера (**Fortran**, **Basic**, **Pascal**, **C++** и т.п.), а термин «алгоритмический язык» употребляется в более широком смысле и включает в себя как языки программирования, так и другие языки инструкций – например, псевдоязыки, предназначенные для описания алгоритмов.

Алгоритм должен быть составлен таким образом, чтобы исполнитель мог однозначно и точно следовать командам алгоритма и эффективно получить требуемый результат.

По этой причине **Алгоритм** должен обладать следующими **свойствами**:

- **дискретность** – описываемый процесс должен быть разбит на последовательность шагов (четко отделенные друг от друга команды), образующих дискретную структуру алгоритма;
- **понятность** – каждое предписание должно быть понятно исполнителю, для которого оно было создано;
- **определенность** – **детерминируемость** – все предписания алгоритма должны быть однозначны и не должно быть ситуаций, когда после выполнения определенного предписания алгоритма исполнителю не ясно, что выполнять следующим;
- **результативность** – при точном исполнении всех предписаний алгоритма процесс должен прекратиться за определенное конечное количество шагов и должен быть получен

определенный результат, вывод сообщения о том, что решения не существует – тоже результат;

• **массивность** – алгоритм должен быть разработан таким образом, чтобы его можно было использовать не для одной задачи, а для группы задач с использованием различных исходных данных из области допустимых значений;

• **конечность** – любой алгоритм за конечное количество шагов должен приводить к получению определенного результата, алгоритм не должен быть бесконечным и всегда должен иметь точку выхода.

Каждый алгоритм имеет дело с данными – входными, промежуточными и выходными. Данные как объекты, с которыми могут работать алгоритмы, должны быть четко определены и отличимы как друг от друга, так и от другой информации. Данные для своего размещения требуют памяти. Память обычно считается однородной и дискретной. Единицы измерения объема данных и памяти согласованы, при этом память может быть бесконечной. Если выполнение алгоритма заканчивается получением результатов, то говорят, что он применим к рассматриваемой совокупности исходных данных.

Для задания алгоритма необходимо выделить и описать следующие его элементы:

• набор объектов, составляющих совокупность данных: исходных, промежуточных и конечных;
• правило начала;
• правила непосредственной переработки информации;
• правило окончания;
• правило извлечения результатов.

Алгоритм всегда рассчитан на конкретного исполнителя, поэтому, если исполнителем является компьютер, то алгоритм должен быть записан на языке программирования.

Начальным этапом разработки любой программы является составление собственно алгоритма, т.е. схемы решения поставленной задачи. После этого создается программа – программная реализация данного алгоритма.

2.2 Основные этапы алгоритмизации

Алгоритмизация – процесс создания алгоритма решения задачи. Состоит из следующих этапов:

• разработка;
• обоснование;
• представление;
• анализ и тестирование.

1. Разработка

Разработка должна осуществляться в соответствии с правилами и принципами **структурного проектирования** (в частности с применением пошаговой детализации), делающего представление и понимание алгоритма простым.

Структурное проектирование – методология разработки сложных алгоритмов и их компонентов, представляющих алгоритм как совокупность иерархии модулей, и позволяет определенным образом структурировать данный алгоритм с целью лучшего понимания его исполнителем, сокращения сроков обработки, повышения его надежности и упрощения математического доказательства его корректности.

Основные принципы структурного проектирования:

• **принцип разделения** – алгоритм разделяется на независимые части (модули). **Модуль** – простой независимый фрагмент алгоритма, физически и логически отделенный от других модулей, реализующий только одну часть большой задачи и допускающий независимую проверку;

• **принцип пошаговой детализации алгоритма** (нисходящее структурное проектирование) – задача разбивается на отдельные подзадачи, каждой из которых ставится в соответствие свой блок и составляется алгоритм, указывающий порядок выполнения этих частей. Так как каждая из подзадач решает лишь часть поставленной задачи и она проще исходной, то она тоже может оказаться довольно сложной задачей и может требовать дальнейшего разбиения ее на подзадачи и т.д. Поэтому на дальнейших этапах алгоритмизации в качестве задач рассматриваются подзадачи, которые рассматриваются как самостоятельные. Если на некотором этапе какая-то из подзадач оказывается достаточно простой, ее дальнейшее разбиение становится ненужным. Процесс пошаговой детализации завершается.

2. Обоснование

Этап обоснования алгоритма предполагает доказательство того, что в процессе исполнения алгоритма никогда не возникнет ошибочной ситуации, и требуемая точность результата будет достигнута за конечное количество шагов. Обоснование позволяет определить область применяемости и научную уверенность в том, что алгоритм дает правильное решение задачи при всех значениях исходных данных из области применимости.

3. Представление

Представление – это запись алгоритма на обычном или алгоритмическом языке или представление его в виде схемы, таблицы.

Способы представления алгоритма:

• **словесное описание** – текст, в котором на простом языке идет описание алгоритма по пунктам;

• **алгоритмический** – на языке программирования;

• **структурная схема алгоритма**.

4. Анализ и тестирование

Анализ и тестирование включает доказательство правильности алгоритма и его тестирование на различных наборах данных. Этот этап является важным, т.к. не может привести к созданию программы, выдающей неправдоподобные результаты.

Ошибки в алгоритмах:

• **синтаксические** – запись команд;

• **логические** – для решения задачи выбран не верный метод.

Тестированием алгоритма называется процесс выполнения алгоритма с намерением найти в нем ошибки путем решения тестовых задач, которое заключается в решении определенного класса задач по данному алгоритму с определенным набором исходных данных, результат которых известен заранее и получен без использования этого алгоритма. Таким образом, для проведения тестирования требуются наборы исходных данных и соответствующие им правильные результаты решения данной задачи (если они не совпадают, то это ошибка).

Задача тестирования – найти компромиссное решение, т.е. минимальный тестовый набор данных, гарантирующий максимальное обнаружение ошибок.

2.3 Правила оформления схемы алгоритма

Правила выполнения схем определяются следующими документами:

• **ГОСТ 19.701-90** «Схема алгоритмов программ, данных и систем, условные обозначения и правила выполнения».

• **ГОСТ 19.002-80** «Схемы алгоритмов и программ. Правила выполнения».

• **ГОСТ 19.003-80** «Схемы алгоритмов и программ. Обозначения условные графические».

Данные документы в частности регулируют способы построения схем и внешний вид их элементов.

Согласно **ГОСТу 19.701-90** все схемы алгоритмов состоят из:

1) предопределенных символов;
2) пояснительного текста;
3) соединительных линий.

Предопределенные символы:

• **основные** – используются в тех случаях, когда неизвестен вид, тип процесса или носителя данных, или когда нет необходимости конкретизации этого вида или типа процесса или данных;

• **специфические** – используются, когда известен точный вид, тип процесса или носителя данных или когда он должен быть конкретизирован.

Схема – графическое представление метода решения задачи, в котором используются графические фигуры (символы) для отображения операций, данных, каких-либо действий и т.д.

Описание символов схем алгоритма

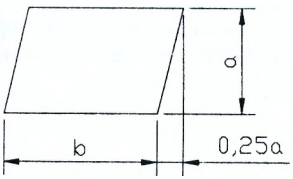
Условные обозначения для указания размеров символов схем:

a = 10, 15, 20, 25, ... (шаг 5)

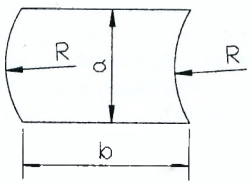
b = 1,5·a

Символы данных:

I. Основные:

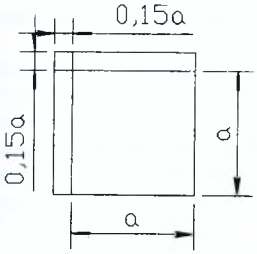


• **ввод-вывод данных.**
Отображаемые данные. Носитель данных не определен. Внутри блока необходимо пояснить выполняемые действия;

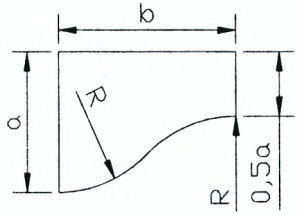


• **хранение данных.**
Отображает хранимые в запоминающем устройстве данные в виде, пригодном для обработки, носитель данных не определен.
$R = a$

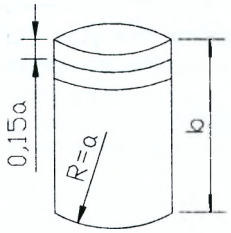
II. Специфические:



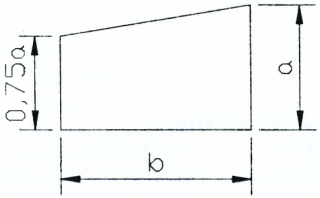
• **оперативная память.**
Отображает данные, хранящиеся в оперативном запоминающем устройстве;



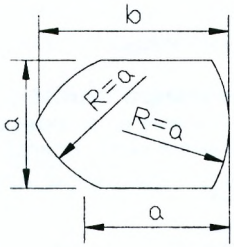
• **документ.**
Отображает данные, носителем которых является бумага.
$R = a$



• **магнитный диск.**
Отображает данные, носителем которых является магнитный диск;

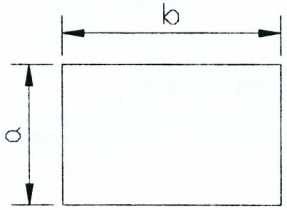


• **ввод данных с клавиатуры.**
Символ отображает данные, вводимые вручную во время обработки с устройств любого типа (клавиатура, переключатели, кнопки, световое перо, полоски со штриховым кодом);

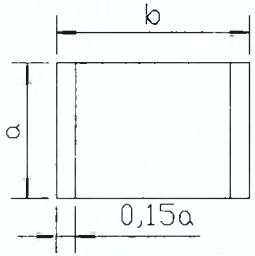


• **дисплей.**
Отображение данных в читаемой форме на устройстве отображения (экран для визуального наблюдения, индикаторы ввода информации).

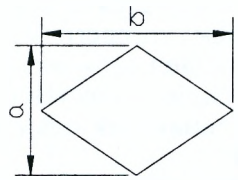
III. Символы процесса:



• **отображает обработку данных любого вида.**
Выполнение определенной операции или группы операций, приводящее к изменению значения, формы или размещения информации или к определению, по которому из нескольких направлений потока следует двигаться;

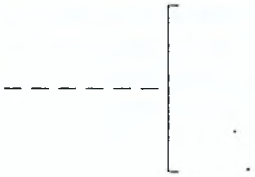


• **предопределенный процесс.**
Отображает предопределенный процесс, состоящий из одной или нескольких операций (шагов алгоритма), которые определены в другом месте, используются для обозначения подпрограмм и модулей;

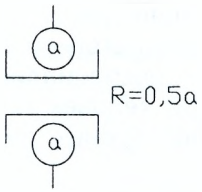


• **решение (условие).**
Выбор направления выполнения алгоритма в зависимости от некоторого условия, которое указывается внутри блока, имеет один вход и ряд альтернативных выходов, только один из которых может быть активизирован после проверки условия внутри блока.

IV. Специальные символы:

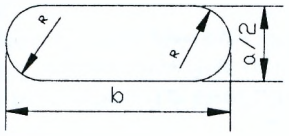


• **комментарий.**
Связывает какой-либо текст схемы и пояснение к нему;



• **соединитель.**
Служит для указания связи между прерванными линиями потока;

• **линия потока.**
Связывает отдельные символы схемы;



• **терминатор.**
Обозначает начало и конец программы.
$R = 0.25a$

3 ПРЕДСТАВЛЕНИЕ ЧИСЛОВОЙ ИНФОРМАЦИИ

3.1 Системы счисления

Система счисления (СС) – это совокупность правил наименования и изображения чисел с помощью набора знаков (чаще всего цифр и символов). Наиболее часто используют **позиционные** системы счисления. В современной информатике используются в основном следующие системы счисления: **двоичная, восмеричная**, **шестнадцатеричная** и **десятичная**.

Двоичная система счисления используется для кодирования дискретного сигнала, потребителем которого является вычислительная техника. Такое положение дел сложилось исторически, поскольку двоичный сигнал проще представлять на аппаратном уровне. В этой системе счисления для представления числа применяются два знака – **0** и **1**.

Шестнадцатеричная система счисления используется для кодирования дискретного сигнала, потребителем которого является хорошо подготовленный пользователь – специалист в области информатики. В такой форме представляется содержимое любого файла, затребованное через интегрированные оболочки операционной системы, например средствами специализированных текстовых редакторов. Используемые знаки для представления числа – десятичные цифры от **0** до **9** и буквы латинского алфавита – **A**, **B**, **C**, **D**, **E**, **F**.

Десятичная система счисления используется для кодирования дискретного сигнала, потребителем которого является так называемый конечный пользователь – неспециалист в области информатики (очевидно, что и любой человек может выступать в роли такого потребителя). Используемые знаки для представления числа – цифры от **0** до **9**.

Соответствие между первыми несколькими натуральными числами всех трех систем счисления представлено в таблице 3.1.

Для различения систем счисления, в которых представлены числа, в обозначение двоичных и шестнадцатеричных чисел вводят дополнительные реквизиты:

- для **двоичных** чисел – нижний индекс справа от числа в виде цифры **2** или букв **B** либо **b** (binary – двоичный), либо знак B или b справа от числа. Например, 101000_2 = 101000_B = 101000_b = 101000B = 101000b;
- для **шестнадцатеричных** чисел – нижний индекс справа от числа в виде числа **16** или букв **H** либо **h** (hexadecimal – шестнадцатеричный), либо знак **H** или **h** справа от числа. Например, $3AB_{16}$ = $3AB_H$ = $3AB_h$ = 3ABH = 3ABh.

Таблица 3.1 – Запись чисел в различных системах счисления

Десятичная система	Двоичная система	Шестнадцатиричная система
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

Для перевода чисел из одной системы счисления в другую существуют определенные правила. Они различаются в зависимости от формата числа – целое или правильная дробь. Для вещественных чисел используется комбинация правил перевода для целого числа и правильной дроби.

3.2 Правила перевода чисел

Правила перевода целых и дробных чисел из одной системы счисления в другую различны для целой и дробной частей числа. При переводе смешанного числа из одной системы в другую целая и дробная части числа обрабатываются порознь по определенным правилам, а затем объединяются результаты в смешанное число в новой системе счисления.

3.2.1 Перевод целого числа A в СС с основанием N

Для перевод целого числа **A** из десятичной системы счисления в систему счисления с основанием **N** число **A** необходимо последовательно делить на основание **N** той СС, в которую число переводится. Деление следует выполнять до тех пор, пока частное не окажется меньше делителя. Полученные остатки от деления и последнее частное, записанные в той СС, в которую осуществляется перевод, будут являться разрядами числа в новой СС, причем старшим разрядом будет цифра последнего частного.

Пример. Перевести целое десятичное число 46 в двоичную и шестнадцатеричную системы счисления (рисунок 3.1).

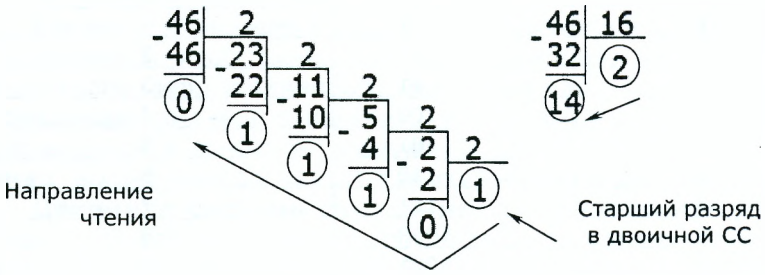


Рисунок 3.1 – Перевод числа 46 в другие СС

Следовательно, $46_{10} = 101110_2 = 2E_{16}$.

3.2.2 Перевод правильной десятичной дроби в СС с основанием N

Перевод правильной десятичной дроби в систему счисления с основанием **N** выполняется путем умножения исходной дроби последовательно на основание **N**. Цифра полученной целой части произведения является первой цифрой дробной части в новой СС. Затем умножается дробная часть полученного произведения на основание **N** новой СС и т.д. Разряды целых частей получаемых произведений являются последующими цифрами числа в новой системе по основанию **N**. Процесс прекращается, если получается нулевая дробная часть, либо при достижении требуемой точности.

Пример. Перевести десятичную дробь **0.375** в двоичную (рис. 3.2, а) и шестнадцатеричную (рис. 3.2, б) системы счисления.

а) $_{x}0.375$ $_{x}0.750$ $_{x}0.500$

2 2 2

0.750 **1**.500 **1**.000

б) $_{x}0.375$

16

6.000

Рисунок 3.2 – Перевод дроби 0.375 в другие СС

Следовательно, $0.375_{10} = 0.011_2 = 0.6_{16}$.

3.2.3 Перевод из недесятичной СС в десятичную

В позиционной системе для записи чисел используется ограниченное число знаков, интерпретация которых зависит от их места в записи числа. Количество различающихся цифр (знаков, символов) соответствует основанию системы счисления. Цифры, записанные в ряд, образуют число. Позиция цифры в изображении числа называется разрядом. «Вес» цифры зависит от занимаемой ею позиции.

Число в позиционной системе счисления представляет собой сумму степеней основания, умноженных на соответствующий коэффициент, который должен быть одной из цифр данной системы. В общем случае в позиционной системе число $\mathbf{N_X}$ с основанием **X** можно представить в следующем виде:

$$N_X = K_n \cdot X^n + K_{n-1} \cdot X^{n-1} + \ldots + K_1 \cdot X^1 + K_0 \cdot X^0 + K_{-1} \cdot X^{-1} + \ldots + K_{-m} \cdot X^{-m},$$

где **n** – количество разрядов целой части;

m – количество разрядов дробной части числа.

Наиболее известная позиционная система счисления – десятичная. В информатике широко применяется двоичная система (**X=2**), в ней для записи числа используются две цифры: **0** и **1.**

Пример. Перевести двоичное число **110110.1** в десятичную систему счисления:

$$110110.1_2 = (1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 + 1\cdot 2^{-1})_{10} =$$
$$= (32+16+4+2+0.5)_{10} = 54.5_{10}.$$

Один **двоичный** разряд соответствует одному **биту** информации. Широко используется укрупненная единица информации – **байт**, включающая **8** двоичных разрядов (**8** бит).

Для сокращения длины записи кодов команд и адресов при составлении программ используется **шестнадцатеричная** система счисления. Она удобна тем, что их основание – целая степень числа два. Так, $16_{10} = 2^4{}_{10}$. Поэтому для перевода числа из этой системы счисления в двоичную достаточно заменить каждую шестнадцатеричную цифру двоичной тетрадой.

Например, число $\mathbf{14A.1B_{16}}$ в двоичной форме записи имеет вид:

$$\underbrace{1}_{0001}\ \underbrace{4}_{0100}\ \underbrace{A}_{1010}\ .\ \underbrace{1}_{0001}\ \underbrace{B}_{1011} = 101001010.00011011$$

3.3 Операции над числами в различных СС

3.3.1 Математические операции над двоичными числами

Математические операции над числами, записанными в двоичной системе счисления, выполняются в соответствии со следующими правилами:

Таблица сложения	Таблица вычитания	Таблица умножения
0+0=0	0-0=0	0*0=0
0+1=1	1-0=1	0*1=0
1+0=1	1-1=0	1*0=0
1+1=10	10-1=1	1*1=1

Операция **сложения** выглядит так:

```
  1101
+ 0110
------
 10011
```

Операция **вычитания**:

```
  101101
-  10110
-------
   10111
```

В десятичном эквиваленте это **13 + 6 = 19**. Следует помнить, что при сложении двух единиц получается ноль, а единица переносится в старший разряд.

В десятичном эквиваленте это **45 – 22 = 23**.

Операция **умножения** рассчитывается по тому же принципу, что и в десятичной системе счисления:

```
   1011
x   110
------
   0000
  1011
 1011
-------
 1000010
```

При умножении на разряд множимое либо повторяется, либо записываются нули. В данном случае была произведена операция умножения **11 x 6 = 66**.

3.3.2 Операции над шестнадцатиричными числами

Шестнадцатиричная таблица **сложения**:

+	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
4	5	6	7	8	9	`A	B	C	D	E	F	10	11	12	13	14
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F

Шестнадцатиричная таблица **умножения**:

+	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F

Операция **сложения**:

```
  287.AB
+  65.52
-------
  2EC.FD
```

Операция **вычитания**:

```
  EC2A.82
- BD7C.9A
--------
  2EAD.E8
```

Операция **умножения**:

```
   3721
x    54
-------
   DC84
 113A5
-------
 1216D4
```

4 ЯЗЫК ПРОГРАММИРОВАНИЯ СИ

4.1 История создания

Язык **Си** исторически неразрывно связан с операционной системой **Unix**. В **1960-е** годы разрабатывались операционные системы и, соответственно, языки программирования высочайшего класса. В те времена каждый тип компьютера имел не только свою операционную систему, но и собственные языки программирования. Однако уже тогда стало очевидным, что необходимо создать универсальную операционную систему. Но для решения такой задачи понадобился и универсальный язык программирования, которым стал **Си**, разработанный в начале **1970-х** годов сотрудниками лаборатории **Bell Labs** компании **AT&T Кеном Томпсоном** и **Деннисом Ритчи**. **Unix**, разработанная в той же лаборатории и ставшая первой по-настоящему универсальной операционной системой, была первой, написанной на данном языке.

Благодаря такому повороту событий язык **Си** не только получил широкое распространение, но и появилась возможность выявить все нюансы работы с ним. Вопросы системного программирования на тот момент были самыми сложными. Подобные языки отличались высоким уровнем и были специализированы под прикладное программирование, хотя в них имелись функции и для системных работ, причем нередко только для какого-то одного типа машин.

Язык **Си** изначально создавался для решения системных задач. В первую очередь предпочтение отдавалось средствам работы с памятью, модульной организации программы и структурным конструкциям управления. Ничего больше по существу язык в себя не включал. Остальные функции накладывались на библиотеку времени исполнения. Благодаря такому подходу язык обладал большими возможностями при общей простоте.

Популярность язык завоевал благодаря еще одному фактору: создатели сумели разделить в нем независимые и зависимые от машин свойства. Данный фактор позволил писать программы для любого типа машин, независимо от архитектуры памяти и процессора. Остальные аппаратно-зависимые части кода спокойно прописывались отдельно.

Изначально **Си** создавался как универсальный, но он не остался в рамках системного программирования. К концу **80-х** годов, оттеснив **Fortran** с лидерских позиций, язык **Си** получил широчайшее распространение и использовался уже и в решении прикладных задач. Огромное значение на такое распростране-

ние имел тот факт, что в университетах, где шла подготовка будущих программистов, была установлена **Unix**, а значит **Си**.

С течением времени язык **Си** постоянно совершенствовался, самым значительным изменением можно считать изобретение строгой спецификации типов функций, которые были закреплены в **1989** г. в стандарте **ANSI**. Он же до настоящего времени и определяет язык **Си**. Однако, несмотря на все преимущества, остальные языки программирования почему-то живут и развиваются. Все дело в том, что язык **Си** оказался чересчур низкого уровня для поставленных задач **90-х** годов. Проблема заключалась в недостатке средств высокого уровня: обработке исключений и полиморфизма. Следовательно, в программах, написанных на языке **Си**, способы реализации задачи превалируют над содержанием.

Попытки исправить данные недочеты предпринимались в **80-е** годы, новые разработки были направлены на комфорт в работе. Так в **1983** г. и появился первый транслятор языка, названный **C++**. Однако всемирную популярность язык приобрел только в **1985** г. после выхода книги Страуструпа. Самым важным нововведением был механизм, позволяющий определять новые типы данных, которые к тому же можно было использовать повторно. Еще одним положительным моментом было введение механизма обработки исключений, что позволяло писать программы, отличающиеся надежностью.

4.2 Достоинства и особенности языка Си

1. **Современный** язык. Он включает в себя те управляющие конструкции, которые рекомендуются теоретическим и практическим программированием. Его структура побуждает программиста использовать в своей работе нисходящее проектирование, структурное программирование и пошаговую разработку модулей. Результатом такого подхода является надежная и читаемая программа.

2. **Эффективный язык**. Его структура позволяет наилучшим образом использовать возможности современных ЭВМ. На языке **Си** программы обычно отличаются компактностью и быстротой исполнения.

3. **Переносимый** или **мобильный** язык. Это означает, что программа, написанная на **Си** для одной вычислительной системы, может быть перенесена с небольшими изменениями (или вообще без них) на другую. Если модификации все-таки необходимы, то часто они могут быть сделаны путем простого изменения нескольких элементов в «головном» файле, который сопутствует главной программе. Язык **Си** предоставляет исключи-

тельные возможности для переноса программ. Компиляторы с данного языка реализованы почти на 40 типах вычислительных систем, начиная от 8-разрядных микропроцессоров и заканчивая **CRAY-1** одним из самых мощных в настоящее время суперкомпьютеров.

4. **Мощный** и **гибкий язык**. Например, большая часть операционной системы **UNIX** написана на языке **Си**. Программы, написанные на **Си**, используются для решения физических и технических проблем и даже для производства мультипликационных фильмов.

5. **Удобный язык**, т.к. он структурирован и не содержит большого количества ограничений.

4.3 Алфавит языка Си

Множество символов языка **Си** можно разделить на **4 группы**:

В **первую** группу входят все буквы латинского алфавита и символ подчеркивания. Строчные буквы используются для написания ключевых слов языка. Одинаковые строчные и прописные буквы (например, **a** и **A**) имеют различные коды и при записи имен переменных (идентификаторов) в языке **Си** различаются. Буквы русского алфавита используются для ввода информации в текстах и комментариях.

Вторую группу используемых символов составляют цифры от **0** до **9**.

В **третью** группу входят специальные символы (табл. 4.1). Большинство этих знаков используется для разных целей.

Таблица 4.1 – Специальные символы

Символ	Назначение	Символ	Назначение
.	Точка	(	Круглая скобка левая
,	Запятая	)	Круглая скобка правая
;	Точка с запятой	[	Квадратная открывающая
:	Двоеточие	]	Квадратная закрывающая
?	Вопросительный знак	{	Фигурная открывающая
!	Восклицательный знак	}	Фигурная закрывающая
'	Апостроф	<	Знак «меньше»
|	Вертикальная черта	>	Знак «больше»
/	Дробная черта	=	Знак «равно»
\	Обратная черта	^	Исключающее «или»
~	Тильда	&	Знак амперсанд
+	Плюс	%	Знак процента
-	Минус	#	Номер
*	Звездочка	"	Кавычки

Четвертую группу символов составляют управляющие последовательности, используемые при вводе/выводе информации. Управляющая последовательность начинается со знака обратной черты.

4.4 Идентификаторы

Идентификатор – имя, которым обозначается некоторый объект в программе. Данные в оперативной памяти размещаются по некоторым адресам, заранее неизвестным программисту. Для того, чтобы в программе иметь возможность обращаться к данным и обрабатывать их, программист этим данным дает условные имена, которые компилятор в программе заменит адресами данных в оперативной памяти.

Для записи идентификаторов используются **буквы латинского алфавита**, **цифры** и **знак подчеркивания**. Идентификатор может начинаться с буквы или знака подчеркивания. Компилятор различает идентификаторы по первым **32**-м символам.

Идентификаторы являются составной частью языка, имеют фиксированное написание и определенный смысл и не используются для других целей.

При выборе идентификатора необходимо учитывать следующее:

- идентификатор не должен совпадать с ключевыми словами языка и именами функций из библиотеки языка **Си**;
- не рекомендуется начинать идентификатор со знака подчеркивания, т.к. этот символ используется в именах некоторых библиотечных функций, и при совпадении имен эти функции будут недоступны программе. Кроме того, к именам транслируемых функций, составляющих программу, компилятор автоматически добавляет первым символом знак подчеркивания.

Правильно выбранные идентификаторы пользователя облегчают чтение и понимание программы.

4.5 Ключевые (зарезервированные) слова

Ключевые (зарезервированные) слова – это имена, используемые в языке **Си** с некоторым заранее определенным смыслом. Данные слова нельзя применять в качестве идентификаторов объектов (данных) пользователя. Ключевые слова сообщают компилятору о типе данных, способе их организации и о последовательности выполнения операторов.

К ключевым словам относятся:

auto	**break**	**case**	**char**	**const**
continue	**default**	**do**	**double**	**else**
enum	**extern**	**float**	**for**	**if**
int	**long**	**register**	**return**	**short**
signed	**sizeof**	**struct**	**switch**	**typedef**
union	**unsigned**	**void**	**volatile**	**while.**

Ключевые слова **near**, **far**, **huge** определяют тип (размер) указателя на данные, а слова **_asm**, **cdecl**, **fortran**, **pascal** используются для организации связи с функциями, написанными на других языках программирования.

4.6 Типы данных языка Си

Следует различать **тип** данных и **модификатор** типа.

Существуют следующие **базовые типы**:

- **char** (символьный);
- **int** (целый);
- **float** (вещественный);
- **double** (вещественный с двойной точностью);
- **void** (пустой тип).

К **модификаторам** относятся:

- **signed** (знаковый);
- **unsigned** (беззнаковый);
- **short** (короткий);
- **long** (длинный).

Тип данных и модификатор типа определяют:

- формат хранения данных в оперативной памяти (внутреннее представление данных);
- диапазон значений, в пределах которого может изменяться переменная;
- операции, которые могут выполняться над данными соответствующего типа.

В зависимости от типа данных и модификатора типа для обработки будут использоваться те или иные машинные команды микропроцессора.

Все типы данных (рисунок 4.1) можно разделить на две категории: **скалярные** и **составные**.

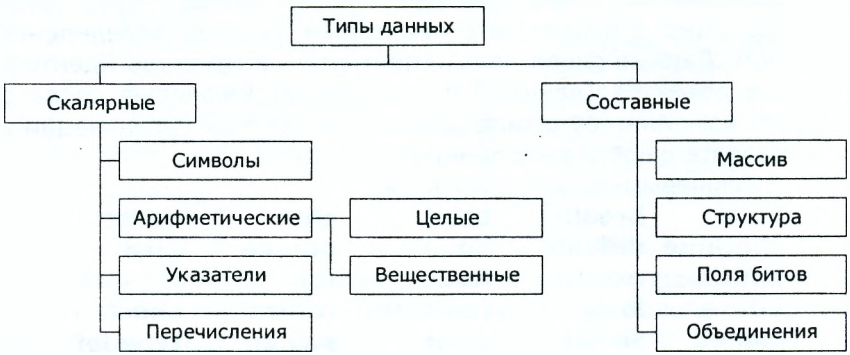


Рисунок 4.1 – Типы данных языка Си

4.7 Константы и переменные

Переменные – элементы данных, значения которых во время выполнения программы можно изменять. Неизменяемые данные – **константы**. В программе все данные перед их использованием должны быть объявлены или определены.

В операторах **определения** данных указывается тип данных и перечисляются через запятую имена переменных, имеющих данный тип. В модуле, где записано определение переменных, для каждой переменной в соответствии с типом выделяется необходимое количество байтов памяти. Выделенному полю байтов присваивается имя переменной, которое в дальнейшем используется в программе.

Отличие объявления от определения заключается в том, что при объявлении переменной место ей в памяти соответствующей функцией не выделяется, объявление лишь сообщает компилятору тип переменной. Место для значения переменной запрашивается в другой функции.

Определение переменных имеет следующий формат (квадратные скобки означают, что элемент может отсутствовать, т.е. является необязательным):

[спецификатор_класса_памяти] спецификатор_типа идентификатор [= начальное значение]
[, идентификатор [= начальное значение]];

```
int a, b, c;
int a = 2, b = 3;
int a = 4, b;
int a = b = 5;
```

Идентификатор (**имя переменной**) может быть записан с квадратными скобками, круглыми скобками или перед ним может быть один или несколько знаков * (звездочка).

Спецификатор типа – одно или несколько ключевых слов, определяющих тип переменной. Язык **Си** определяет стандартный набор основных типов данных (**int**, **char**, **double**), применяя которые пользователь может объявлять свои производные типы (массив, структура, объединение и др.). При определении переменных им можно присвоить начальное значение.

Ключевые слова **auto**, **register**, **extern**, **static** определяют **класс памяти**, или каким образом для объявляемой переменной распределяется память и в каких фрагментах программы можно использовать ее значение. Если ключевое слово, определяющее класс памяти, опущено, то класс памяти определяется по контексту.

4.8 Данные целого типа

Характеристики данных целого типа приведены в табл. 4.2.

Ключевые слова **signed** и **unsigned** необязательны. Они указывают, как интерпретируется старший бит переменной. Если указано ключевое слово **unsigned**, то левый бит данного рассматривается как часть числа. Если же указано ключевое слово **signed**, то левый бит интерпретируется как знак. Старший бит знаковых типов **int**, **short int** и **long int** хранит знак числа. Если в этом бите записан **0**, то число положительное, если он равен **1** – отрицательное.

Таблица 4.2 – Данные целого типа

Тип	Синоним	Размер в байтах	Диапазон значений
signed char	char	1	-128 ... 127
unsigned char	—	1	0 ... 255
signed int	signed, int	2*	-32 768 ... 32 767
unsigned int	unsigned	2*	0 ... 65 535
signed short int	short, signed short	2	-32 768 ... 32 767
unsigned short int	unsigned short	2	0 ... 65 535
signed long int	long, signed long	4	-2 147 483 648 ... 2 147 483 647
unsigned long int	unsigned long	4	0 ... 4 294 967 295

*- в зависимости от типа микропроцессора эти типы данных имеют разный размер: для 16-битовых – 2 байта, для 32-битовых – 4 байта.

По умолчанию все переменные считаются **signed**. Ключевые слова **signed** и **unsigned** могут предшествовать любому целому типу. Они определяются самостоятельно при определении переменной. В этом случае ключевые слова рассматриваются соответственно как **signed int** и **unsigned int**.

4.9 Вещественные типы данных

Для объявления переменных **плавающего** типа используются ключевые слова **float**, **double** и **long double**.

Характеристики данных вещественного типа приведены в таблице 4.3.

Таблица 4.3 – Данные вещественного типа

Тип	Точность	Размер в байтах	Диапазон значений
float	7 знаков	4	1.75494 E-38 ... 3.402823 E+38
double	15 знаков	8	2.225074 E-308 ... 1.797693 E+308
long double	19 знаков	10	1 E-4932 ... 1.189731 E+4932

Математически непрерывный ряд вещественных чисел внутри ЭВМ представляется дискретным рядом с определенным шагом. В связи с этим при работе с вещественными числами

возникают ошибки округления. Поэтому все проверки над вещественными числами следует проводить на диапазон, а не на точные равенства или неравенства.

При вычислениях с вещественными числами могут возникать ситуации переполнения и исчезновения порядка. Если значение вещественного типа превышает по модулю верхнюю границу диапазона значений, допустимых для данного типа, возникает переполнение, и система программирования выдает соответствующую ошибку. Если значение вещественного типа становится меньше по модулю нижней границы диапазона значений, допустимых для данного типа, возникает исчезновение порядка, и оно обращается в ноль. При этом система программирования не выдает ошибку.

4.10 Символьный тип данных

Данные типа **char** занимают в памяти **1** байт. Код от **0** до **255** в этом байте задает один из **256** возможных символов. Тип **char** является типом «целое». Данные типа **char** могут рассматриваться и как данные со знаком (**signed char**) и без знака (**unsigned char**). В случае **signed char** диапазон значений – от **-128** до **127**, а в случае **unsigned char** – от **0** до **255**. Константа типа **char** – это символ, заключенный в одиночные кавычки. Например: **'a'**, **'d'**.

Каждому символу ставился в соответствие некоторый код. Закрепление конкретных символов за кодами задается кодовыми таблицами. В операционной системе **DOS** используется кодовая таблица **ASCII**. По этому коду из таблицы описания конфигурации символа выбирается изображение этого символа, которое выводится на экран.

Коды цифр и латинских букв идут в порядке возрастания, т.е.

'0' < '1' < '2' < ... < '9' ...< 'A' < 'B' < ... < 'Z' < 'a' < 'b' < ... < 'z'

Код символов является его порядковым номером во множестве **ASCII**-кодов.

В символьных и строковых константах могут использоваться специальные управляющие знаки, приведенные в табл. 4.4.

Таблица 4.4 – Управляющие символьные константы

Управляющий знак	Наименование
\n	Переход на новую строку
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\b	Перевод курсора влево на одну позицию
\r	Перевод курсора в начало строки
\f	Новая страница

Продолжение таблицы 4.4

\a	Кратковременная подача звукового сигнала
\'	Вывод на экран одиночных кавычек
\"	Вывод на экран двойных кавычек
****	Вывод на экран черты влево (обратный слеш)

4.11 «Пустой» тип void

Слово **void** означает «**пустота**». Тип **void** в **Си** обозначает отсутствие чего-либо там, где обычно предполагается описание типа.

Тип **void** используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции.

Другое применение ключевого слова **void** состоит в описании указателя общего типа, когда заранее не известен тип объекта, на который он будет ссылаться.

Также тип **void** используется в операции приведения типов.

4.12 Совместимость типов

Если в выражениях встречаются операнды различных типов, то они преобразуются к общему типу в соответствии со следующим набором правил:

- Типы **char** и **int** могут свободно смешиваться в арифметических выражениях: каждая переменная типа **char** автоматически преобразуется в **int**.
- Арифметические преобразования работают следующим образом: если операция типа «**+**» или «*****», которая связывает два операнда (**бинарная операция**), имеет операнды разных типов, то перед выполнением операции «низший» тип преобразуется к «высшему» и получается результат «высшего» типа (например, **char** в **int**, **unsigned long** – во **float**).
- Если один из операндов имеет тип **unsigned**, то другой преобразуется в **unsigned** и результат имеет тип **unsigned**.
- Преобразования возникают и при присваивании значения одной переменной другой. При «сужающем» преобразовании в целых типах лишние биты более старших порядков отбрасываются, а дробные типы округляются.
- В любом выражении может быть осуществлено явное преобразование типа с помощью конструкции, называемой переводом (**cast**). Эта конструкция имеет вид:

  (имя_типа) выражение

5 СТРУКТУРА ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ СИ

Любая достаточно большая программа на **Си** (программисты используют термин **проект**) состоит из файлов. Файлы транслируются **Си**-компилятором независимо друг от друга и затем объединяются программой-построителем задач, в результате чего создается файл с программой, готовой к выполнению. Файлы, содержащие тексты **Си**-программы, называются **исходными**.

В языке **Си** исходные файлы бывают двух типов:

- заголовочные, или **h-файлы**;
- файлы реализации, или **Си-файлы**.

Имена заголовочных файлов имеют расширение «**.h**». Имена файлов реализации имеют расширения «**.c**» для языка **Си** и «**.cpp**», «**.cxx**» или «**.cc**» для языка **C++**.

Заголовочные файлы содержат только описания. Прежде всего, это прототипы функций. Прототип функции описывает имя функции, тип возвращаемого значения, число и типы ее аргументов. Сам текст функции в **h-файле** не содержится. Также в **h-файлах** описываются имена и типы внешних переменных, константы, новые типы, структуры и т.п. В общем, **h-файлы** содержат лишь **интерфейсы**, т.е. информацию, необходимую для использования программ, уже написанных другими программистами (или тем же программистом раньше). Заголовочные файлы лишь сообщают информацию о других программах. При трансляции заголовочных файлов, как правило, никакие объекты не создаются.

Файлы реализации, или **Си-файлы**, содержат тексты функций и определения глобальных переменных. Говоря упрощенно, **Си-файлы** содержат сами программы, а **h-файлы** – лишь информацию о программах. Представление исходных текстов в виде заголовочных файлов и файлов реализации необходимо для создания больших проектов, имеющих модульную структуру. **Заголовочные файлы** служат для передачи информации между модулями. **Файлы реализации** – это отдельные модули, которые разрабатываются и транслируются независимо друг от друга и объединяются при создании выполняемой программы.

Любая программа на **Си**, каков бы ни был ее размер, состоит из одной или более функций, указывающих фактические операции компьютера, которые должны быть выполнены. Функциям можно давать любые имена, но в программе всегда должна присутствовать функция **main**, с которой всегда начинается выполнение любой программы. Для выполнения определенных

действий функция **main** обычно обращается к другим функциям, часть из которых находится в той же самой программе, а часть в библиотеках, содержащих ранее написанные функции.

Структура программы на языке **Си** представлена в листинге 5.1.

Листинг 5.1

```
# include <stdio.h>
/*подключение библиотеки стандартного ввода-вывода*/
# include <math.h>
/*подключение библиотеки стандартных математических
функций*/
int x = 1;
int y = 2;          /* определение глобальных переменных */

int Sq (int a);     /* прототип функции */

void main ()        /* главная функция */
{
   int b;           /* объявление локальных переменных */

   b = Sq (x+y);                /* операции */
   printf ("b = %d", b);        /* вывод результата */
}
int Sq (int a)
{                               /* тело функции Sq */
   return a * a;
}
```

Исходная программа состоит из следующих объектов: **директив**, **указаний компилятору**, **объявлений** и **определений**.

Препроцессор – это программа предварительной обработки текста непосредственно перед трансляцией. Команды препроцессора называются **директивами**. Директивы препроцессора содержат символ «**#**» в начале строки. Препроцессор используется в основном для подключения **h-файлов**.

Указания компилятору – это команды, выполняемые компилятором во время процесса компиляции.

Объявления задают имена и атрибуты переменных, функций и типов, используемых в программе.

Определения – это объявления, определяющие переменные и функции. Определение переменной в дополнение к ее имени и типу задает начальное значение объявленной переменной.

5.1 Директивы препроцессора Си

Директивы препроцессора представляют собой инструкции, записанные в тексте программы на **Си** и выполняемые до транс-

ляции программы. Директивы препроцессора позволяют изменить текст программы, например заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить трансляцию части текста и т.п. Все директивы препроцессора начинаются со знака «#». После директив препроцессора точка с запятой не ставится.

5.1.1 Директива #include

Директива **#include** включает в текст программы содержимое указанного файла. Эта директива имеет две формы:

#include "имя файла"
#include <имя файла>

Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему маршрутом. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных директориях операционной системы.

Директива **#include** может быть вложенной, т.е. во включаемом файле тоже может содержаться директива **#include**, которая замещается после включения файла, содержащего эту директиву.

Директива **#include** широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому большинство программ на **Си** начинаются с этой директивы.

5.1.2 Директива #define

Директива **#define** служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы.

Директива **#define** имеет две синтаксические формы:

#define идентификатор текст
#define идентификатор (список параметров) текст

Эта директива заменяет все последующие вхождения идентификатора на **текст**. Такой процесс называется макроподстановкой. Текст может представлять собой любой фрагмент про-

граммы на **Си**, а также может и отсутствовать. В последнем случае все экземпляры идентификатора удаляются из программы.

Например:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

Эти директивы изменят в тексте программы каждое слово **WIDTH** на число **80**, а каждое слово **LENGTH** на выражение **(80+10)** вместе с окружающими его скобками.

Скобки, содержащиеся в макроопределении, позволяют избежать недоразумений, связанных с порядком вычисления операций. Например, при отсутствии скобок выражение **t=LENGTH*7** будет преобразовано в выражение **t=80+10*7**, а не в выражение **t=(80+10)*7**, как это получается при наличии скобок, и в результате получится **150**, а не **630**.

Во второй синтаксической форме в директиве **#define** имеется список формальных параметров, который может содержать один или несколько идентификаторов, разделенных запятыми. Формальные параметры в тексте макроопределения отмечают позиции, на которые должны быть подставлены фактические аргументы макровызова. Каждый формальный параметр может появиться в тексте макроопределения несколько раз.

При макровызове вслед за идентификатором записывается список фактических аргументов, количество которых должно совпадать с количеством формальных параметров.

Например:

```
#define MAX(x,y) ((x)>(y))?(x):(y)
```

Эта директива заменит фрагмент

```
t=MAX(i,s[i]);
```

на фрагмент

```
t=((i)>(s[i]))?(i):(s[i]);
```

Как и в предыдущем примере, круглые скобки, в которые заключены формальные параметры макроопределения, позволяют избежать ошибок, связанных с неправильным порядком выполнения операций, если фактические аргументы являются выражениями.

Например, при наличии скобок фрагмент

```
t=MAX(i & j, s[i] || j);
```

будет заменен на фрагмент

```
t=((i & j)>(s[i] || j)) ? (i & j) : (s[i] || j);
```

а при отсутствии скобок – на фрагмент

```
t=(i & j > s[i] || j) ? i & j : s[i] || j;
```

в котором условное выражение вычисляется в совершенно другом порядке.

5.1.3 Директива #undef

Директива **#undef** используется для отмены действия директивы **#define**. Синтаксис этой директивы следующий:

#undef идентификатор

Директива отменяет действие текущего определения **#define** для указанного идентификатора. Не является ошибкой использование директивы **#undef** для идентификатора, который не был определен директивой **#define**.

Например:

```
#undef WIDTH
#undef MAX
```

Эти директивы отменяют определение именованной константы **WIDTH** и макроопределения **MAX**.

5.2 Комментарии

Комментарии представляют собой текстовые части, предназначенные для **аннотирования** программы. Комментарии используются исключительно самим программистом. Перед лексическим анализом они исключаются из исходного текста программы транслятором. Существуют **два способа** указания на комментарий:

1. Комментарий представляет собой любую последовательность символов, помещенную после пары символов **/*.** Признаком конца комментария служит первая пара символов ***/**, встретившаяся после исходной пары **/*.** Комментарии допускаются везде, где разрешены пробельные символы. Компилятор игнорирует символы комментария, в частности, в комментариях допускается запись ключевых слов, и это не приведет к ошибке. Так как компилятор рассматривает комментарий как символ пробела, то комментарии не могут появляться внутри лексем.

Например, закомментируем строку:

```
int   /* объявление */   i   /* как счетчика */;
```

после лексического анализа даст

```
int i;
```

т.е. три лексемы: «**int**», «**i**» и «**;**».

В **Си** не допускается вложенность комментариев. Попытка комментировать приведенную выше строку в виде

```
/*  int  /* объявление */  i  /* как счетчика */;  */
```

окончится неудачей, поскольку область действия первой пары **/*** будет ограничена первой парой ***/**. Это даст в результате лексического анализа

```
i ; */
```

что приведет к генерированию состояния синтаксической ошибки.

2. **Си** позволяет создание размещаемого в одной строке комментария при помощи **двух** символов наклонной черты (**//**). Такой комментарий может начинаться в любой позиции строки и включает в себя все, что расположено до символа новой строки:

```
int  i ;  // Это комментарий
```

5.3 Операции в языке Си

Операции – это специальные комбинации символов, специфицирующие действия по преобразованию различных величин. Компилятор интерпретирует каждую из этих комбинаций как самостоятельную единицу, называемую **лексемой**.

Все операции в **Си** делятся на:

- **унарные** (действия над одним операндом);
- **бинарные** (действия над двумя операндами).

5.3.1 Основные операции языка Си

В таблице 5.1 представлен список операций, они должны использоваться точно так, как представлены в таблице: без пробелов между символами в тех операциях, которые представлены несколькими символами.

Таблица 5.1 – Операции языка **Си**

Знак	Наименование операции	Пример
	Арифметические операции	
+	Сложение	**a = b + c**; если **b = 6**, **c = 4**, то **a = 10**
-	Вычитание	**a = b - c**; если **b = 3**, **c = 8**, то **a = -5**
-	Арифметическое отрицание	**a = - b**; если **b = 132**, то **a = -132**
*	Умножение	**a = b * c**; если **b = 2**, **c = 4**, то **a = 8**
/	Деление	**a = b / c**; если **b = 7.0**, **c = 2.0**, то **a = 3.5**
%	Остаток от деления	**a = b % c**; если **b = 10**, **c = 3**, то **a = 1**

Продолжение таблицы 5.1

	Логические операции	
&&	Логическое «И» (конъюнкция)	**a = b && c**; если **b** и **c** равны **1**, то **a = 1**, в противном случае **a = 0**
||	Логическое «ИЛИ» (дизъюнкция)	**a = b || c**; если **b** и **c** равны нулю, то **a = 0**, в противном случае **a = 1**
!	Логическое отрицание (инверсия)	**a = !b**; если **b = 0**, то **a = 1**, в противном случае **a = 0**
	Побитовые операции	
&	Поразрядная конъюнкция	**a = b & c**; если оба сравниваемых бита единицы, то бит результата равен **1**, в противном случае – **0**. Если **b = 1010**, **c = 0110**, то **a = 0010**
|	Поразрядная дизъюнкция	**a = b | c**; если любой (или оба) из сравниваемых битов равен **1**, то бит результата равен **1**, в противном случае – **0**. Если **b = 1010**, **c = 0110**, то **a = 1110**
^	Поразрядное «Исключающее ИЛИ»	**a = b ^ c**; если один из сравниваемых битов равен **0**, а второй бит равен **1**, то бит результата равен **1**, в противном случае (оба бита равны **1** или **0**) – **0**. Если **b = 1010**, **c = 0110**, то **a = 1100**
~	Поразрядная инверсия	**a = ~b**; если **b = 1010**, то **a = 0101**
<<	Сдвиг влево	**a = b << c**; осуществляется сдвиг значения **b** влево на **c** разрядов; в освободившиеся разряды заносятся нули. Если **b = 1011**, **c = 2**, то **a = 101100**
>>	Сдвиг вправо	**a = b >> c**; осуществляется сдвиг значения **b** вправо на **c** разрядов; в освободившиеся разряды заносятся нули, если **b** имеет тип **unsigned**, и копии знакового бита в противном случае. Если **b = 1011**, **c = 2**, то **a = 0010**
	Операции отношения	
==	Сравнение на равенство	**a == b**; вырабатывает **1**, если **a** равно **b**, и **0** – в противном случае
>	Больше	**a > b**; вырабатывает **1**, если **a** больше **b**, и **0** – в противном случае
>=	Больше или равно	**a >= b**; вырабатывает **1**, если **a** больше или равно **b**, и **0** – в противном случае
<	Меньше	**a < b**; вырабатывает **1**, если **a** меньше **b**, и **0** – в противном случае
<=	Меньше или равно	**a <= b**; вырабатывает **1**, если **a** меньше или равно **b**, и **0** – в противном случае
!=	Не равно	**a != b**; вырабатывает **1**, если **a** не равно **b**, и **0** – в противном случае
	Операции присваивания	
=	Простое присваивание	**a = b**; **a** присваивается значение **b**
++	Унарный инкремент	**a++** (**++a**) значение **a** увеличивается на единицу
--	Унарный декремент	**a--** (**--a**) значение **a** уменьшается на единицу

Окончание таблицы 5.1

Операции адресации и разадресации		
&	Адресация	**ptr** = &**b**; **ptr** присваивается адрес **b**
*****	Разадресация	**a** = ***ptr**; **a** присваивается значение по адресу **ptr**
Операция последовательного вычисления		
,	Запятая	**a** = (**c**--, ++**b**); «,» вычисляет два своих операнда слева направо. Результат операции имеет значение и тип второго операнда. Если **c** = **2**, **b** = **3**, то **a** = **4**, **c** = **1**, **b** = **4**
Операция условного выражения		
?:	Условная (тернарная) операция	**a** = (**b** < 0) ? (-**b**) : (**b**); **a** присваивается абсолютное значение **b**
size-операция		
sizeof	Определение размера в байтах	**a** = sizeof(**int**); определяет размер памяти, которому соответствует идентификатор или тип. Переменной **a** присваивается размер типа **int**

5.3.2 Сокращенные операции языка Си

В большинстве алгоритмов при выполнении операции сложения чаще всего переменная-результат операции совпадает с первым аргументом:

x = x + y;

Здесь складываются значения двух переменных **x** и **y**, результат помещается в первую переменную **x**. Таким образом, значение переменной **x** увеличивается на значение **y**. Подобные фрагменты встречаются в программах гораздо чаще, чем фрагменты вида

x = y + z;

где аргументы и результат различны.

В случае, когда, например, сумма элементов накапливается в переменной **s**. В строке

s = s + i;

к сумме **s** прибавляется значение счетчика **i**, т.е. значение **s** увеличивается на **i**. В **Си** существует сокращенная запись операции увеличения:

s += i;

Оператор **+=** читается как «увеличить на». Строка

x += y; *// Увеличить значение **x** на **y***

эквивалентна в **Си** строке

x = x + y; *// **x** присвоить значение **x** + **y***

но короче и нагляднее.

Оператор вида **?=** существует для любой операции **?**, допустимой в **Си**. Например, для арифметических операций **+**, **-**, *****, **/**, **%** можно использовать операции, приведенные в табл. 5.2.

Таблица 5.2 – Сокращенные операции

+=	Увеличить на	**&=**	Эквивалентно операции «**&**»
-=	Уменьшить на	**|=**	Эквивалентно операции «**|**»
***=**	Домножить на	**^=**	Эквивалентно операции «**^**»
/=	Поделить на	**&&=**	Эквивалентно операции «**&&**»
%=	Поделить с остатком на	**||=**	Эквивалентно операции «**||**»

Например, строка

x *= 2.0;

удваивает значение вещественной переменной **x**.

5.3.3 Операция приведения типа

Операция приведения типа (**type cast**) является одной из самых важных в **Си**. Без знакомства с синтаксисом этой операции и сознательного ее использования написать на **Си** что-нибудь более или менее полезное невозможно.

Операция приведения типа используется, когда значение одного типа преобразуется к другому типу, в том случае, если существует некоторый разумный способ такого преобразования. Операция обозначается именем типа, заключенным в круглые скобки; она записывается **перед** ее единственным аргументом.

Пример. Пусть требуется преобразовать целое число к вещественному типу. Как известно, целые и вещественные числа по-разному представляются в компьютере. Тем не менее, существует однозначный способ преобразования целого числа типа **int** к вещественному типу **double**. В первом примере значение целой переменной **n** приводится к вещественному типу и присваивается вещественной переменной **x**:

```
double x;
int n;
. . .
x = (double) n; // Операция приведения к типу double
```

В данном случае никакой потери информации не происходит, поэтому такое приведение допустимо и по умолчанию:

```
x = n; // Эквивалентно x = (double) n;
```

Пример. Вещественное значение преобразуется к целому типу. При этом дробная часть вещественного числа отбрасывается, а знак числа сохраняется:

```
double x, y;
int n, k;
. . .
x = 3.7;
y = (-1.5);
n = (int) x;   // n присваивается значение 3
k = (int) y;   // k присваивается значение -1
```

В результате выполнения операции приведения вещественного числа к целому типу происходит отбрасывание дробной части числа, т.е. потеря информации. Поэтому, если использовать операцию приведения типа **неявно** (т.е. в результате простого присваивания целой переменной вещественного значения), например,

```
double x; int n;
. . .
n = x; // неявное приведение вещественного к целому
```

то компилятор обязательно выдаст предупреждение (или даже ошибку, если компилятор строгий). Поэтому так писать ни в коем случае нельзя!

Когда используется **явное** приведение типа, компилятору сообщается, что это не случайная ошибка, а намеренное приведение вещественного значения к целому типу, при котором дробная часть отбрасывается. При этом компилятор никаких предупреждений не выдает.

Операция приведения типа чаще всего используется для преобразования **указателей**.

Пример. Разница между **явным** и **неявным** приведением типов:

```
double a, b;
int m;
a = 1.6;
b = 1.7;
m = a + b;  // 3   m = 1.6 + 1.7 = 3.3
m = (int) a + (int) b;  // 2   m = 1.6 + 1.7 = 2
```

5.3.4 Операции адресации «&» и разадресации «*»

Операция «&» – операция взятия адреса, является унарной.

```
y = &x
```

– переменной **y** присваивается адрес переменной **x** (можно применять только к переменным и к элементам массивов).

Операция «*» – воспринимает операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого по этому адресу.

z = *y

– переменной **z** присваивается значение, расположенное по адресу **y**.

Так как все переменные перед их использованием должны быть описаны, то и переменные, содержащие какой-либо адрес, должны быть описаны. Эти переменные называются указателями и описываются следующим образом:

int *L, float *p, char *ch;

где **L, p** и **ch** – указатели.

Идентификатор типа задает тип переменных, на которые ссылается указатель. Это необходимо, так как переменные разных типов занимают разное число ячеек в памяти, а для некоторых операций, связанных с указателями, требуется знать объем памяти, отведенный под переменную.

i – указатель, ***i** – значение, записанное по адресу **i**.

i = 7;** *// Означает – записать знак **7** по адресу **i

Указатели и целые числа можно суммировать, при этом транслятор будет масштабировать приращение адреса в соответствии с типом, определенном при описании указателя. Это позволяет задавать адрес **n-ого** объекта, на который указывает указатель.

int * i, n;
i + = n;

5.4 Выражения. Приоритет операций

Выражение – это комбинация операндов и операций, значением которой является отдельная величина. Операнд – это константная или переменная величина. Каждый операнд выражения – это также выражение, представляющее отдельную величину. Операции определяют действия над операндами.

Операнд на языке **Си** – это константа, идентификатор строк, вызов функции, индексное выражение, выражение выбора структурного элемента или более сложное выражение, сформированное комбинацией операндов и операций или заключением операндов в скобки. Любой операнд, который имеет константное значение, называется **константным выражением**.

Каждый операнд имеет тип. Операнд может быть преобразован из одного типа к другому посредством операции преобра-

зования типов. Выражение преобразования типа может быть использовано в качестве операнда выражения.

Вычисление выражений выполняется по определенным правилам преобразования, группировки, ассоциативности и приоритета, которые зависят от используемых в выражениях операций, наличия круглых скобок и типов данных операндов. Последовательность вычисления операций в выражении зависит от приоритета операций и наличия круглых скобок. Существует пятнадцать категорий приоритетов, некоторые из них содержат только одну операцию. Выражения с более приоритетными операциями вычисляются первыми. В таблице 5.3 приведены приоритет и порядок выполнения операций в языке **Си**. Приоритет операций уменьшается сверху вниз. Операции, относящиеся к одной и той же категории, имеют одинаковый приоритет выполнения. Каждой категории соответствует собственное правило ассоциативности: слева направо или справа налево. Когда несколько операций одной категории появляются в выражении, то они обрабатываются в соответствии с этим правилом ассоциативности.

Таблица 5.3 – Приоритет операций

Операция	Вид операции	Порядок выполнения
() [] . ->	Выражение	Слева направо
- ~ ! * & ++ -- sizeof (тип)	Унарный	Справа налево
* / %	Мультипликативный	Слева направо
+ -	Аддитивный	Слева направо
<< >>	Сдвиг	Слева направо
< > <= >=	Отношение	Слева направо
== !=	Отношение	Слева направо
&	Побитовое «И»	Слева направо
^	Побитовое исключающее «ИЛИ»	Слева направо
|	Побитовое включающее «ИЛИ»	Слева направо
&&	Логическое «И»	Слева направо
||	Логическое «ИЛИ»	Слева направо
? :	Условная	Справа налево
= *= /= %= += -= <<= >>= &= != ^=	Простое и составное присваивание	Справа налево
,	Последовательное преобразование	Слева направо

Результат вычисления выражения, включающего несколько операций одного и того же приоритета, не зависит от порядка вычисления для операций умножения, сложения и побитовых операций.

6 ФУНКЦИИ ВВОДА-ВЫВОДА В ЯЗЫКЕ СИ

Для использования функций ввода/вывода должна быть подключена соответствующая библиотека с помощью заголовочного файла «**stdio.h**»:

#include <stdio.h>

6.1 Функция форматного вывода printf()

Функция **printf()** формально описывается следующим образом:

printf("управляющая строка", арг1, арг2 ...);

«**Управляющая строка**» содержит объекты трех типов: обычные символы, которые просто выводятся на экран дисплея; спецификации преобразования, каждая из которых вызывает вывод на экран значения очередного аргумента из последующего списка, и управляющие символьные константы.

Каждая спецификация преобразования начинается со знака **%** и заканчивается некоторым символом, задающим преобразования. Между знаком **%** и символом преобразования могут встречаться знаки:

- **ЗНАК МИНУС**, указывающий, что преобразованный параметр должен быть выровнен влево в своем поле;
- **СТРОКА ЦИФР**, задающая максимальный размер поля;
- **ТОЧКА**, отделяющая размер поля от последующей строки цифр;
- **СТРОКА ЦИФР**, задающая максимальное число символов, которые нужно вывести, или же количество цифр, которые нужно вывести справа от десятичной точки в значениях типов **float** или **double**;
- **СИМВОЛ ДЛИНЫ** «**l**», указывающий, что соответствующий аргумент имеет тип **long**.
- **СИМВОЛ ДЛИНЫ** «**L**», указывающий, что соответствующий аргумент имеет тип **long double**.

Символы преобразования:

- **d** – значением аргумента является десятичное целое число;
- **o** – значением аргумента является восьмеричное целое число;
- **x** – значением аргумента является шестнадцатеричное целое число;
- **c** – значением аргумента является символ;
- **s** – значением аргумента является строка символов (символы строки выводятся до тех пор, пока не встретится признак

конца строки или же не будет выведено число символов, заданное точностью);

• **e** – значением аргумента является вещественное десятичное число в экспоненциальной форме;

• **f** – значением аргумента является вещественное десятичное число с плавающей точкой;

• **g** – используется как **%e** или **%f** и исключает вывод незначащих нулей;

• **p** – значением аргумента является указатель (адрес);

• **u** – значением аргумента является беззнаковое целое число.

Если после знака **%** записан не символ преобразования, то он выводится на экран.

Функция **printf()** использует управляющую строку, чтобы определить, сколько всего аргументов и каковы их типы (аргументами могут быть переменные, константы, выражения, вызовы функции; главное, чтобы их значения соответствовали заданной спецификации). Также в «**управляющей строке**» указываются **управляющие символьные константы** (табл. 4.4).

6.2 Функция puts()

Функция **puts()** записывает символьную строку в стандартный поток данных (т.е. выводит ее на экран). Функция **puts()** возвращает код символа «**\n**».

```
int puts(const char *string);
```

После выполнения функции **puts()** курсор переводится на новую строку.

6.3 Функция putchar()

Функция **putchar()** записывает символ в стандартный поток данных (т.е. выводит его на экран). Функция **putchar()** возвращает выведенный на экран символ.

```
int putchar(int ch);
```

6.4 Функция форматного ввода scanf()

Функция **scanf()** формально описывается следующим образом:

```
scanf("управляющая строка", &арг1, &арг2 ...);
```

Аргументы **scanf()** должны быть указателями на соответствующие значения (перед именем переменной записывается символ **&**). Назначение указателей будет подробнее рассмотрено

позже. Символ **&** перед именем переменной воспринимается как адрес ячейки, где содержится значение этой переменной.

«**Управляющая строка**» содержит спецификации преобразования и используется для установления количества и типа аргументов. В нее могут включаться: пробелы, символы табуляции и перехода на новую строку (все они игнорируются); спецификации преобразования, состоящие из знака **%**, возможно символа ***** (запрещение присваивания), возможно числа, задающего максимальный размер поля, и самого преобразования; обычные символы, кроме **%**.

В функции **scanf()** допустимы многие из символов преобразования функции **printf()**:

- **d** – на входе ожидается десятичное число;
- **o** – на входе ожидается восьмеричное число;
- **x** – на входе ожидается шестнадцатеричное число;
- **u** – на входе ожидается беззнаковое число;
- **c** – на входе ожидается появление одиночного символа;
- **s** – на входе ожидается появление строки символов;
- **f** – на входе ожидается появление вещественного числа;
- **p** – на входе ожидается появление указателя (адреса) в виде шестнадцатеричного числа;

Перед символами **d**, **o**, **x**, **f** может стоять буква **l**. В первых трех случаях соответствующие переменные должны иметь тип **long**, а в последнем – **double**.

После выполнения функции **scanf()** осуществляется автоматический перевод курсора на следующую строку.

6.5 Функция gets()

Функция **gets()** считывает символьную строку стандартного входного потока и помещает ее по адресу, заданному указателем **buffer**; прием строки заканчивается, если функция обнаруживает символ конца строки «**\n**», данный символ удаляется и заменяется нуль-терминатором «**\0**».

char *gets (char*buffer);

Функция **gets()** возвращает указатель на считанную строку.

6.6 Функция getchar()

Функция **getchar()** считывает символ из стандартного входного потока.

int getchar(void);

Функция **getchar()** возвращает считанный символ.

7 ОПЕРАТОРЫ ЯЗЫКА СИ

Операторы в **Си** делятся на **четыре** группы:

1) условные операторы:
   - оператор разветвления **if**;
   - оператор выбора **switch**;
2) операторы организации циклов:
   - оператор **while**;
   - оператор **do ... while**;
   - оператор **for**;
3) операторы перехода:
   - оператор **break**;
   - оператор **continue**;
   - оператор **return**;
   - оператор **goto**;
4) оператор выражение, пустой оператор.

7.1 Условные операторы

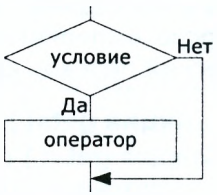
7.1.1 Оператор if

Условный оператор **if** предназначен для организации выполнения какого-либо действия в зависимости от выполнения поставленного условия.

Оператор **if** может быть записан **тремя** разными способами:

1)способ

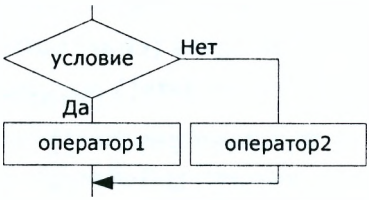
```
if (условие)
   оператор;
```



2)способ

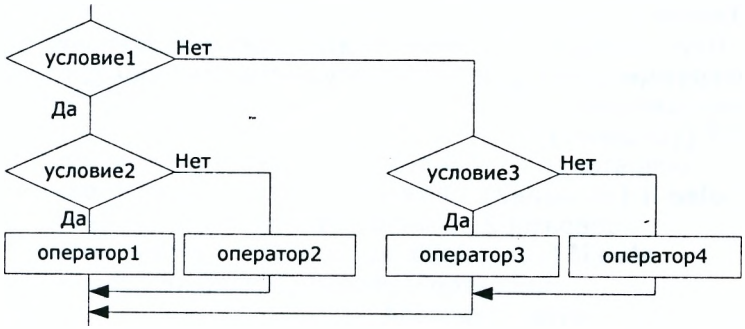
Общая форма записи оператора **if**:

```
if (условие)
{
   оператор1;
   ...
}
else
{
   оператор2;
   ...
}
```



3)способ
Оператор **if** может быть вложенным:

```
if (условие1)
{
   if (условие2)
   {
      оператор1;
      ...
   }
   else
   {
      оператор2;
      ...
   }
   ...
}
else
{
   if (условие3)
   {
      оператор3;
      ...
   }
   else
   {
      оператор4;
      ...
   }
   ...
}
```



При выполнении оператора **if** сначала выполняется условие.

Если результат анализа условия в общей форме записи – **истина** (любое отличное от нуля значение), то выполняется **оператор1**.

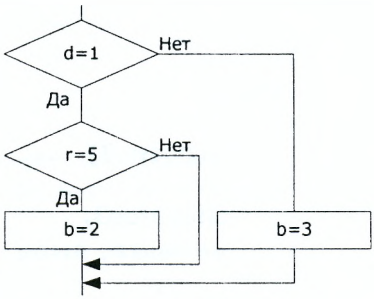
Если результат анализа условия – **ложь** (равен **0**), то выполняется **оператор2**. Если слово **else** отсутствует (как указано в способе 1), то **оператор1** пропускается, а управление передается на следующий после **if** оператор.

В качестве условия может использоваться арифметическое, логическое выражение, выражение сравнения, целое число, переменная целого типа, вызов функции с соответствующим типом возвращаемого значения.

Замечание: Если условие задано целым числом или переменной, то **условие**, **не равное 0**, всегда **истинно**, а **равное 0**, всегда **ложно**.

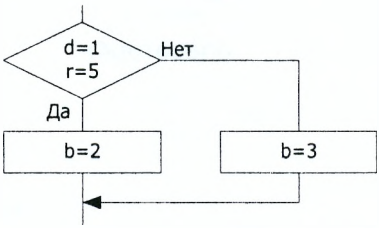
Пример:

```
if (d == 1)
{
   if (r == 5)
      b = 2;
}
else b = 3;
```



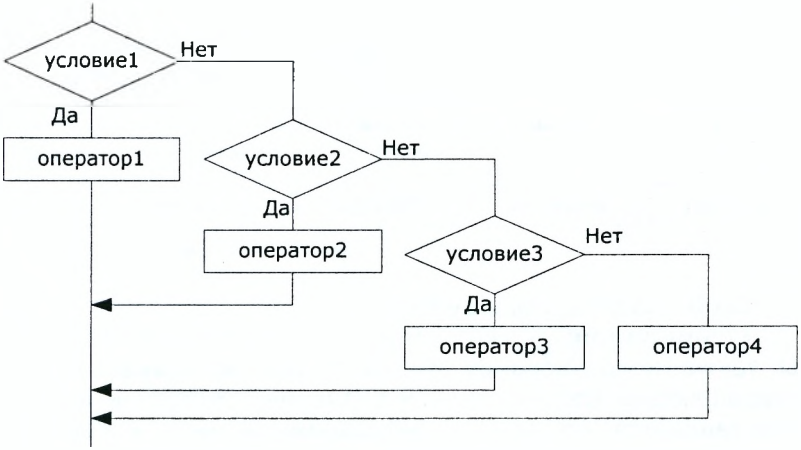
Эта запись эквивалента следующей:

```
if (d == 1 && r == 5) b = 2;
else b = 3;
```



При помощи **if** и **else** можно также составлять **else-if-конструкции**, которые могут осуществлять проверку нескольких выражений:

```
if (условие1)
   оператор1;
else if (условие2)
        оператор2;
     else if (условие3)
             оператор3;
          else оператор4;
```



Свойства **else-if–конструкции**:

• условия проверяются в том порядке, в котором они перечислены в программе;

• если одно из условий истинно, то выполняется оператор, соответствующий этому условию, а проверка оставшихся условий не производится;

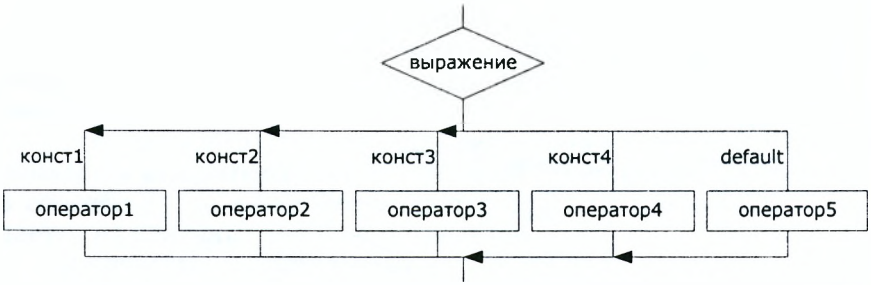
• если ни одно из проверенных условий не дало истинного результата, то выполняются операторы, относящиеся к последнему **else**;

• последний **else** является необязательным, следовательно, он и относящийся к нему оператор могут отсутствовать.

7.1.2 Оператор выбора switch

В программировании часто встречается задача выбора одного варианта решения задачи из многих существующих. Это можно сделать с помощью вложенных **if...else**. Однако более удобный способ – использование оператора выбора **switch**, общий формат которого следующий:

```
switch (выражение)
{
   case конст1: оператор1; break;
   case конст2: оператор2; break;
   case конст3: оператор3; break;
   case конст4: оператор4; break;
   default: оператор5; break;
}
```



Свойства оператора **switch**:

• **выражение** должно иметь целочисленный тип (допустимы также использование перечислений или вызов функции, возвращающей целочисленное значение). Значение выражения сопоставляется со всеми находящимися внутри конструкции константами (**const1** – **constN**), стоящими после **case**;

• оператор, указанный после **case** (в примере **operator1** – **operator4**), выполняется, если значение выражения равно соответствующей константе. Если ни с одной из констант совпадений нет, то управление передается оператору, стоящему после **default**, если, конечно, **default** есть, так как его существование необязательно. **default** может быть записан в любом месте оператора **switch**;

• константы сравниваются с **выражением** в той последовательности, в какой они перечислены в программе;

• символьные константы в **switch** (**const1** – **constN**) автоматически преобразуются в целочисленные;

• **break** передает управление за пределы оператора **switch**;

• если после какого-либо из операторов отсутствует **break**, то константа в следующем **case** считается подходящей по условию и соответствующие операторы выполняются;

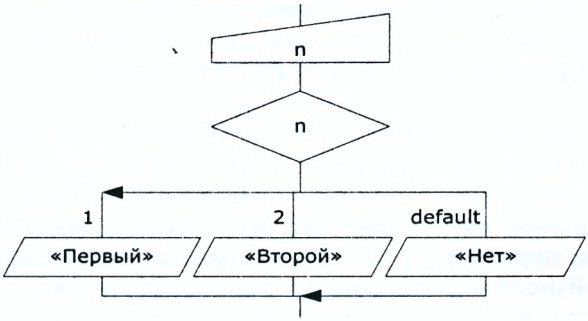
• не может быть двух констант в одном операторе **switch**, имеющих одинаковое значение, но оператор **switch**, включающий в себя другой оператор **switch**, может содержать аналогичные константы;

• **switch** отличается от **if** тем, что он может выполнять только операции строгого равенства, в то время как **if** может вычислять логические выражения и отношения.

Пример:

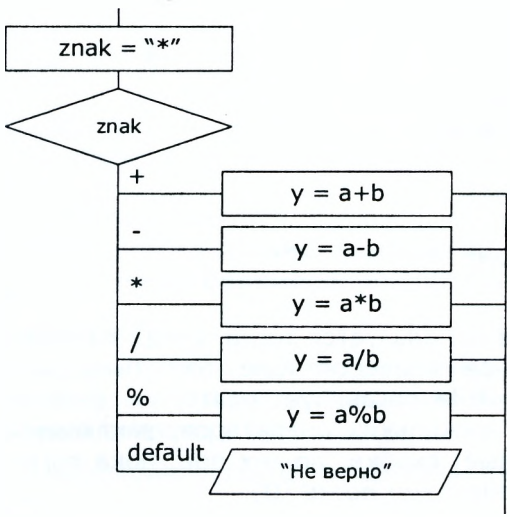
```
int n;
scanf ("%d",&n);
switch (n)
```

```
{
   case 1: printf ("Первый"); break;
   case 2: printf ("Второй"); break;
   default: printf ("Нет"); break;
}
```



Пример:

```
char znak;
znak='*';
switch (znak)
{
   case '+': y=a+b; break;
   case '-': y=a-b; break;
   case '*': y=a*b; break;
   case '/': y=a/b; break;
   case '%': y=a%b; break;
   default: printf ("Не верно"); break;
}
```



7.2 Операторы цикла

В большинстве задач, встречающихся на практике, вычисления по некоторой группе формул необходимо выполнять многократно. Такой многократно повторяющийся участок вычислительного процесса называют **циклом**. Таким образом, **цикл** – это оператор или группа операторов, повторяющихся некоторое количество раз. Каждый проход по телу цикла называет **итерацией**.

Цикл, не содержащий внутри себя других циклов, называют **простым**. Если цикл содержит внутри себя другие циклы или разветвления, то цикл называется **сложным**. Обычно при каждом повторении цикла вычисления осуществляются с новыми значениями переменных. В любом циклическом процессе в ходе вычислений необходимо решать вопрос: повторять вычисления или нет. Это решается в результате анализа некоторого условия. Анализируемую переменную называют **параметром** цикла.

Циклический процесс – это разветвляющийся процесс с двумя ветвями, из которых одна возвращается на предыдущие блоки, т.е. реализует цикл.

Способ организации цикла зависит от условия задачи. Количество повторений цикла может указываться, а может быть вычислено. Это так называемые **циклы со счетчиками**.

7.2.1 Оператор while

Общая форма записи оператора **while**:

```
while (условие)
   оператор;
```

или

```
while (условие)
{
   оператор1;
   ...
   операторN;
}
```

Цикл **while** используется тогда, когда количество повторений цикла заранее неизвестно и не может быть вычислено.

Оператор **while** организует повторное выполнение одного оператора или нескольких операторов, заключенных в операторные (фигурные) скобки, до тех пор, пока логическое **условие** не примет значение **ложь** (**0**).

Оператор **while** называют **циклом с предусловием**, так как истинность **условия** проверяется перед вхождением в цикл. Следовательно, возможна ситуация, когда цикл не выполнится ни разу.

В качестве **условия** можно использовать довольно сложные выражения, включающие и операцию присваивания.

Структура оператора **while** аналогична структуре оператора **if**. Основное отличие заключается в том, что в операторе **if** проверка условия и возможное выполнение оператора осуществляется только один раз, а в цикле **while** эти действия производятся неоднократно.

При построение цикла **while** необходимо включить в него какие-то конструкции, изменяющие величину проверяемого выражения так, чтобы в итоге оно стало ложным. В противном случае выполнение цикла никогда не завершится.

7.2.2 Оператор do...while

Если необходимо обеспечить выполнение цикла **хотя бы один раз**, то удобно использовать оператор **цикла с постусловием**:

```
do
   оператор
while (условие);
```

или

```
do
{
   оператор1;
   ...
   операторN;
} while (условие);
```

Здесь сначала выполняется оператор или группа операторов, а затем проверяется **условие**. Повторение тела цикла происходит до тех пор, пока **условие** не примет значение **ложь**.

7.2.3 Оператор for

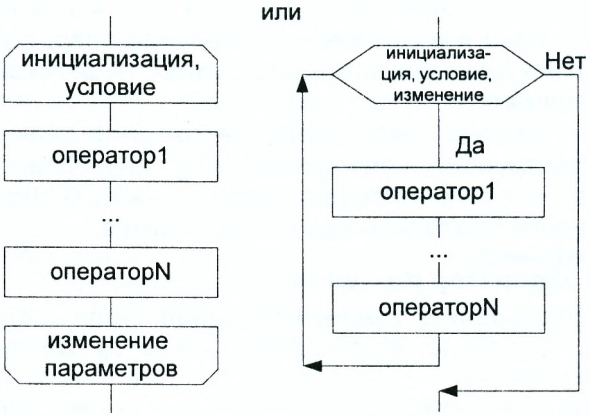
Цикл **for** используется тогда, когда количество повторений цикла заранее известно или может быть вычислено.

Форма записи оператора **for**:

```
for (инициализация; условие; изменение_параметров)
   оператор;
```

или

```
for (инициализация; условие; изменение_параметров)
{
   оператор1;
   ...
   операторN;
}
```



Инициализирующее выражение вычисляется только один раз до начала выполнения какого-нибудь из операторов.

Очередность выполнения этого оператора:

1) выполняется поле **инициализация**, которое служит для присваивания начальных значений переменным, используемым в цикле;

2) проверяется значение **условия**: если **ложь**, то завершается выполнение цикла;

3) выполняется тело цикла (оператор или группа операторов);

4) выполняется поле **изменение_параметров**, которое служит для изменения значения переменных, используемых в цикле, и значений переменных, управляющих циклом;

5) возвращение к пункту 2.

Оператор **for** является **циклом с предусловием**, поэтому решение, выполнить в очередной раз тело цикла или нет, принимается до начала его прохождения. Может случиться так, что тело цикла не будет выполнено ни разу.

7.2.4 Выбор оператора цикла

Сначала необходимо решить, требуется **цикл с предусловием** или **с постусловием**. В ряде задач тело цикла не должно

выполняться, если проверяемое условие принимает значение **ложь**, поэтому чаще используется цикл с предусловием.

Цикл с предусловием можно реализовать, используя как оператор **for**, так и оператор **while**. Для преобразования цикла **for** в цикл **while** следует первую часть оператора (инициализирующую часть) записать перед оператором **while**, а третью часть оператора **for** (изменение параметров цикла) перенести в цикл.

Для превращения цикла **while** в цикл **for** необходимо инициализацию параметров цикла и их изменение включить в оператор **for**.

Цикл **for** имеет более компактную форму записи. Ни один из перечисленных операторов организации цикла не является незаменимым.

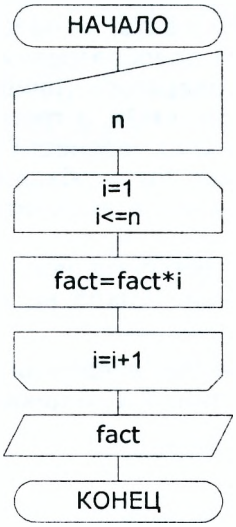
Пример. Вычисление факториала числа, введенного с клавиатуры при помощи цикла **for** (листинг 7.1) и цикла **while** (листинг 7.2).

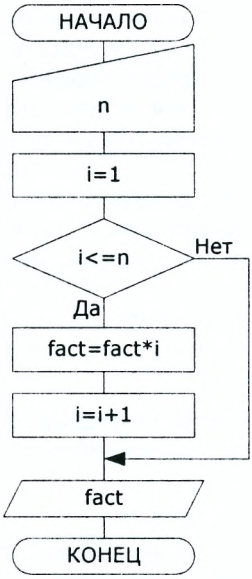
Листинг 7.1

```
// при помощи оператора for
#include<stdio.h>

void main()
{
   int i, n, fact;
   scanf ("%d", &n);
   for (i = 1; i <= n; i ++)
      fact *= i;
   printf ("%d", fact);
}
```

Листинг 7.2

```
// при помощи оператора while
#include<stdio.h>

void main()
{
   int i, n, fact;
   scanf ("%d", &n);
   i = 1;
   while (i <= n)
   {
      fact *= i;
      i++;
    printf ("%d", fact);
}
```





В языке **Си** любое число, не равное **нулю**, обозначает **истинное** условие, а **ноль** – **ложное** условие. Поэтому бесконечный цикл выглядит следующим образом:

```
while (1)
  {
    ...
  }
```

а цикл, который ни разу не выполнится, так:

```
while (0)
  {
    ...
  }
```

7.3 Операторы переходов break, continue, return, goto

Оператор **break**, стоящий в теле цикла, прекращает выполнение цикла и передает управление оператору, следующему после данного цикла, содержащего **break**.

Часто при написании программ необходимо при каком-то условии немедленно завершить данную итерацию и перейти на новую. Для этого служит оператор **continue**. Он вызывает пропуск той части цикла, которая находится после записи этого оператора.

В цикле с использованием выполнения оператора **for** после выполнения оператора **continue** управление передается на вычисление третьего выражения в скобках, а затем на проверку условия. При использовании **while** или **do...while** часть цикла, располагающаяся после **continue** не выполняется, а управление сразу передается на проверку условия в операторе **while**, что может привести к зацикливанию.

Пример использования **break** и **continue** представлен в листинге 7.3.

Листинг 7.3

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

void main()
{
   int A, B;
   while (1) // бесконечный цикл, выход только по break!
   {
      printf("\nВведите делимое и делитель:");
      scanf("%d%d", &A, &B);
      if (A==0 && B==0)
         break; // выход из цикла
      if (B == 0)
      {
         printf ( "Деление на ноль" );
         continue; // досрочный переход к новому шагу
цикла
      }
      printf ( "Частное %d остаток %d", A/B, A%B )
   }
   getch();
}
```

Оператор возврата **return** передает управление программой вызывающей функции. Обычно оператор **return** возвращает значение некоторого выражения, которое становится значением функции. Пустая функция – это функция, не возвращающая никакое значение. В этом случае с оператором **return** не связывается никакое выражение. Например:

```
return 1;
```

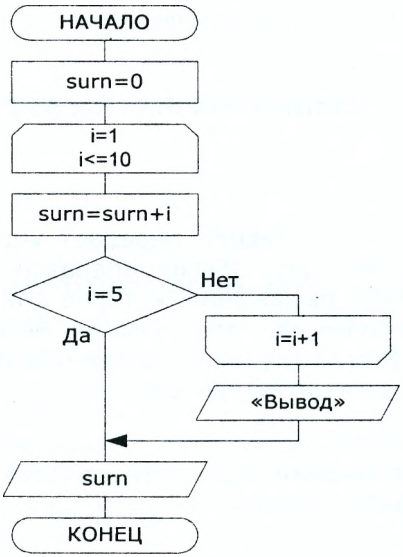
Правила выполнения безусловного перехода можно представить в следующей форме:

```
goto метка;
```

Метка – это любой идентификатор. Оператор **goto** указывает, что выполнение программы необходимо продолжить, начиная с инструкции, перед которой записана метка. В программе обязательно должна быть строка, где указана метка, поставлено двоеточие и записана инструкция, к которой должен выполниться переход. Метку можно поставить перед любой инструкцией в той функции, где находится соответствующий оператор **goto**.

Листинг 6.4

```
#include<stdio.h>
}
void main()
{
   int surn = 0;
   int i;
   for (i = 1; i <= 10; i ++)
   {
      surn + = i;
      if (i = = 5)
         goto metka;
   }
   printf ("Вывод");
 metka:
   printf ("%d", surn);
}
```



7.4 Пустой оператор

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он обычно используется в следующих случаях:

- в операторах **do**, **for**, **while**, **if** в строках, когда место оператора не требуется, но по синтаксису требуется хотя бы один оператор;
- при необходимости пометить фигурную скобку.

Синтаксис языка **Си** требует, чтобы после метки обязательно следовал оператор. Фигурная скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

```
while (i>=0) ;
```

7.5 Оператор выражение

Любое выражение, которое заканчивается точкой с запятой, является оператором. Выполнение **оператора выражение** заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызвать функцию, невозвращающую значения можно только при помощи оператора выражения. Правила вычисления выражений были сформулированы выше.

```
++i;
```

– этот оператор представляет выражение, которое увеличивает значение переменной **i** на единицу.

```
a=cos(b * 5);
```

– этот оператор представляет выражение, включающее в себя операции присваивания и вызова функции.

```
a(x,y);
```

– этот оператор представляет выражение, состоящее из вызова функции.

8 МАССИВЫ И УКАЗАТЕЛИ В СИ

8.1 Одномерные массивы

Массив – группа элементов одного типа, расположенных друг за другом в памяти и имеющих одно общее имя. Определение массива должно включать его имя и содержать информацию о типе и количестве составных элементов. Доступ к элементам массива осуществляется по **имени массива** и **индексу элемента**.

Индекс – порядковый номер элемента в массиве, индекс первого элемента всегда **0**, далее следуют целые положительные числа (**1**, **2** и т.д.).

Массивы задаются так же, как и обычные переменные, но после имени следуют **квадратные скобки**, в которых может быть заключена константа или константное выражение, которое задает количество элементов массива.

```
float array[12];    // Массив array, содержащий 12 элементов типа float
int buffer[5*10];   // Массив buffer, содержащий 50 элементов типа int
```

```
int n = 15;    // Ошибочное объявление, так как переменная
int mas[n];    // не может задавать размер массива
```

Константное выражение, определяющее размер массива, **не может** принимать **нулевое** значение.

Как и обычные переменные, массивы при объявлении могут быть явно проинициализированы. Для этого при объявлении помещается список начальных значений элементов, заключенный в **фигурные скобки**.

```
char array[5] = {'A', 'B', 'C', 'D', 'E'};
```

Здесь объявлен массив из пяти элементов, все элементы этого массива проинициализированы соответствующими значениями. Количество проинициализированных элементов не должно превышать его размерность, иначе будет выдано сообщение об ошибке.

При объявлении массива можно инициализировать не все элементы, а любое количество первых:

```
int array [15] = {3, 6, 7, 12};
```

Все остальные одиннадцать элементов будут проинициализированы **нулями**. Если значений в фигурных скобках будет

больше, чем указано в квадратных скобках, то компилятор выдаст сообщение об ошибке.

Допустимо объявлять массивы только со списком начальных значений. В этом случае число элементов массива компилятор определяет по списку инициализации:

```
int array[] = {34, 78, ,3, 98};
```

Чтобы получить доступ к определенному элементу массива, нужно указать его порядковый номер в массиве, заключенный в квадратные скобки. В остальном работа с массивом осуществляется, как с обычной переменной.

В языке **Си** для повышения производительности системы не выполняются проверки на контроль допустимости значения индекса массива. Для упрощения контроля границ индекса массива можно использовать операцию **sizeof**, которая применительно к массивам возвращает число байтов памяти, зарезервированных компилятором для массива, а применительно к обычным переменным или типам данных – число байт, занимаемых переменной данного типа:

```
int array [10] ={1,2,3,4,5,6,7};
int size;
size = sizeof(array)/sizeof(int);  // size = 10
```

В выражении, задающем индекс элемента массива, можно использовать не только числовые выражения.

```
index = 2;
array[index] = 5;
```

```
array[3] = 4;
buffer[array[3]] = 7;
```

Пример. Дан массив из четырех элементов. Если элемент массива является четным, поменять его знак (листинг 8.1).

Листинг 8.1

```
#include<stdio.h>

void main()
{
   int i, array [4] = {3, 6, 7, 12};
   for (i = 0; i < 4; i++)
      if (array[i] % 2 = 0)
         array [i] *= (-1);
   printf("Результат:\n");
   for (i = 0; i < 4; i++)
      printf("%5d",array[i]);
}
```

8.2 Многомерные массивы

Си поддерживает многомерные массивы. **Многомерный массив** – это **массив массивов**, т.е. массив, элементами которого являются массивы. **Размерность массива** – это количество индексов, используемых для ссылки на конкретный элемент массива. Многомерные массивы объявляются точно так же, как и одномерные, только после имени массива ставится **более одной** пары квадратных скобок.

```
int array[10][30];   // двухмерный массив на 10 строк и
                        30 столбцов
```

Фактически двухмерный массив представляется как одномерный, элементы которого тоже массивы.

Константное выражение, определяющее одну из размерностей массива, **не может** принимать **нулевое** значение.

Можно инициализировать и многомерные массивы. Инициализация происходит построчно, т.е. в порядке возрастания самого правого индекса. Именно в таком порядке элементы многомерных массивов располагаются в памяти компьютера.

Инициализация трехмерного массива с восемью элементами может выглядеть следующим образом:

```
int array[2][2][2] = {23, 54, 16, 43, 82, 12, 9, 75};
```

Проинициализированный массив будет выглядеть так:

```
[0][0][0] = 23
[0][0][1] = 54
[0][1][0] = 16
  ... ... ...
[1][1][0] = 9
[1][1][1] = 75
```

Для наглядности при инициализации двухмерного массива список начальных значений следует оформлять в виде таблицы:

```
int array[3][3] = {  34,    23,   67,
                     16,    43,   82,
                     12,    9,    75  };
```

Многомерные массивы могут инициализироваться и без указания одной (самой левой) из размерностей массива. Компилятор в этом случае определяет количество элементов по количеству членов в списке инициализации.

```
int array[][3] = {   34,   23,   67,
                     16,   43,   82,
                     12,   9,    75  };
```

Если необходимо проинициализировать не все элементы строки, а только несколько первых элементов, в списке инициализации можно использовать фигурные скобки, охватывающие значения для строки. Т.е.:

```
int array[][3] = {   { 0 },
                     { 10, 11 },
                     { 20, 21, 22} };
```

Работа с многомерными массивами не отличается от работы с одномерными, для доступа к элементу многомерного массива необходимо указать все его индексы:

```
tmp = array[1][2];
```

Задать элементы двухмерного массива с клавиатуры можно следующим образом:

```
int i, j, mas[3][5];
for (i = 0; i < 3; i ++)
   for (j = 0; j < 5; j ++)
      scanf("%d", &mas[i][j]);
```

Си позволяет описывать и массивы с размерностью больше чем **3** или **5**, хотя задачи, требующие использования таких размерностей, встречаются редко.

8.3 Указатели

Указатель – особый вид переменной, которая хранит адрес элемента в памяти, где может быть записано **значение другой переменной**.

Определение указателя:

```
тип *имя_указателя;
```

где * – звездочка, определяющая тип «**указатель**».

Например:

```
int variable, *point; // Переменная целого типа variable
                      // и указатель на целый тип *point
```

Существует операция, неразрывно связанная с указателями. Это унарная операция **взятие адреса**: **&**.

Операции с указателями:

1)операция доступа по указателю

Операция **доступа по указателю *E**, где **E** – переменная типа «указатель», – операция **разадресации**. Результат – содержимое ячейки памяти, на которую указывает **E**. С переменной ***E** можно работать как с обычной переменной.

```
...
int *p;      // Объявление переменной типа указатель на
int
int a;       // Объявление переменной a
a = 18;
p = &a; //Присвоение адреса переменной a переменной p
*p += 8;  // Значение переменной a после выполнения
          // этого оператора равно 26
...
```

2)операция присваивания

Операция **присваивания** для указателей аналогична соответствующей операции для других типов данных. Необходимо применять операцию **приведения** типа, если используются **указатели** на **разные типы** данных;

3)операция увеличения/уменьшения указателя

```
int i, *p;
p + i // указатель на адрес i-го элемента после данного
p - i  // указатель на адрес i-го элемента перед данным
```

4)операция сложного присваивания

```
p += i;    p -= i;
```

аналогичны выражениям:

```
p = p + i;    p = p - i;
```

5)операции инкремента/декремента

```
p++;    ++p;    p--;    --p;
```

Выполнение данных операций аналогично соответствующим операциям над целочисленными типами, т.е. указатель будет смещаться (увеличиваться или уменьшаться в зависимости от операции) на один элемент, фактически указатель (адрес) изменить на количество байтов, занимаемых этим элементом в памяти;

6)операция индексирования

p[i]

Эта операция полностью аналогична ***(E+i)**, т.е. из памяти извлекается и используется значение **i**-го элемента массива, адрес которого присвоен указателю **E**;

7)вычитание указателей

E1-E2

где **E1**, **E2** – переменные типа «указатель», причем указатели на один и тот же набор данных и одного типа, **иначе** операция **бессмысленна**. Результат имеет тип **int** и равен количеству элементов, которые можно расположить в ячейках памяти с **E2** по **E1**;

8)операции отношений

Для указателей определяются операции отношения:

E1 == E2; E1 >= E2; E1 > E2; E1 <= E2; E1 < E2; E1 != E2;

Результат всех операций имеет тип **int**. Кроме того, любой указатель может быть проверен на равенство (==) или неравенство (**!=**) со специальным значением **NULL** (**NULL = 0**). Функции выделения памяти возвращают **NULL** при появлении каких-либо ошибок. Поэтому сравнение с **NULL** часто используется для определения ошибки выделения памяти.

8.4 Строки

Разновидностью массива является так называемый **строковый литерал** (или просто строка) – последовательность любых символов, заключенных в парные двойные кавычки. В **Си**, в отличие от многих других языков программирования, отсутствует специальный строковый тип. Вместо этого строковый литерал представляется как массив элементов типа **char**, в конце которого помещен символ **'\0'** (**нуль-терминатор**). Такой массив называют строкой в формате **ASCIIZ** или просто **ASCIIZ-строкой**.

При объявлении массива под строку необходимо предусмотреть место для нуль-терминатора, которым должна заканчиваться любая строка в **Си**. Поскольку строка – это массив, то его можно проинициализировать. Вот три совершенно идентичных объявления:

```
char str[7] = {'с','т','р','о','к','а','\0'};
```

или

```
char str[7] = "строка";
```

или

```
char str[] = "строка";
```

В первом операторе седьмым символом добавлен **'\0'** в конце строки, а в последних двух его нет, так как, если текст помещается в двойные кавычки, то **Си** рассматривает его как строку, а, значит, компилятор автоматически добавляет в конец строки нуль-терминатор.

Для работы со строками обычно используют указатели типа **char ***. Если строковая константа используется для инициализации указателя типа **char ***, то адрес **первого** символа переменной будет **начальным значением** указателя. Например:

```
char *str="строка";
```

Здесь описывается только указатель **str**, и указатель получает начальное значение, равное адресу первого элемента (символа «**с**») строковой константы. Компилятор выделит память как для строки, так и для размещения значения указателя.

Если строковая константа используется в выражении в тех местах, где разрешается применять указатель, компилятор подставляет в выражение вместо константы адрес первого его символа. Например:

```
char *str;
str = "строка";
```

При выполнении операции присваивания, в ячейку памяти, отведенную для указателя **str**, пересылается не массив символов, а только указатель на его начало.

Строку можно ввести с помощью функции **scanf**, используя указатель:

```
char *str;
scanf("%s", str);
```

либо

```
scanf("%s", &str[0]);
```

Преимущества использования указателей в том, что можно оперировать не самими объектами, а только их адресами, что дает значительный выигрыш в скорости выполнения программы.

Наиболее типичные операции над строковыми данными оформлены в виде функций стандартной библиотеки работы со строками, подключаемой к программе с помощью файла-заголовка «**string.h**».

9 АЛГОРИТМЫ ПОИСКА И СОРТИРОВКИ

9.1 Организация последовательного поиска

Одно из наиболее часто встречающихся в программировании действий – поиск. Последовательный поиск используется в целях нахождения некоторого значения в списке. **Последовательный поиск** применим для любого списка.

Код для алгоритма последовательного поиска представлен в листинге 9.1.

Листинг 9.1

```
// Поиск в массиве list из n элементов для нахождения
// соответствия с ключом key
// Выводит на экран индекс соответствующего элемента
// массива и его значение или сообщение о том,
// что значение не найдено
#include<stdio.h>
#include<stdlib.h>

void main()
{
   int list[50], n, key, i, rez=-1;
   randomize();
   printf("Введите размерность массива (<=50): ");
   scanf("%d",&n);
   for (i = 0; i < n; i ++)
   { // заполняем массив случайными значениями
     // от -50 до 50
      list[i]=random(101)-50;
      printf("%5d",list[i]);
   }
   printf("\nВведите значение, которое будем искать: ");
   scanf("%d",&key);
   for (i = 0; i < n; i++)
   {
      if (list [i] == key)
      {
         rez=i;
         // rez хранит индекс соответствующего элемента
         printf("list[%d] = %d\n",rez,key);
      }
   }
   if (rez == -1)
      // если поиск неудачный, то rez остается равным -1
      printf("Такой элемент не найден!\n");
}
```

При определении эффективности алгоритма последовательного поиска различают поведение наилучшего и наихудшего случаев. Наилучшему случаю соответствует нахождение ключа в первом элементе. Наихудший случай имеет место, когда ключ не находится в списке или обнаруживается в конце списка, он требует проверки всех **n** элементов.

9.2 Организация бинарного поиска

Если список является упорядоченным, то **бинарный поиск** представляет улучшенный метод поиска.

Модель такого алгоритма – нахождение номера в телефонном справочнике. Зная нужные имя и фамилию, вы открываете справочник ближе к началу, середине или концу, в зависимости от первой буквы фамилии. Вам может повезти, и вы сразу попадете на нужную страницу. В противном случае вы переходите к более ранней или более поздней странице в справочнике в зависимости от относительного месторасположения имени человека по алфавиту. Процесс продолжается до тех пор, пока вы не найдете соответствие или не обнаружите, что этого имени нет в справочнике.

Предположим, что список упорядочен как массив. Индексами в концах списка являются: **low = 0** и **high = n-1**, где **n** – это количество элементов в массиве; **low**, **high** и **mid** – значения целого типа (рисунок 9.1).

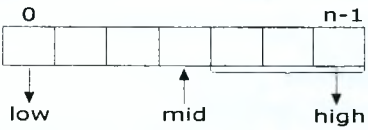


Рисунок 9.1 – Организация бинарного поиска

Алгоритм выполнения бинарного поиска:

1) находим середину списка: **mid = (low + high) / 2**;

2) сравниваем значение в серединном элементе с ключом **key**; если совпадение найдено, то возвращаем индекс **mid** для нахождения ключа.

```
if (list[mid] == key)
   printf("list[%d] = %d\n", mid+1, key);
```

3) определение новых границ:

```
if (list[mid] > key)
   high = mid – 1;

if (list[mid] < key)
   low = mid + 1;
```

4) переход к шагу **1**, пока не будет найден ключ либо не будет пройден весь список.

Код для алгоритма бинарного поиска представлен в листинге 9.2.

Листинг 9.2

```
// Поиск в отсортированном массиве list из n элементов
// для нахождения соответствия с ключом key
// Выводит на экран порядковый номер соответствующего
// элемента массива и его значение или сообщение о том,
// что нет соответствия
#include<stdio.h>
#include<stdlib.h>

void main()
{
   int list[50], n, key, i, low, high, mid;
   randomize();
   printf("Введите размерность массива (<=50): ");
   scanf("%d",&n);
   for (i = 0; i < n; i ++)
   {
      list[i]=i*2-1;
      // заполняем массив значениями функции list_i=2i-1
      printf("%5d",list[i]);
   }
   printf("\nВведите значение, которое будем искать: ");
   scanf("%d",&key);
   i=0;
   low=0;
   high=n-1;
   while (low<=high)
   {
      mid=(low+high)/2;
      if (list [mid] == key)
      {
         printf("list[%d] = %d\n",mid+1,key);
         return;
      }
      if (list[mid]>key)
         high=mid-1;
      if (list[mid]<key)
         low=mid+1;
   }
   printf("Такой элемент не найден!\n");
}
```

Алгоритм уточняет местоположение совпадающего с ключом элемента, деля пополам длину интервала, в котором может находиться этот элемент, и затем выполняет тот же алгоритм поиска в меньшем подсписке. В конце концов, если искомый элемент не находится в списке, **low** превысит **high**, алгоритм выводит на экран сообщение о том, что совпадение не произошло. Если ключ был найден, то алгоритм выводит на экран порядковый номер элемента и его значение и завершает свое выполнение.

Наилучший случай имеет место, когда совпадающий с ключом элемент находится в середине списка.

9.3 Основные алгоритмы сортировки массивов

Сортировка – процесс перегруппировки заданного множества в некотором определенном порядке.

Цель сортировки – облегчить последующий поиск элементов в таком отсортированном множестве. Мы встречаемся с отсортированными объектами в телефонных книгах, в библиотеках, в словарях – почти везде, где нужно искать хранимые объекты.

Основное условие – выбранный метод сортировки массивов должен экономно использовать доступную память. Это предполагает, что перестановки должны выполняться на том же месте, т.е. методы, в которых элементы из массива **A** передаются в результирующий массив **B**, представляют существенно меньший интерес.

Будем классифицировать методы сортировки по их экономичности, т.е. по времени их работы.

9.3.1 Сортировка методом пузырька

Для сортировки массива **A** из **n** элементов требуется до **n-1** проходов (прохождение всего массива один раз от первого до последнего элемента массива). В каждом проходе сравниваются соседние элементы, и если первый из них больше второго, они меняются местами. К моменту окончания каждого прохода наименьший элемент поднимается к вершине текущего подмассива. Код алгоритма сортировки методом пузырька представлен в листинге 9.3.

Листинг 9.3

```
#include<stdio.h>
#include<stdlib.h>
void main()
```

```
{
   int A[50], n, aux, i, j;
   randomize();
   printf("\nВведите размерность массива (<=50): ");
   scanf("%d",&n);
   for (i = 0; i < n; i++)
   {
      // запоняем массив случайными значениями от -50 до 50
      A[i]=random(101)-50;
      printf("%5d",A[i]);
   }
   for (i = 0; i < (n-1); i++)
      for (j = 0; j < (n-1); j++)
      {
         if (A[j] > A[j+1])
         {
            aux = A[j];
            A[j] = A[j+1];
            A[j+1] = aux;
         }
      }
   printf("\nОтсортированный массив:\n");
   for (i = 0; i < n; i++)
      printf("%5d", A[i]);
}
```

При пузырьковой сортировке сохраняется индекс последнего обмена во избежание избыточного просмотра. Это придает алгоритму заметную эффективность в некоторых особых случаях.

В общем случае число сравнений при сортировке пузырьком составляет $(n^2-n)/2$, а максимальное число перестановок – $3\cdot(n^2-n)/4$.

9.3.2 Шейкерная сортировка

Улучшение алгоритма сортировки методом пузырька:

1) если в последнем проходе перестановок не было, то алгоритм можно заканчивать (т.е., если последние элементы последовательности оказались упорядоченными);

2) ясно, что все пары соседних элементов ниже индекса **i** уже находятся в желаемом порядке, поэтому просмотры можно начинать именно с этого индекса, а не с **1**-го элемента;

3) чередовать направление последовательности просмотров.

Получающийся при этом алгоритм называется **шейкерной** сортировкой (от слова «шейкер» – смеситель).

Пример программы шейкерной сортировки представлен в листинге 9.4.

Листинг 9.4

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

void main()
{
   int A[50], k, i, L, R, x, n;
   randomize();
   printf("\nВведите размерность массива (<=50): ");
   scanf("%d",&n);
   for (i = 0; i < n; i++)
   {
      // заполняем массив случайными значениями
      // от -50 до 50
      A[i]=random(101)-50;
      printf("%5d",A[i]);
   }
   L=0;
   R=n-1;
   k=n-1;
   while (L<R) {
      for(j=L;j<R;i++)
         if (A[i]>A[i+1])
         {
            x=A[i];
            A[i]=A[i+1];
            A[i+1]=x;
            k=i;
         }
      R=k;
      for(i=R;i>L;i--)
         if (A[i-1]>A[i])
         {
            x=A[i-1];
            A[i-1]=A[i];
            A[i]=x;
            k=i;
         }
      L=k;
   }
   printf("\nОтсортированный массив:\n");
   for(i=0;i<n;i++)
     printf("%5d",A[i]);
}
```

Для шейкерной сортировки число сравнений составляет **(n-1)**, а максимальное число перестановок – $\mathbf{1/2 \cdot (n^2 - n \cdot (k + \ln n))}$.

9.3.3 Сортировка методом выбора

Сортировка выбором моделирует наш повседневный опыт, когда воспитатели в детском саду пытаются выстроить детей по росту.

Для компьютерного алгоритма предполагается, что **n** элементов данных хранятся в массиве **A**, и по этому массиву выполняется **n-1** проход. В нулевом проходе выбирается наименьший элемент, который затем меняется местами с **A[0]**. После этого неупорядоченными остаются элементы **A[1] ... A[n-1]**. В следующем проходе просматривается неупорядоченная хвостовая часть списка, откуда выбирается наименьший элемент и запоминается в **A[1]**. В следующем проходе производится поиск наименьшего элемента в подмассиве **A[2] ... A[n-1]**. Найденное значение меняется местами с **A[2]**. Таким образом, выполняется **n-1** проход, после чего неупорядоченный хвост сокращается до одного элемента, который и является наибольшим. Код программы для сортировки массива методом выбора представлен в листинге 9.5.

Листинг 9.5

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

void main()
{
   int A[50], k, i, j, n, min;
   randomize();
   printf("\nВведите размерность массива (<=50): ");
   scanf("%d",&n);
   for (i = 0; i < n; i++)
   {
      // запоняем массив случайными значениями от -50 до 50
      A[i]=random(101)-50;
      printf("%5d",A[i]);
   }
   for (i = 0; i < (n-1); i++)
   {
      min = A[i];
      k = i;
      for (j = i+1; j < n; j++)
     {
         if (A[j] < min)
         {
            k = j;
            min = A[j];
         }
     }
```

```
            A[k] = A[i];
            A[i] = min;
        }
        printf("\nОтсортированный массив:\n");
        for (i = 0; i < n; i++)
            printf("%5d", A[i]);
    }
```

Сортировка посредством выбора требует фиксированного числа сравнений, зависящего только от размера массива, а не от начального распределения в нем данных. Общее число сравнений составляет $(n^2-n)/2$. Максимальное число перестановок – $n^2/4+3\cdot(n-1)$. Сложность алгоритма измеряется числом сравнений. Наилучшего и наихудшего случаев не существует, так как алгоритм делает фиксированное число проходов, в каждом из которых сканирует определенное число элементов.

9.3.4 Сортировка вставками

Сортировка вставками похожа на процесс тасования карточек с именами. Регистратор заносит каждое имя на карточку, а затем упорядочивает карточки по алфавиту, вставляя карточку в верхнюю часть стопки в подходящее место (рисунок 9.2).

Алгоритм этой сортировки таков:

1) просматривается вся последовательность, начиная со второго элемента, т.е. индекс его **i=1**;

2) извлекается **i**-й элемент, запоминается;

3) в образовавшееся пустое место сдвигается слева направо последовательность до тех пор, пока извлеченный элемент меньше сдвигаемого;

4) в пустое место ставится извлеченный элемент.

Для сортировки вставками число сравнений составляет $(n^2-n)/4$. В отличие от других методов, сортировка вставками не использует обмены. Сложность алгоритма измеряется числом сравнений.

Недостатком этого метода является то, что включение одного элемента с последующим сдвигом на одну позицию целой группы элементов не экономно!

Для сортировки массива **A** из **n** элементов также требуется до **(n-1)** проходов. Код программы для сортировки массива методом вставки представлен в листинге 9.6.

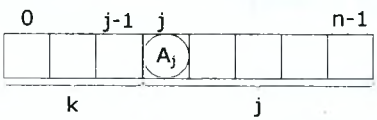


Рисунок 9.2 – Сортировка вставками

Листинг 9.6

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

void main()
{
   int A[50], k, i, j, n, m, Aj;
   randomize();
   printf("\nВведите размерность массива (<=50): ");
   scanf("%d",&n);
   for (i = 0; i < n; i++)
   {
      // запоняем массив случайными значениями от -50 до 50
      A[i]=random(101)-50;
      printf("%5d",A[i]);
   }
   for (i = 1; i < n; i++)
   {
      Ai=A[i];
      k=i-1;
      while (k >= 0)
     {
         if (Ai < A[k])
         {
            A[k+1] = A[k];
            k--;
         }
         else break;
      }
      A[k+1] = Ai;
   }
   printf("\nОтсортированный массив:\n");
   for (i = 0; i < n; i++)
      printf("%5d", A[i]);
}
```

9.3.5 Быстрая сортировка

Общая схема быстрой сортировки:

1) есть некий целочисленный массив **A** из нескольких элементов, и из массива выбирается некий опорный элемент **p**;

2) запускается процедура разделения массива, которая перемещает все элементы, которые меньше либо равны **p**, влево от него, а большие или равные – вправо;

3) таким образом, мы получили новый массив из двух подмассивов, причём левый меньше либо равен правому;

4) для обоих подмассивов, если в подмассиве более двух элементов, рекурсивно запускаем ту же процедуру.

В конце получается полностью отсортированная последовательность.

Достоинства этого метода:

1) самый быстродействующий из всех существующих алгоритмов обменной сортировки;

2) простая реализация;

3) хорошо работает на почти отсортированных данных;

4) хорошо сочетается с алгоритмами кэширования и подкачки памяти.

Два недостатка:

1) требует много дополнительной памяти;

2) алгоритм является неустойчивым.

Функция **qsort()** объявлена в библиотеке «**stdlib.h**».

```
qsort(a, n, sizeof(int), cmp);
```

Функцией **qsort()** можно упорядочивать объекты любой природы. По сути, она предназначена упорядочивать множества блоков байтов равной длины.

Первый аргумент – это адрес, где находится начало первого блока (предполагается, что блоки в памяти расположены друг за другом подряд). **Второй** аргумент функции – это число таких блоков, **третий** аргумент – длина каждого блока. **Четвёртый** аргумент функции **qsort()** – это имя функции, которая умеет сравнивать два элемента массива.

```
int cmp(const void *a, const void *b)
{
   return *(int*)a - *(int*)b;
}
```

В силу указанной универсальности функции сортировки, функция сравнения получает в качества аргумента адреса двух блоков, которые нужно сравнить и возвращает **1**, **0** или **-1**:

- **положительное** значение, если **a > b**;
- **0**, если **a = b**;
- **отрицательное** значение, если **a < b**.

Поскольку у нас блоки байт – это целые числа (в **32**-битной архитектуре это четырёхбайтовые блоки), то необходимо привести данные указатели типа (**const void***) к типу (**int ***), и осуществляется это с помощью дописывания перед указателем выражения «**const int***». Затем нужно получить значение переменной типа **int**, которая лежит по этому адресу. Это делается с помощью дописывания спереди звездочки.

Пример. Сортировка целочисленного массива (листинг 9.7).

Листинг 9.7

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

# define N 1000

int cmp (const void *a, const void *b)
{
   return (*(int*)a – *(int*)b);
}

void main ()
{
   int n, i, A[N];
   randomize();
   printf("\nВведите размерность массива (<=1000): ");
   scanf("%d", &n);
   for (i = 0; i < n; i ++)
   {
      // запоняем массив случайными значениями от -50 до 50
      A[i]=random(101)-50;
      printf("%5d",A[i]);
   }
   qsort (A, n, sizeof(int), cmp);
   printf("\nОтсортированный массив:\n");
   for (i = 0; i < n; i ++)
      printf("%5d", A[i]);
}
```

Пример. Программа упорядочивания строк в алфавитном порядке (листинг 9.8).

Листинг 9.8

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

# define N 100
# define M 30

void main ()
{
   int n, i;
   char A[N][M];
```

```
    printf("\nВведите размерность массива (<=100): ");
    scanf("%d", &n);
    for (i = 0; i < n; i ++)
       scanf("%s", &A[i]);
    qsort (A, n, sizeof(char[M]), (int (*)(const void*,
const void*)) strcmp);
    printf("\nУпорядоченный массив:\n");
    for (i = 0; i < n; i ++)
       printf("%s\n", A[i]);
}
```

Функция **strcmp()**, объявленная в файле «**string.h**», имеет следующий прототип:

```
int strcmp(const char*, const char*);
```

Т.о., функция получает два аргумента – указатели на кусочки памяти, где хранятся элементы типа **char**, то есть два массива символов, которые могут быть изменены внутри функции **strcmp** (запрет на изменение задается с помощью модификатора **const**).

В то же время в качестве четвертого элемента функция **qsort** хотела бы иметь функцию типа

```
int cmp(const void*, const void*);
```

В языке **Си** можно осуществлять приведение типов, являющихся типами функции. В данном примере тип

```
int (*)(const char*, const char*);
// функция, получающая два элемента
// типа «const char *» и возвращающая элемент типа int,
```

приводится к типу

```
int (*)(const void*, const void*);
// функция, получающая два элемента
// типа «const void *» и возвращающая элемент типа int
```

Функция **strcmp** в соответствии с описанием осуществляет сравнение двух строк и определяет, какая из двух строк идет первой в алфавитном порядке (сравнивает две строки в лексикографическом порядке), а именно, она возвращает **1**, если первая строка больше второй (идет после второй в алфавитном порядке), **0** – если они совпадают, и **-1** – если первая меньше второй.

10 ФУНКЦИИ ПОЛЬЗОВАТЕЛЯ В СИ

Программы на **Си** обычно состоят из множества подпрограмм, называемых **функциями**. **Функции** имеют небольшой размер и могут располагаться в разных файлах. Независимо от расположения, все функции являются **глобальными**. В языке **Си** запрещено определять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные.

В любой программе на **Си** присутствует как минимум одна функция – функция **main()**, с которой начинается выполнение программы. Аналогичным образом могут быть описаны любые другие пользовательские функции.

Причины использования функций:

• уменьшение дублирования кода;

• возможность создания более структурированной и понятной пограммы.

10.1 Функция пользователя и ее прототип

В общем случае функции в языке **Си** необходимо объявлять в самом начале программы. Такое объявление выглядит как отдельно описанный заголовок, в конце которого стоит знак «**;**». Такое объявление называется **прототипом функции**. Прототип должен предшествовать любому использованию функции в программе, а полное описание функции может быть помещено как после тела программы, так и до него.

Прототип функции выглядит следующим образом:

тип имя_функции (тип имя_арг1, тип имя_арг2, ...);

Тип перед именем функции определяет тип значения, которое возвращает функция. Если тип не указан, то предполагается, что функция возвращает целое значение **int**. В прототипе функции для каждого ее параметра можно указать только его тип, а можно дать и его имя. В полном описании функции имена параметров необходимы для того, чтобы функция могла к ним обращаться.

int function (int, int); *// прототип*

или **int function (int a, int b);**

или **int function (int a, b, c, ...);**

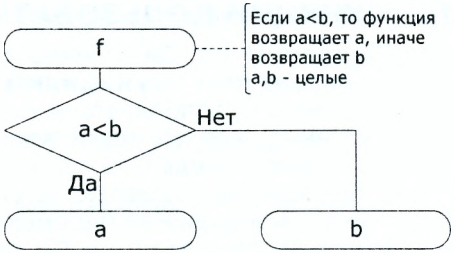
Передача значения из вызванной функции в вызвавшую происходит с помощью оператора возврата **return** в следующей форме:

return значение**;**

Таких операторов в функции может быть несколько, и тогда они фиксируют соответствующие точки выхода из функции (листинг 10.1).

Листинг 10.1

```
int f(int a, int b)
{
   if a < b
      return a;
    else
      return b;
}
```



Вызвать пользовательскую функцию можно теми же способами, что и стандартную:

```
c = f(15, 5);
c = f(d, g);
f(d, g);
```

Вызвавшая функция при необходимости может игнорировать возвращаемое значение. После слова **return** можно ничего не записывать. В этом случае вызвавшей функции никакого значения не передается. Управление передается вызвавшей функции и в случае окончания тела функции (последняя закрывающая фигурная скобка тела функции).

В языке **Си** аргументы функции передаются по значению, т.е. вызванная функция получает временную копию каждого аргумента, а не его адрес. Это означает, что вызванная функция не может изменить значение переменной вызвавшей ее программы. Однако это сделать легко, если передавать в функцию не переменные, а их адреса (листинг 10.2).

Листинг 10.2

```
void swap(int *a, int *b);
{
   int *tmp;
   *tmp = *a;
   *a = *b;
   *b = *tmp;
}
```

swap(&b, &c) – программе передаются адреса переменных **b** и **c**. Вызов ее приведет к тому, что значения **b** и **c** поменяются местами.

По определению функция может вернуть только одно значение-результат. Если надо вернуть два и больше результатов, приходится использовать специальный прием – передачу параметров по ссылке.

Пример. Написать функцию, которая определяет максимальное и минимальное из двух целых чисел.

В следующей программе используется достаточно хитрый прием: мы сделаем так, чтобы функция изменяла значение пе-

ременной, которая принадлежит основной программе. Один результат (минимальное из двух чисел) функция вернет как обычно, а второй – за счет изменения переменной, которая передана из основной программы (листинг 10.3).

Листинг 10.3

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

int MinMax(int a, int b, int *Max);

void main()
{
   int N, M, min, max;
   printf("\nВведите 2 целых числа:");
   scanf("%d%d", &N, &M);
   min=MinMax(N, M, &max); // вызов функции
   printf("Наименьшее из них %d, наибольшее –
%d\n", min, max);
   getch();
}

int MinMax(int a, int b, int *Max)
{
   if (a>b)
   {
      *Max=a;
      return b;
   }
   else
   {
      *Max=b;
      return a;
   }
}
```

Т.о., если надо, чтобы функция вернула два и более результатов, то поступают следующим образом: один результат передается как обычно с помощью оператора **return**, а остальные возвращаемые значения передаются через изменяемые параметры.

Если в качестве аргумента функции используется имя массива, то передается только адрес начала массива, а сами элементы не копируются. Это происходит потому, что имя массива само по себе является указателем.

10.2 Функции с переменным числом аргументов

В языке **Си** разрешается создавать **функции с переменным числом аргументов**. Тогда при задании прототипа вместо последнего аргумента указывается **троеточие**. Например, так объявлены библиотечные функции **scanf()** и **printf()**. Доступ к

дополнительным параметрам при таком объявлении функции осуществляется средствами адресной арифметики, причем программист несет ответственность за правильное определение количества параметров и их типов.

10.3 Командная строка. Параметры функции main()

Разработчики языка **Си** предусмотрели возможность передачи **аргументов головному** модулю запущенной на выполнение программы – функции **main()** – посредством использования **командной** строки.

Аргументы командной строки – это **текст**, записанный после имени запускаемого на выполнение файла ***.com** или ***.exe**, либо передаваемый программе с помощью опции интегрированной среды **Си** – **arguments**.

В качестве аргументов целесообразно передавать имена файлов, функций, текст, задающий режим работы программы, а также сами данные (числа). Т.о., число аргументов командной строки не известно заранее. Теоретически в этом случае можно было бы применить объявление функции с троеточием в списке аргументов (как в п. 10.2), однако на практике для функции **main()** использован другой метод.

Turbo C поддерживает **три** параметра функции **main()**. Для их обозначения рекомендуется использовать общепринятые имена **argc**, **argv**, **evp** (но не запрещается использовать любые другие имена). Вход в функцию **main()** при использовании командной строки имеет вид:

```
int main(int argc, char *argv[], char *envp[])
{ ... }
```

Либо:

```
int main(int argc, char **argv, char **envp)
{ ... }
```

Первый параметр (**argc**) сообщает функции количество передаваемых в командной строке аргументов, учитывая в качестве первого аргумента имя самой выполняемой программы (т.е. количество слов, разделенных пробелами). Отсюда следует, что количество параметров **не может быть меньше единицы**, так как первый аргумент – имя программы с полным путем к ней присутствует всегда.

Второй параметр (**char** ***argv**, **char** ***argv[]**) является указателем на массив из указателей на слова (т.е. сами параметры) из командной строки. Каждый параметр хранится в виде **ASCIIZ**-строки. Под словом понимается любой текст, не содержащий символов пробела или табуляций. Аргументы должны разделяться пробелами или знаками табуляции. Запятые, точки и прочие символы не рассматриваются как разделители. По-

следним элементом массива указателей является нулевой указатель (**NULL**).

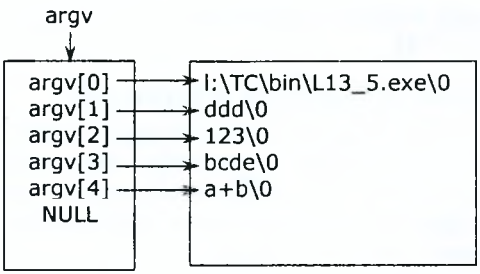
Например, пусть программа **L13_15.exe** запускается следующим образом:

I:\TC\bin\L13_5.exe ddd 123 bcde a+b

и заголовок функции **main** имеет вид:

```
void main (int argc,   char   *argv[])
{ ... }
```

тогда **argc=5** и создается массив из пяти указателей, каждый из которых указывает на отдельное слово (аргумент).



Если необходимо в качестве аргумента передавать строку, содержащую пробелы или символы табуляции, то ее необходимо заключить в двойные кавычки.

Например:

I:\TC\bin\L13_5.exe "Иванов Виктор" Simps

Третий параметр функции **main** (**char** ****envp) служит для передачи в программу информации о системном окружении операционной системы, т.е. о среде, в которой выполняется программа. Он является указателем на массив указателей, каждый указатель определяет адрес, по которому записана строка информации о среде, определяемая операционной системой. Признаком конца массива (как и в **char** ***argv[]**) является нулевой указатель. Среда, в которой выполняется программа, определяет некоторые особенности поведения оболочки и ядра операционной системы.

11 РЕКУРСИЯ

11.1 Рекурсивные объекты

Рекурсия – это определение объекта через самого себя.

С помощью рекурсии в математике определяются многие бесконечные множества, например, множество натуральных чисел.

Натуральное число:

1) **1** – натуральное число.

2) Число, следующее за натуральным, – натуральное.

Понятие факториала **n!=1·2·3·...·(n-1)·n** также можно задать рекурсивно:

1) **0! = 1** (так условно принято).

2) **n! = n*(n-1)!**

11.2 Рекурсивные процедуры и функции

Процедуры и функции в современных языках программирования также могут вызывать сами себя. Рекурсивными называются процедуры и функции, которые вызывают сами себя.

Например, функцию вычисления факториала можно записать, как показано в листинге 11.1.

Листинг 11.1

```
int Factorial ( int n )
{
   if ( n <= 0 )
      return 1; // вернуть 1
   else
      return n*Factorial(n-1); // рекурсивный вызов
}
```

Обратите внимание, что функция **Factorial** вызывает сама себя, если **n > 0**. Для решения этой задачи можно использовать и рекурсивную процедуру (а не функцию). Рекурсивная процедура может вернуть значение-результат через параметр, переданный по ссылке (в объявлении процедуры у его имени стоит знак ссылки **&**). При рекурсивных вызовах процедура меняет это значение (листинг 11.2).

Листинг 11.2

```
void Factorial ( int n, int &fact )
{
   if ( n == 0 )
      fact = 1; // рекурсия закончилась
   else
```

```
    {
        Factorial(n-1, fact);
        // рекурсивный вызов, считаем (n-1)!
        fact *= n; // n! = n*(n-1)!
    }
}
```

В отличие от функции, процедура может (если надо) возвращать несколько значений результатов с помощью параметров, передаваемых по ссылке.

11.3 Косвенная рекурсия

Реже используется более сложная конструкция, когда процедура вызывает саму себя не напрямую, а через другую процедуру (или функцию). Такой прием называется косвенная рекурсия. Это описывается схемой, изображенной на рис. 11.1.

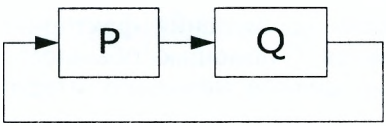


Рисунок 11.1 – Косвенная рекурсия

11.4 Бесконечная рекурсия

При использовании рекурсивных процедур и функций велика опасность, что вложенные вызовы будут продолжаться бесконечно (это похоже на зацикливание цикла **while**). Поэтому в таких функциях необходимо предусмотреть условие, которое проверяется на каждом шаге и заканчивает вложенные вызовы, когда перестает выполняться.

Для функции вычисления факториала таким условием является **n<=0**. Докажем, что рекурсия в примере, рассмотренном выше, закончится.

1. Рекурсивные вызовы заканчиваются, когда **n** становится равно **нулю**.

2. При каждом новом рекурсивном вызове значение **n** уменьшается на **1** (это следует из того, что вызывается **Factorial(n-1,...)**).

3. Поэтому, если в начале **n>0**, то, постепенно уменьшаясь, **n** достигает значения **0** и рекурсия заканчивается.

Рекурсивная процедура или функция должна содержать условие, при котором рекурсия заканчивается (не происходит нового вложенного вызова).

11.5 Когда рекурсия не нужна

При новом рекурсивном вызове компьютер делает так:

1. Запоминается состояние вычислений на данном этапе.

2. В стеке (особой области памяти) создается новый набор локальных переменных (чтобы не испортить переменные текущего вызова).

Поскольку при каждом вызове затрачивается новая память и расходуется время на вызов процедуры и возврат из нее, при использовании рекурсии необходимо помнить, что глубина рекурсии (количество вложенных вызовов) должна была достаточно мала.

Программа, использующая рекурсию с большой глубиной, будет выполняться долго и может вызвать переполнение стека (нехватку стековой памяти). Поэтому если задача может быть легко решена без использования рекурсии, рекурсию использовать нежелательно.

Например, задача вычисления факториала (листинг 11.3) очень просто решается с помощью обычного цикла **for** (такое решение с помощью циклов называют итеративным, циклическим).

Листинг 11.3

```
int Factorial ( int n )
{
   int i, fact = 1;
   for (i = 2; i <= n; i++)
      fact *= i;
   return fact;
}
```

Эта функция работает намного быстрее, чем рекурсивная.

Доказано, что любая рекурсивная программа может быть написана без использования рекурсии, хотя такая реализация может оказаться очень сложной.

Пример. Составить функцию для вычисления чисел Фибоначчи f_i, которые задаются так (листинг 11.4):

1. $f_0 = 0$, $f_1 = 1$.

2. $f_i = f_{i-1} + f_{i-2}$ для $i > 1$.

Использование рекурсии «в лоб» дает функцию:

Листинг 11.4

```
int Fib ( int n )
{
   if ( n == 0 )
      return 0;
```

```
    if ( n == 1 )
       return 1;
    return Fib(n-1) + Fib(n-2);
}
```

Заметим, что каждый рекурсивный вызов при **n>1** порождает еще **2** вызова функции, многие выражения (числа Фибоначчи для малых **n**) вычисляются много раз. Поэтому практического значения этот алгоритм не имеет, особенно для больших **n**. Схема вычисления **Fib(5)** показана в виде дерева на рис. 11.2.

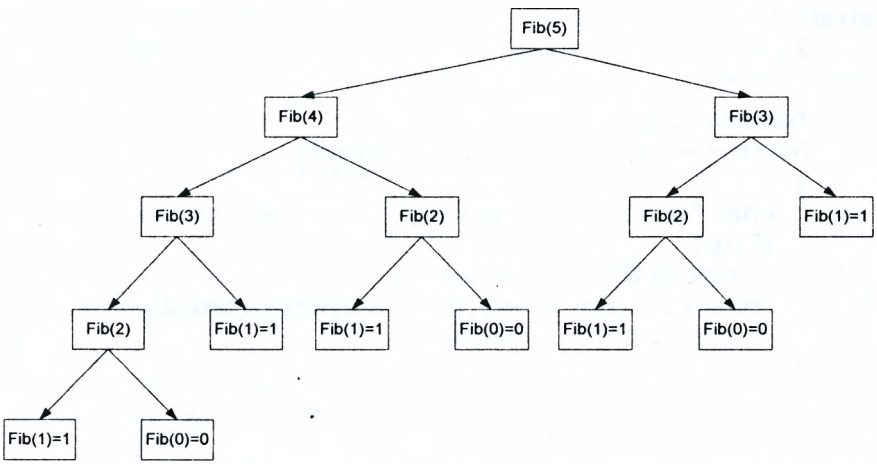


Рисунок 11.2 – Схема вычисления чисел Фибоначчи

Заметим, что очередное число Фибоначчи зависит только от двух предыдущих, которые будем хранить в переменных **f1** и **f2**. Сначала примем **f1=1** и **f2=0**, затем вычисляем следующее число Фибоначчи и записываем его в переменную **x**. Теперь значение **f2** уже не нужно и мы скопируем **f1** в **f2** и **x** в **f1** (листинг 11.5).

Листинг 11.5

```
int Fib2(int n)
{
   int i, f1 = 1, f2 = 0, x;
   for (i = 2; i <= n; i++)
   {
      x = f1 + f2; // следующее число
      f2 = f1; f1 = x; // сдвиг значений
   }
   return x;
}
```

Такая процедура для больших **n** (>**20**) работает в сотни тысяч раз быстрее, чем рекурсивная. **Вывод**: там, где можно легко обойтись без рекурсии, надо без нее обходиться.

11.6 Рекурсивный поиск

Пример, в котором необходимо было определить, сколько раз встречается заданное слово в предложении, можно реализовать при помощи рекурсии. Рекурсивная процедура будет выглядеть следующим образом (листинг 11.6):

1) ищем первое заданное слово с помощью функции **strstr()** – если не нашли, то стоп;

2) количество слов = **1** + количество слов в оставшейся части строки.

Листинг 11.6

```
int HowMany( char *s, char *word )
{
   char *p = strstr(s, word); // ищем первое слово
   if (!p)
      return 0; // не нашли – 0 слов
   return 1 + HowMany(p+strlen(word),word);
   // одно уже нашли, ищем дальше
}
```

Функция получилась короткая, понятная, но по скорости работы не самая эффективная.

11.7 Перебор вариантов

Одна из главных практически важных областей, где применение рекурсии значительно упрощает решение, – задачи на перебор вариантов.

11.7.1 Сочетания

Необходимо получить все возможные сочетания чисел от **1** до **K**, которые имеют длину **N** (в последовательности могут быть одинаковые элементы). Для решения задачи выделим в памяти массив **A[N]**. Представим себе, что все его первые **q** элементов (с номерами от **0** до **q-1**) уже определены, и надо найти все сочетания, при которых эти элементы не меняются.

Сначала ставим на место элемент с номером **q**. По условию на этом месте может стоять любое число от **1** до **K**, поэтому сначала ставим **1** и находим все варианты, при которых **q+1** элементов уже поставлены (здесь будет рекурсивный вызов), затем ставим на это место **2** и т.д. Если **q=N**, то все элементы уже по-

ставлены на место, и надо печатать массив – одна из нужных комбинаций готова. Рекурсивная процедура приведена в листинге 11.7, которая использует массив **A[N]** и печатает все комбинации на экране.

Листинг 11.7

```
void Combinations ( int A[], int N, int K, int q )
{
   int i;
   if ( q == N ) // одна комбинация получена
      PrintData ( A, N );
   else
   for ( i = 1; i <= K; i ++ )
   {
      A[q] = i;
      Combinations(A, N, K, q+1); // рекурсивный вызов
   }
}
```

Для вывода массива на экран используется такая процедура приведенная в листинге 11.8.

Листинг 11.8

```
void PrintData ( int Data[], int N )
{
   int I;
   for ( i = 0; i < N; i++ )
      printf("%2d ", Data[i]);
   printf("\n");
}
```

В основной программе надо вызвать процедуру с параметром **q = 0**, поскольку ни один элемент еще не установлен (листинг 11.9).

Листинг 11.9

```
#include <stdio.h>
// здесь надо разместить процедуры

void main()
{
   int A[5], N = 5, K = 10;
   Combinations ( A, N, K, 0 );
}
```

11.7.2 Перестановки

Существует еще одна весьма непростая задача, которая хорошо решается с помощью рекурсии. Представьте, что к вам пришли **N** гостей. Сколько существует различных способов рассадить их за столом?

Сформулируем условие задачи математически. В массиве **A[N]** записаны целые числа. Надо найти все возможные перестановки этих чисел (в перестановку должны входить все элементы массива по **1** разу).

Предположим, что **q** элементов массива (с номерами от **0** до **q-1**) уже стоят на своих местах. Тогда в позиции **q** может стоять любой из неиспользованных элементов – их надо перебрать в цикле и вызвать рекурсивно эту же процедуру, увеличив на **1** количество установленных элементов. Чтобы не потерять никакой элемент, в цикле для всех **i** от **q** до **N-1** будем переставлять элементы **A[i]** и **A[q]**, генерировать все возможные комбинации, а затем менять их местами, восстанавливая исходный порядок.

Для обмена местами двух элементов массива будем использовать вспомогательную переменную **temp** (листинг 11.10).

Листинг 11.10

```
void Permutations ( int A[], int N, int q )
{
   int temp, i;
   if ( q == N ) // перестановка получена
      PrintData ( A, N ); // вывод на экран
   else
      for ( i = q; i <= N; i ++ )
      {
         temp = A[q];
         A[q] = A[i];
         A[i] = temp; // A[q]<->A[i]
         Permutations(A, N, q+1); // рекурсивный вызов
         temp = A[q];
         A[q] = A[i];
         A[i] = temp; // A[q]<->A[i]
      }
}
```

12 СТРУКТУРЫ В СИ

Структура – это конструкция, которая позволяет объединить несколько переменных с разными типами и именами в один составной объект. Она позволяет строить новые типы данных языка **Си**. В других языках программирования структуры называют записями или кортежами.

Отличия от массива:

- в структуре могут содержаться данные различных типов;
- в массиве обращение к данным идет по номеру, а в структуре – по имени.

12.1 Объявление шаблонов структур

Общий синтаксис объявления **шаблона структуры**:

```
struct имя_шаблона {
                    тип_элемента1 имя1 ;
                    тип_элемента2 имя2 ;
                    .....................
                    тип_элементаN имяN ;
                   };
```

Имена **шаблонов** должны быть **уникальными** в пределах их **области определения** для того, чтобы компилятор мог различать типы шаблонов. Задание шаблона осуществляется с помощью ключевого слова **struct**, за которым следует имя шаблона структуры и список элементов, заключенных в фигурные скобки:

```
struct DataBase {
                 char Fam[20];
                 char Name[15];
                 long TelephoneNumber;
                 char *Adress;
                 double w;
                };
```

Имена элементов **в одном шаблоне** также должны быть **уникальными**. Однако в разных шаблонах можно использовать одинаковые имена элементов.

Задание только шаблона **не влечет** резервирования памяти компилятором. Шаблон представляет компилятору необходимую информацию об элементах структурной переменной для **резервирования** места в оперативной памяти и организации доступа к ней **при определении** структурной переменной и использовании ее отдельных элементов.

В числе членов данных структуры могут также находиться, кроме стандартных типов данных (**int**, **float**, **char** и т.д.), ранее определенные типы, например:

```
struct date {// Объявление шаблона структуры типа date
              int day, month, year;
            };
struct person {    // Шаблон структуры типа person
              char fam[30],
                im[20],
                otch[15];
                float weight;
                int height;
                struct date birthday;
                };
```

Структура **date** имеет три поля типа **int**. Шаблон структуры **person** в качестве элемента включает поле **birthday**, которое, в свою очередь, имеет ранее объявленный тип данных: **struct** date. Этот элемент (**birthday**) содержит в себе все компоненты шаблона **struct** date.

12.2 Объявление структур-переменных

Определение структуры-переменной ничем не отличается от объявления обычной переменной с преопределенным типом. Общий синтаксис:

struct имя_шаблона имя_переменной;

Например:

```
struct date datel[15];
// Объявляется массив из 15 структур
struct person bstu[20], *per_ptr=&bstu[0];
/* либо *per_ptr=bstu;
   здесь объявили массив из 20 структур типа person
   и указатель на данный тип */
```

Компилятор выделит под каждую переменную количество байтов памяти, необходимое для хранения всех ее элементов.

Разрешается совмещать описание шаблона и определение структурной переменной.

Например:

```
struct date {
              int day, month, year;
            } a, b, datel[15], *ptr_a=&a;
```

12.3 Доступ к компонентам структуры

Доступ к компонентам структуры осуществляется с помощью оператора «.» при непосредственной работе со структурой или «->» – при использовании указателей на структуру. Общий синтаксис для доступа к компонентам структуры следующий:

```
имя_переменной_структуры.имя_поля;
имя_указателя->имя_поля;
(*имя_указаетля).имя_поля;
```

Прямой доступ к элементу (оператор «.»):

```
datel[5].day=10;
datel[5].year=1983;
strcpy(bstu[10].fam,"Иванов");
strcpy(bstu[10].im,"Сергей");
bstu[10].weight=70.0;
bstu[10].height=180;
bstu[10].birthday.day=15;
bstu[10].birthday.month=1;
bstu[10].birthday.year=1980;
```

Доступ по указателю (оператор «->»):

```
strcpy((per_ptr+10)->fam,"Иванов");
strcpy((per_ptr+10)->im,"Сергей");
(per_ptr+10)->weight=70.0;
(per_ptr+10)->height=180;
(per_ptr+10)->birthday.day=15;
(per_ptr+10)->birthday.month=1;
(per_ptr+10)->birthday.year=1980;
```

В том случае, если начальные значения членов структуры известны, можно присвоить их при определении переменной. Если вы создаете только одну переменную структурного типа, можно инициализировать ее как часть определения структуры:

```
struct CD {
            char name[20];
            char description[40];
            char category[12];
            float cost;
            int number;
          } disc = {"Лучшие песни", "Агата Кристи",
"рок", 12.50, 12};
```

Как другой возможный вариант, можно инициализировать члены структуры при определении переменной:

```
struct CD disc = {"Моя жизнь", "Б.Гейтс", "книга на диске", 24.99, 213};
```

12.4 Анонимное определение структуры и оператор определения типа typedef

Возможно анонимное определение структуры, когда имя структуры после ключевого слова **struct** опускается. В этом случае список описываемых переменных должен быть непустым (иначе такое описание совершенно бессмысленно).

```
struct {
        double x;
        double y;
       } t, *p;
```

Здесь имя шаблона структуры отсутствует. Определены две переменные **t** и **p**, первая имеет структурный тип с полями **x** и **y** типа **double**, вторая – указатель на данный структурный тип. Такие описания в чистом виде программисты обычно не используют, гораздо чаще анонимное определение структуры комбинируют с оператором определения имени типа **typedef**. Например, можно определить два типа **R2Point** (точка вещественной двумерной плоскости) и **R2PointPtr** (указатель на точку вещественной двумерной плоскости) в одном предложении, комбинируя оператор **typedef** с анонимным определением структуры:

```
typedef struct {
                  double x;
                  double y;
                } R2Point, *R2PointPtr;
```

Такая технология довольно популярна среди программистов и применяется в большинстве системных **h**-файлов. Преимущество ее состоит в том, что в дальнейшем при описании переменных структурного типа не нужно использовать ключевое слово **struct**, например:

```
R2Point a, b, c;    // Описываем три точки a, b, c
R2PointPtr p;       // Описываем указатель на точку
R2Point *q;         // Эквивалентно R2PointPtr q;
```

Эквивалентно следующим описаниям:

```
struct R2_Point{
                  double x, y;
                 };
struct R2_Point a, b, c;
struct R2_Point *p;
struct R2_Point *q;
```

Первый способ лаконичнее и нагляднее.

Вовсе не обязательно комбинировать оператор **typedef** непременно с анонимным определением структуры. Можно в одном предложении как определить имя структуры, так и ввести новый тип. Например:

```
typedef struct R2_Point {
                          double x;
                          double y;
                         } R2Point, *R2PointPtr;
```

определяет структуру **R2_point**, а также два новых типа **R2Point** (структура **R2_Point**) и **R2PointPtr** (указатель на структуру **R2_Point**). К сожалению, имя структуры не должно совпадать с именем типа, именно поэтому здесь в качестве имени структуры приходится использовать несколько вычурное имя **R2_Point**. Впрочем, обычно в дальнейшем оно не нужно.

12.5 Объединения

Объединение - это особая разновидность структурированного типа данных в **Си**, которая так же как структура позволяет объединить несколько разнотипных полей в один составной объект. Но, в отличие от структуры, все поля объединения начинаются с одного и того же адреса, и таким образом используют один и тот же блок памяти. Поэтому объединения используют не для соединения нескольких разнотипных переменных, а для объявления нескольких вариантов доступа к одному и тому же блоку памяти.

Объявление шаблона объединения отличается от объявления шаблона структуры только ключевым словом union, которое используется вместо **struct**:

```
union имя_шаблона {
                     тип_элемента1 имя1 ;
                     тип_элемента2 имя2 ;
                     .....
                     тип_элементаN имяN ;
                    };
```

Обращение к полям объединения выполняется в точности так же, как к полям структуры. Однако необходимо учитывать, что обращаясь к разным полям объединения - мы всего лишь обращаемся к одному и тому же элементу данных как к объекту разных типов.

13 РАБОТА С ПАМЯТЬЮ В СИ

В традиционных языках программирования, таких как **Си**, **Фортран**, **Паскаль**, существуют три вида памяти: статическая, стековая и динамическая. Конечно, с физической точки зрения никаких различных видов памяти нет: оперативная память – это массив байтов, каждый байт имеет адрес, начиная с нуля. Когда говорится о видах памяти, имеются в виду способы организации работы с ней, включая выделение и освобождение памяти, а также методы доступа.

13.1 Статическая память

Статическая память выделяется еще до начала работы программы, на стадии компиляции и сборки. Статические переменные имеют фиксированный адрес, известный до запуска программы и не изменяющийся в процессе ее работы. Статические переменные создаются и инициализируются до входа в функцию **main**, с которой начинается выполнение программы.

Существует два типа переменных в статической памяти:

1. Глобальные переменные – это переменные, определенные **вне функций**, в описании которых **отсутствует** слово **static**. Обычно описания глобальных переменных, включающие слово **extern**, выносятся в заголовочные файлы (**h**-файлы). Слово **extern** означает, что переменная описывается, но не создается в данной точке программы. Определения глобальных переменных, т.е. описания без слова **extern**, помещаются в файлы реализации (**c**-файлы или **cpp**-файлы).

Пример: глобальная переменная **maxind** описывается дважды:

1) в **h**-файле с помощью строки

extern int maxind**;**

Это описание сообщает о наличии такой переменной, но не создает эту переменную!

2) в **cpp**-файле с помощью строки

int maxind **= 1000;**

Это описание создает переменную **maxind** и присваивает ей начальное значение **1000**. Заметим, что стандарт языка не требует обязательного присвоения начальных значений глобальным переменным, но, тем не менее, это лучше делать всегда, иначе в переменной будет содержаться непредсказуемое

значение. Инициализация всех глобальных переменных при их определении – это правило хорошего стиля.

Глобальные переменные называются так потому, что они доступны в любой точке программы во всех ее файлах. Поэтому имена глобальных переменных должны быть достаточно длинными, чтобы избежать случайного совпадения имен двух разных переменных. Например, имена **x** или **n** для глобальной переменной не подходят.

2. Статические переменные – это переменные, в описании которых **присутствует** слово **static**. Как правило, статические переменные описываются вне функций. Такие статические переменные во всем подобны глобальным, с одним исключением: область видимости статической переменной ограничена одним файлом, внутри которого она определена, – и, более того, ее можно использовать только после ее описания, т.е. ниже по тексту. По этой причине описания статических переменных обычно выносятся в начало файла. В отличие от глобальных переменных, статические переменные никогда не описываются в **h**-файлах (модификаторы **extern** и **static** конфликтуют между собой). **Совет**: используйте статические переменные, если нужно, чтобы они были доступны только для функций, описанных внутри одного и того же файла. По возможности, не применяйте в таких ситуациях глобальные переменные, это позволит избежать конфликтов имен при реализации больших проектов, состоящих из сотен файлов.

Статическую переменную можно описать и внутри функции, хотя обычно так никто не делает. Переменная размещается не в стеке, а в статической памяти, т.е. ее нельзя использовать при рекурсии, а ее значение сохраняется между различными входами в функцию. Область видимости такой переменной ограничена телом функции, в которой она определена. В остальном она подобна статической или глобальной переменной. Заметим, что ключевое слово **static** в языке **Си** используется для двух различных целей:

1) как указание типа памяти: переменная располагается в статической памяти, а не в стеке;
2) как способ ограничить область видимости переменной рамками одного файла (в случае описания переменной вне функции).

Слово **static** может присутствовать и в заголовке функции. При этом оно используется только для того, чтобы ограничить область видимости имени функции рамками одного файла.

Пример:

```
static int gcd(int x, int y); // Прототип функции
...
static int gcd(int x, int y)
{          // Реализация
   ...
}
```

Совет: используйте модификатор **static** в заголовке функции, если известно, что функция будет вызываться лишь внутри одного файла. Слово **static** должно присутствовать как в описании прототипа функции, так и в заголовке функции при ее реализации.

13.2 Стековая, или локальная, память

Локальные, или **стековые**, **переменные** – это переменные, описанные внутри функции. Память для таких переменных выделяется в аппаратном стеке. Память выделяется в момент входа в функцию или блок и освобождается в момент выхода из функции или блока. При этом захват и освобождение памяти происходят практически мгновенно, т.к. компьютер только изменяет регистр, содержащий адрес вершины стека.

Локальные переменные можно использовать при рекурсии, поскольку при повторном входе в функцию в стеке создается новый набор локальных переменных, а предыдущий набор не разрушается. Это очень важное качество с точки зрения надежности и безопасности программы! Программа, работающая со статическими переменными, этим свойством не обладает, поэтому для защиты статических переменных приходится использовать механизмы синхронизации, а логика программы резко усложняется. Всегда следует избегать использования глобальных и статических переменных, если можно обойтись локальными.

Недостатки локальных переменных являются продолжением их достоинств. Локальные переменные создаются при входе в функцию и исчезают после выхода из нее, поэтому их нельзя использовать в качестве данных, разделяемых между несколькими функциями. К тому же, размер аппаратного стека не бесконечен, стек может в один прекрасный момент переполниться (например, при глубокой рекурсии), что приведет к катастрофическому завершению программы. Поэтому локальные переменные не должны иметь большого размера. В частности, нельзя использовать большие массивы в качестве локальных переменных.

Пример совместного использования глобальных и локальных переменных представлен в листинге 13.1.

Листинг 13.1

```
#include <stdio.h>

int var = 0; // объявление глобальной переменной

void ProcNoChange ()
{
   int var; // локальная переменная
   var = 3; // меняется локальная переменная
}

void ProcChange1 ()
{
   var = 5; // меняется глобальная переменная
}

void ProcChange2 ()
{
   int var; // локальная переменная
   var = 4; // меняется локальная переменная
   ::var = ::var * 2 + var;
   // меняется глобальная переменная
}

void main()
{
   ProcChange1(); // var = 5;
   ProcChange2(); // var = 5*2 + 4 = 14;
   ProcNoChange(); // var не меняется
   printf ( "%d", var );
   // печать глобальной переменной (14)
}
```

13.3 Динамическая память, или куча

Помимо **статической** и **стековой** памяти, существует еще практически неограниченный ресурс памяти, которая называется **динамической**, или **кучей** (**heap**). Программа может захватывать участки динамической памяти нужного размера. После использования ранее захваченный участок динамической памяти следует освободить.

Под **динамическую** память отводится пространство виртуальной памяти процесса между статической памятью и стеком.

Обычно стек располагается в старших адресах виртуальной памяти и растет в сторону уменьшения адресов. Программа и константные данные размещаются в младших адресах, выше располагаются статические переменные. Пространство выше статических переменных и ниже стека занимает динамическая память:

адрес	содержимое памяти
0 4 8	код программы и данные, защищенные от изменения
...	статические переменные программы
	динамическая память
max.адрес (2^{32}-4)	стек ↑

Структура динамической памяти автоматически поддерживается исполняющей системой языка **Си**. Динамическая память состоит из захваченных и свободных сегментов, каждому из которых предшествует описатель сегмента. При выполнении запроса на захват памяти исполняющая система производит поиск свободного сегмента достаточного размера и захватывает в нем отрезок требуемой длины. При освобождении сегмента памяти он помечается как свободный, при необходимости несколько подряд идущих свободных сегментов объединяются.

В языке **Си** для захвата и освобождения динамической памяти применяются стандартные функции **malloc()** и **free()**, описания их прототипов содержатся в стандартном заголовочном файле «**stdlib.h**». Имя **malloc** является сокращением от **memory allocate** – «захват памяти». Прототипы этих функций выглядят следующим образом:

```
void *malloc(size_t n);
// Захватить участок памяти размером в n байт
void free(void *p);
// Освободить участок памяти с адресом p
```

Здесь **n** – это размер захватываемого участка в байтах, **size_t** – имя одного из целочисленных типов, определяющих максимальный размер захватываемого участка. Тип **size_t** задается в стандартном заголовочном файле «**stdlib.h**» с помощью оператора **typedef**. Это обеспечивает независимость текста **Си**-программы от используемой архитектуры. В **32**-разрядной архитектуре тип **size_t** определяется как беззнаковое целое число:

```
typedef unsigned int size_t;
```

Функция **malloc()** возвращает адрес захваченного участка памяти или ноль в случае неудачи (когда нет свободного участ-

ка достаточно большого размера). Функция **free()** освобождает участок памяти с заданным адресом. Для задания адреса используется указатель общего типа **void***. После вызова функции **malloc()** его необходимо привести к указателю на конкретный тип, используя операцию приведения типа.

Например, в следующем примере захватывается участок динамической памяти размером в **4000** байтов, его адрес присваивается указателю на массив из **1000** целых чисел:

```
int *a;    // Указатель на массив целых чисел
...
a = (int *) malloc(1000 * sizeof(int));
```

Выражение в аргументе функции **malloc()** равно **4000**, поскольку размер целого числа **sizeof(int)** равен **4** байтам. Для преобразования указателя используется операция приведения типа **(int *)** от указателя обобщенного типа к указателю на целое число.

Ошибки, связанные с выделением памяти

Самые тяжелые и трудно вылавливаемые ошибки в программах на языке **Си** связаны именно с неверным использованием динамических массивов. В таблице перечислены наиболее тяжелые случаи и способы борьбы с ними.

Ошибка	Причина и способ лечения
Запись в чужую область памяти	Память была выделена неудачно, а массив используется. **Вывод**: надо всегда делать проверку указателя на **NULL**.
Повторное удаление указателя	Массив уже удален и теперь удаляется снова. **Вывод**: если массив удален из памяти, обнулите указатель – ошибка быстрее выявится.
Выход за границы массива	Запись в массив в элемент с отрицательным индексом или индексом, выходящим за границу массива
Утечка памяти	Неиспользуемая память не освобождается. Если это происходит в функции, которая вызывается много раз, то ресурсы памяти скоро будут израсходованы. **Вывод**: убирайте за собой «мусор» (освобождайте память).

14 КОНСОЛЬНЫЙ ВВОД/ВЫВОД

Консоль – совокупность основных устройств ввода/вывода информации. Некоторые из консолей представляют информацию только в текстовом виде с использованием экранных шрифтов формата видеосистемы (текстовые консоли). Графические консоли представляют информацию в графическом виде, используя графический пользовательский интерфейс.

Текстовый интерфейс пользователя – система средств взаимодействия пользователя с компьютером, основанное на использовании текстового (буквенно-цифрового) режима дисплея или аналогичных устройств (командная строка).

Приложения, использующие текстовый интерфейс, называются консольными. На программном уровне для ввода и вывода информации консольные приложения используют стандартные устройства ввода/вывода (**stdin**, **stdout**, **stderr**).

Функции, прототипы которых помещены в файле «**stdio.h**», осуществляют т.н. потоковый ввод/вывод. Используя их, невозможно выполнить, например, не отображаемый на экране ввод символов, «обойти» обработку реакции на нажатие комбинации клавиш «**Ctrl-Break**», определить нажатие специальных клавиш.

Для выполнения таких действий используются функции **Turbo C**, прототипы которых помещены в файле «**conio.h**». Они не являются частью **ANSI**-стандарта языка и входят в расширение **Turbo C**.

Функции рассчитаны, прежде всего, на построение простейшего оконного интерфейса и поэтому при любом выводе на экран (в т.ч. и «эха») дополнительно корректируются переменные, хранящие текущие координаты курсора активного окна. Кроме того, имеется возможность управления цветом вывода.

14.1 Функции консольного ввода

char *cgets(char *str) – помещает в буфер, на начало которого указывает **str**, строку символов со стандартного ввода. Запись символов начинается с **str[1]**; **str[0]** должен содержать максимальное число символов, которое должно быть прочитано и записано в строку. Функция возвращает указатель на начало буфера **str**.

Перечислим достоинства этой функции по сравнению с **gets()**:

1) возможность определения при вводе длины строки;

2) защита при вводе от «лишних» символов, для которых компилятором не зарезервировано место;

3) возможность ввода за одно обращение к функции строк, длина которых превышает установленный по умолчанию буфер для стандартного ввода в **128** символов.

Пример. Использование функции **cgets()** для ввода строки до 254 символов за одно обращение представлено в листинге 14.1.

Листинг 14.1

```
// Пример ввода строки до 254 символов
// за одно обращение
#include <conio.h>
#include <stdio.h>

void main()
{
   char str[256];
   str[0] = 254;
   /* Ограничитель максимальной длины строки */
   puts("Введите строку, длина которой может быть
до 254 символов");
   cgets(str);
   printf("\nВведено %d символов строки: \n%s\n",
str[1], &str[2]);
}
```

int cscanf(char*format[, argument, ...]) – выполняет форматированный ввод с клавиатуры (квадратные скобки означают, что элемент может отсуствовать, т.е. является необязательным). В отличие от функции **scanf()** не выполняет буферизацию ввода: все символы, вводимые с клавиатуры, доступны программе немедленно. Ввод пробела рассматривается как завершение ввода.

int getch(void) – выполняет ввод символа с клавиатуры. **Turbo C** не выполняет «эхо» ввода. В этой связи полезна для организации интерфейса с пользователем, при котором нажатие той или иной клавиши вызывает немедленную реакцию программы без отображения введённого символа на экране.

int getche(void) – выполняет небуферизуемый ввод символа с клавиатуры. **Turbo C** «эхоирует» ввод на экране. Перевод строки происходит при достижении правой вертикальной границы текущего активного окна.

Пример. Программа (листинг 14.2) иллюстрирует применение функций **getch()** и **getche()** для определения нажатий не только **ASCII**-клавиш, но и специальных клавиш:

Листинг 14.2

```
// Демонстрирует ввод нажатий специальных клавиш
#include <conio.h>
#include <stdio.h>

void main()
{
   int ch;
   do {
      puts("Нажмите любую клавишу...");
      if(!(ch = getch()))
      {
         ch = getch();
         printf("Спец. клавиша. Расширенный скан-код
%#u\n",ch);
      }
      else printf("Символьная клавиша %c (ASCII-код
%#u)\n",ch,ch);
      puts("Продолжение? (y/n)");
   } while ((ch = getch()) == 'y' || ch == 'Y');
}
```

char *getpass(const char *prompt) – выводит на экран **ASCIIZ**-строку, на начало которой указывает **prompt**, а затем принимает с клавиатуры без «эха» строку символов. Вводимые символы (не более 7) помещаются во внутреннюю статическую память. Функция возвращает указатель на внутреннюю статическую строку, переопределяемую каждым новым обращением к функции. Основное назначение данной функции – ввод паролей в программе без отображения их на экран.

Пример. Программа (листинг 14.3) демонстрирует ввод и проверку санкционированности доступа к программе по паролю.

Листинг 14.3

```
// Прописные и строчные буквы в пароле
// и вводимой с клавиатуры строке являются различными
#include <conio.h>
#include <stdio.h>
#include <string.h>

int main()
{
   char *correct_string = "PaSsWoRd";
   char *input_string;
```

```
    input_string = getpass("Введите пароль:");
    if (strcmp(input_string,correct_string) != 0)
    {
       puts("\aНекорректный пароль. Завершение...");
       return 1;
    }
    puts("Доступ разрешен.");
    return 0;
}
```

int kbhit(void) – проверяет, пуст ли буфер клавиатуры. Если в буфере есть символы, функция возвращает ненулевое значение, в противном случае она возвращает **0**. Является удобным средством предотвращения «зацикливания» или «повисания» при ожидании невозможного в данный момент события. Кроме того, осуществляется проверка нажатия комбинации клавиш «**Ctrl-Break**», что позволяет выполнить аварийное завершение программы.

int ungetch(int ch) – записывает непосредственно в буфер клавиатуры символ **ch**. Он будет доступен при выполнении следующей операции чтения с консоли (функциями файла «**conio.h**»). Разрешает помещать только один символ, который не должен совпадать с константой **EOF**, описанной в файле «**stdio.h**». В случае успеха функция возвращает **ch**; в противном случае возвращается **-1**.

Пример. Программа (листинг 14.4) помещает строку символ за символом в буфер клавиатуры. Затем строка читается и выводится на экран.

Листинг 14.4

```
// Помещает строку символ за символом в буфер
// клавиатуры.
// Затем строка считывается и выводится на экран.
#include <conio.h>
#include <stdio.h>

void main()
{
   char *string = "Тестовая строка для вывода на экран";
   while (*string)
   {
      ungetch(*string++);
      putchar(getch());
   }
}
```

14.2 Функции консольного вывода

В текстовом режиме экран можно представить как совокупность так называемых **текстелов** (**texel – Text Element**).

Каждому знакоместу экрана (текселу) в текстовом режиме соответствуют два байта. Первый байт хранит **ASCII**-код символа, а следующий за ним байт кодирует особенности отображения символа на экране: цвет символа (**Foreground Color**) и цвет фона символа (**Background Color**).

Turbo C предоставляет возможность изменять параметры текстового режима при помощи следующей функции, прототип которой находится в файле «**conio.h**»:

void textmode(int newmode) – изменяет текущий текстовый режим. Новый режим указывается единственным параметром **newmode** и может задаваться либо числом, либо с использованием символических констант, значения которых определяет перечислимый тип **text_modes**.

```
enum text_modes
{ LASTMODE=-1, /* Предыдущий текстовый режим */
  BW40=0,      /* 40 столбцов X 25 строк,
                  черно-белое изображение */
  C40,         /* 40 столбцов X 25 строк,
                  цветное изображение */
  B80,         /* 80 столбцов X 25 строк,
                  черно-белое изображение */
  C80,         /* 80 столбцов X 25 строк,
                  цветное изображение */
  MONO=7,      /* 40 столбцов X 25 строк,
                  монохромное изображение */
  C4350=64     /* 80 столбцов X 50 строк,
}                 цветное изображение */
```

Функции консольного вывода используют понятие активного окна экрана. **Активное окно** – это прямоугольная область экрана, в границах которой в данный момент работают функции. Описание активного окна (или, как часто говорят, **фрейм**) хранится во внутренней структурной переменной **Turbo C**. Установку параметров активного текстового окна выполняет функция **window ()**.

void window(int l_t_col, int l_t_row, int r_b_col, int r_b_row) – описывает активное текстовое окно: первая пара аргументов задает столбец и строку левого верхнего угла, вторая пара – правого нижнего угла. Строки и столбцы нумеруются, начиная от **1**. Поэтому, например, координаты левого верх-

него и правого нижнего углов экрана в режимах «25 строк х 80 столбцов» задаются парами **(1,1)** и **(80,25)**. **Ось X** направлена слева направо, а **ось Y** направлена сверху вниз. Следует обратить внимание на то, как в **Turbo C** задаются координаты углов, сначала столбец, затем строка.

Фрейм окна **Turbo C** имеет следующую структуру:

```
struct text_info {
   unsigned char winleft;      /* столбец, строка */
   unsigned char wintop;       /* левого верхнего угла */
   unsigned char winright;     /* столбец, строка */
   unsigned char winbottom;/* правого верхнего угла */
   unsigned char attribute;          /* атрибуты */
   unsigned char normattr;           /* окна */
   unsigned char screenheight;
   /* полная высота экрана */
   unsigned char screenwidth;
   /* полная ширина экрана */
   unsigned char curx;      /* строка, столбец */
   unsigned char cury  /* текущей позиции курсора */
}
```

Информация об активном окне доступна при выполнении функции **gettextinfo()**:

void gettextinfo(struct text_info *r) – заполняет поля структурной переменной по шаблону **text_info**, на которую ссылается. Шаблон структуры **text_info**, описывающей текущее окно экрана, содержится в заголовочном файле «**conio.h**».

Функция **window()** инициализирует поля координат фрейма окна. Функции **textcolor()**, **textbackground()**, **textattr()** и другие управляют цветом отображаемых символов окна.

Описываемые далее функции влияют на атрибут символа. Атрибут задаёт битами **0-2** код цвета символа, бит **3** определяет повышение яркости, биты **4-6** задают код цвета фона символа. **Turbo C** дает возможность задать атрибут полностью либо задать только цвет символа или фона. Цвета могут задаваться либо числом, либо с использованием символических констант, значения которых определяет перечислимый тип **COLORS**:

```
enum COLORS
{
   BLACK,          /* цвета нормальной яркости */
   BLUE,
   GREEN,
   CYAN,
```

```
    RED,
    MAGENTA,
    BROWN,
    LIGHTGRAY    /* цвета повышенной яркости */
    DARKGRAY,
    LIGHTBLUE,
    LIGHTGREEN,
    LIGHTCYAN,
    LIGHTRED,
    LIGHTMAGENTA,
    YELLOW,
    WHITE
};
```

Яркие цвета могут задаваться только цветом символа. Кроме того, **7**-й бит (бит мерцания) может быть задан как непосредственно в коде байта атрибута, так и с использованием символической константы **BLINK**, определяемой как **128** в «**conio.h**». Следует отметить тот факт, что если для цвета фона выбирают цвета с кодами **8-15**, это устанавливает в единицу бит мерцания символа в байте атрибута.

void textattr(int newattr) – устанавливает атрибут для функций, работающих с текстовыми окнами. Атрибут хранится в поле **attribute** структурной переменной по шаблону **text_info**, доступной через функцию **gettextinfo()**. Задаваемый атрибут может быть или числом, например, **0x70** – атрибут инверсного изображения (чёрные символы на светло-сером фоне), или формироваться из символических констант, значения которых задаёт тип **COLORS**. Например, для задания мерцающих ярко-красных символов на сером фоне атрибут можно сформировать следующим образом:

```
BLINK | (BLACK << 4) | LIGHTRED
```

Тот же результат может быть получен и так:

```
(DARKGRAY << 4) | LIGHTRED
```

void textcolor(int newcolor) – задает цвет символов, не затрагивая установленный цвет фона. Цвет может быть или числом, или формироваться из символических констант, значения которых определяет перечисляемый тип **COLORS**.

void textbackground(int newcolor) – задает цвет фона символов, не затрагивая установленный цвет символа. Цвет может быть или числом, или формироваться из символических констант, значения которых определяет перечисляемый тип

COLORS. Для цвета фона выбор ограничен значениями цветов **0-7**. Если для цвета фона выбирается значение **8-15**, то символы будут мерцать, так как бит мерцания установится в единицу, но цвет фона будет соответствовать значениям **0-7**.

Пример. Программа (листинг 14.5), образующая на экране окно ввода-вывода. Программа принимает с клавиатуры целое число и строку символов и затем выводит их на экран. Окно ввода-вывода занимает 25 столбцов и 10 строк в левом верхнем углу экрана. Предыдущее содержимое экрана сохраняется, а перед завершением программы восстанавливается.

Листинг 14.5

```
// Образует окно ввода-вывода и демонстрирует его использование
#include <conio.h>
#include <alloc.h>
#include <string.h>

int main()
{
   unsigned number;
   char string[129], *buf_screen;
   // Описание окна ввода-вывода
   window(1,1,25,10);
   textbackground(BLUE);  // Выбираем синий фон
   textcolor(YELLOW);     // Выбираем желтый цвет
                          // для символов
/* Сохранение предыдущего содержимого экрана для его
восстановления после завершения работы программы. Сначала динамически распределяем память для хранения копии видеобуфера. */
   if ((buf_screen = malloc(25*10*2)) == NULL)
   {
      cputs("Не хватает оперативной памяти для
             работы. \xd\xa");
      return (-1);
   }
   gettext(1,1,25,10,buf_screen);
   // Сохранение содержимого экрана
   clrscr();
   // Очистка экрана и «заливка» его синим цветом
   cputs("Введите целое число: ");
   cscanf("%u", &number);
   cputs("\xd\xaВведите строку символов \xd\xa");
```

```
    cscanf("%128s",string);
    cprintf("\xd\xaВведено число %u\xd\xa",
number);
    cprintf("\xd\xa и строка символов %s\xd\xa",
string);
    cprintf("\xd\xaДлина строки - %d\xd\xa",
strlen(string));
    // Восстановление экрана
    cputs("\xd\xaДля продолжения нажмите любую
клавишу...");
    while (!kbhit()) {}
    puttext(1,1,25,10,buf_screen);
    return 0;
}
```

Среди функций консольного ввода-вывода **Turbo C** текущей позицией курсора в окне управляет функция **gotoxy()**.

void gotoxy(int x, int y) – устанавливает курсор в строку **y** и столбец **x** в текущем активном окне экрана. Верхний левый угол окна имеет координаты **(1,1)**. При попытке позиционировать курсор за границы окна он останавливается на границе окна. Особенностью функции является то, что координаты **x** и **y** являются относительными, приведенными к левому верхнему углу. Например, если текущее окно было описано функцией **window(1,8,80,25)**, обращение **gotoxy(5,5)**; установит курсор в пятый относительный столбец окна (совпадает с абсолютным столбцом **4**, отсчитываемым от **0**) в пятой относительной строке (так как верхняя строка окна задана равной **5**, то абсолютная строка будет равна **5+8-1**, если отсчет строк ведется от **0**).

Текущую позицию курсора в активном текстовом окне сообщают функции **wherex()** и **wherey()**.

int wherex(void), int wherey(void) – сообщают столбец и строку текущей позиции курсора; возвращают целое число в диапазоне соответственно от **1** до **80** и от **1** до **25**.

Кроме того, текущая позиция курсора в окне возвращается в полях **curx** и **cury** структурной переменной по шаблону **text_info**, заполняемой при выполнении функции **gettextinfo()**.

Функции вывода информации в окно экрана, в отличие от функций стандартного ввода-вывода, позволяют управлять цветом выводимых символов и не пересекают пределы активного в данный момент окна. При достижении правой вертикальной границы курсор автоматически переходит на начало следующей строки в пределах окна, а при достижении нижней горизонтальной грани выполняется скроллинг окна вверх.

void clreol(void) – стирает в текстовом окне строку, на которую установлен курсор, начиная с текущей позиции курсора и до конца строки (до правой вертикальной границы окна).

void clrscr(void) – очищает все текстовое окно. Цвет «заливки» окна при очистке будет соответствовать значению, установленному символической переменной **attribute** в описании окна (структурная переменная по шаблону **text_info)**.

Функции управления цветом фона и символа:

void delline(void) – стирает в текстовом окне всю строку текста, на которую установлен курсор.

void insline(void) – вставляет пустую строку в текущей позиции курсора со сдвигом всех остальных строк окна на одну строку вниз. При этом самая нижняя строка текста окна теряется.

int cprintf(const char *format, ...) – выполняет вывод информации с преобразованием по заданной форматной строке, на которую указывает **format**. Является аналогом функции стандартной библиотеки **printf()**, но выполняет вывод в пределах заданного окна. В отличие от **printf()** функция **cprintf()** иначе реагирует на специальный символ '**\n**' – курсор переводится на новую строку, но не возвращается к левой границе окна. Поэтому для перевода курсора на начало новой строки текстового окна следует вывести последовательность символов **CR-LF** (**0x0d.0x0a**). Остальные специальные символы воздействуют на курсор так же, как и в случае функций стандартного ввода-вывода. Функция возвращает число выведенных байтов, а не число обработанных полей, как это делает функция **printf()**.

int cputs(const char *str) – выводит строку символов в текстовое окно, начиная с текущей позиции курсора. На начало выводимой **ASCIIZ**-строки указывает **str**. Является аналогом функции стандартной библиотеки **puts()**, выполняет вывод в пределах заданного окна и при выводе не добавляет специальный символ '**\n**'. Реакция **cputs()** на специальный символ '**\n**' аналогична реакции **cprintf()**: курсор переводится на новую строку, но не возвращается к левой границе окна. Поэтому для перевода курсора на начало новой строки текстового окна следует вывести последовательность символов **CR-LF** (**0x0d.0x0a**). Остальные специальные символы воздействуют на курсор так же, как и в случае функций стандартного ввода/вывода. Функция возвращает **ASCII**-код последнего выведенного на экран символа. В отличие от **puts()** в функции отсутствует возврат символа **EOF** (вывод на экран происходит в любом случае).

int movetext(int left, int top, int right, int bottom, int destleft, int desttop) – переносит окно, заданное координатами

левого верхнего (**left, top**) и правого нижнего (**right, bottom**) углов, в другое место на экране, заданное координатами левого верхнего угла нового положения окна. Размеры окна по горизонтали и вертикали сохраняются. Все координаты задаются относительно координат верхнего левого угла экрана **(1,1)**. Функция возвращает ненулевое значение, если перенос заданного окна выполнен. В противном случае возвращается **0**. Функция корректно выполняет перекрывающиеся переносы, т.е. переносы, в которых прямоугольная область-источник и область, в которую окно переносится, частично покрывают друг друга.

int putch(int ch) – выводит символ в текущей позиции текстового окна экрана. Как и для функций **cprintf()**, **cputs()**, специальный символ '**\n**' вызывает только переход курсора на новую строку текстового окна без возврата к его левой вертикальной границе. Остальные специальные символы воздействуют на курсор так же, как и для функций стандартного ввода-вывода.

int puttext(int left, int top, int right, int bottom, void *source) – выводит на экран текстовое окно, заданное координатами левого верхнего (**left, top**) и правого нижнего (**right, bottom**) углов. Символы и атрибуты располагаются в буфере, адрес начала которого задаёт указатель **source** (функция «открывает» или «восстанавливает» текстовое окно экрана). Обычно используется вместе с функцией **gettext()**, выполняющей обратную операцию – запись в **source** символов/атрибутов, полностью описывающих все знакоместа текстового окна. Функция проверяет по заданным координатам окна, можно ли разместить окно на экране для текущего режима видеоадаптера и корректны ли эти координаты. В случае, когда окно успешно выведено, возвращается ненулевое значение.

int gettext(int left, int top, int right, int bottom, void *destin) – записывает в буфер **destin** символы и атрибуты текстового окна, заданного строкой и столбцом левого верхнего (**left**, **top**) и правого нижнего (**right**, **bottom**) углов. Первые два слова буфера занимают ширина и длина скопированного окна. Работает только в текстовых режимах видеоадаптера. Координаты задаются относительно верхнего левого угла экрана **(1,1)**. В случае успеха возвращает ненулевое число.

void highvideo(void), void lowvideo(void), void normvideo(void) – функции задают соответственно использование повышенной, пониженной и нормальной яркости для последующего вывода символов функциями файла «**conio.h**». Влияют на атрибут символа.

15 РАБОТА С ФАЙЛАМИ

15.1 Файловый ввод/вывод

Прототипы функций ввода-вывода и используемые для этого типы данных описаны в стандартном заголовочном файле «**stdio.h**».

Для файлового ввода/вывода в **Си** предусмотрены две основные группы функций:

- функции низкоуровневого ввода/вывода, использующие для доступа к файлам целочисленные файловые дескрипторы;
- функции более высокого уровня, осуществляющие буферизованный ввод/вывод с применением потоков.

Поток в **Си** – это объект, служащий для доступа к файлам как к упорядоченной последовательности символов.

Поток представляется структурой типа **FILE**, с которой ассоциирован некоторый открытый файл. При необходимости несколько потоков могут ссылаться на один и тот же файл.

Любым операциям ввода/вывода должен предшествовать вызов функции открытия файла. Исключением являются три системных потока, которые открыты заранее:

- **stdin** – стандартный входной поток – по умолчанию назначен на клавиатуру;
- **stdout** – стандартный выходной поток – по умолчанию назначен на экран;
- **srderr** – стандартный поток ошибок – по умолчанию также назначен на экран.

Эти потоки открываются системой еще перед началом работы программы. Переменные **stdin**, **stdout** и **stderr** являются глобальными и описаны в стандартном заголовочном файле «**stdio.h**».

После открытия файла индикатор текущей позиции устанавливается в начало (на нулевой байт) при условии, что файл не открывали на добавление, в этом случае индикатор должен указывать на конец файла.

В дальнейшем индикатор текущей позиции смещается под воздействием операций чтения, записи и позиционирования, чтобы упростить последовательное продвижение по файлу.

15.2 Открытие файла: функция fopen()

Для доступа к файлу применяется тип данных **FILE**. Это структурный тип, имя которого задано с помощью оператора

typedef в стандартном заголовочном файле «**stdio.h**». Программисту не нужно знать, как устроена структура типа файл: ее устройство может быть системно зависимым, поэтому в целях переносимости программ обращаться явно к полям структуры **FILE** запрещено. Тип данных «указатель на структуру **FILE**» используется в программах как черный ящик: функция открытия файла возвращает этот указатель в случае успеха, и в дальнейшем все файловые функции применяют его для доступа к файлу.

Прототип функции открытия файла выглядит следующим образом:

FILE *fopen(const char *path, const char *mode);

path – путь к файлу (например, имя файла или абсолютный путь к файлу);

mode – режим открытия файла.

Значения символов в строке **mode**:

- «**r**» – файл открывается для чтения, индикатор текущей позиции устанавливается в начало файла;
- «**w**» – файл открывается для записи (или создается, если не существовал прежде), индикатор текущей позиции устанавливается на начало файла, размер файла усекается до нуля (т.е. старое содержимое файла теряется);
- «**a**» – файл открывается для дозаписи в конец (или создается, если не существовал прежде), индикатор текущей позиции устанавливается в конец файла;
- «**r+**» – файл открывается для чтения и записи, индикатор текущей позиции устанавливается на конец файла;
- «**w+**» – файл открывается для чтения и записи (или создается, если не существовал прежде), индикатор текущей позиции устанавливается на начало файла, размер файла усекается до нуля;
- «**a+**» – файл открывается для чтения и дозаписи в конец (или создается, если не существовал прежде), индикатор текущей позиции устанавливается в конец файла.

Строка **mode** может так же включать в себя символ «**b**», который должен находиться **не на первой** позиции. Это означает, что файл открывается не в текстовом, а в бинарном режиме. При программировании для ОС DOS и Windows разница между текстовыми и двоичными режимами открытия файла заключается в интерпретации разделителей строк. В двоичном режиме разделитель строки «видится» **Си** как последовательность байт **0xD 0xA**, а в текстовом – как служебный символ «**\n**». При ра-

боте с текстовыми файлами целесообразно не использовать двоичный режим.

Соответственно:

- «**b**» – открытие файла в бинарном режиме;
- «**t**» – открытие файла в текстовом режиме.

Несколько примеров открытия файлов:

```
FILE *f, *g, *h;
...
// 1. Открыть текстовый файл «abcd.txt» для чтения
f = fopen("abcd.txt", "rt");

// 2. Открыть бинарный файл «c:\Windows\Temp\tmp.dat»
// для чтения и записи
g = fopen("c:/Windows/Temp/tmp.dat", "wb+");

// 3. Открыть текстовый файл
// «c:\Windows\Temp\abcd.log»
// для дописывания в конец файла
h = fopen("c:\\Windows\\Temp\\abcd.log", "at");
```

Обратите внимание, что во втором случае мы используем обычную косую черту **/** для разделения директорий, хотя в системах **MS DOS** и **MS Windows** для этого принято использовать обратную косую черту ****. Дело в том, что в операционной системе **Unix** и в языке **Си**, который является для нее родным, символ **** используется в качестве экранирующего символа, т.е. для защиты следующего за ним символа от интерпретации как специального. Поэтому во всех строковых константах **Си** обратную косую черту надо повторять дважды, как это и сделано в третьем примере. Впрочем, стандартная библиотека **Си** позволяет в именах файлов использовать нормальную косую черту вместо обратной; эта возможность была использована во втором примере.

В случае удачи функция **fopen()** открытия файла возвращает ненулевой указатель на структуру типа **FILE**, описывающую параметры открытого файла. Этот указатель надо затем использовать во всех файловых операциях. В случае неудачи (например, при попытке открыть на чтение несуществующий файл) возвращается нулевой указатель. При этом глобальная системная переменная **errno**, описанная в стандартном заголовочном файле «**errno.h**», содержит численный код ошибки. В случае неудачи при открытии файла этот код можно распечатать, чтобы получить дополнительную информацию:

```
#include <stdio.h>
#include <errno.h>
. . .
FILE *f = fopen("filnam.txt", "rt");
if (f == NULL) {
   printf( "Ошибка открытия файла с кодом %d\n",
errno);
   . . .
}
```

15.3 Диагностика ошибок: функция perror()

Использовать переменную **errno** для печати кода ошибки не очень удобно, поскольку необходимо иметь под рукой таблицу возможных кодов ошибок и их значений. В стандартной библиотеке **Си** существует более удобная функция **perror()**, которая печатает системное сообщение о последней ошибке вместо ее кода. Печать производится на английском языке, но есть возможность добавить к системному сообщению любой текст, который указывается в качестве единственного аргумента функции **perror()**. Например, предыдущий фрагмент переписывается следующим образом:

```
#include <stdio.h>
. . .
FILE *f = fopen("filnam.txt", "rt");
if (f == 0) {
   perror("Не могу открыть файл на чтение");
   . . .}
```

Функция **perror()** печатает сначала пользовательское сообщение об ошибке, затем после двоеточия системное сообщение. Например, при выполнении приведенного фрагмента в случае ошибки из-за отсутствия файла будет напечатано:

Не могу открыть файл на чтение: No such file or directory

15.4 Закрытие файла: функция fclose()

По окончании работы с файлом его надо обязательно закрыть. Система обычно запрещает полный доступ к файлу до тех пор, пока он не закрыт (например, в нормальном режиме система запрещает одновременную запись в файл для двух разных программ). Кроме того, информация реально записывается полностью в файл лишь в момент его закрытия. До этого она

может содержаться в оперативной памяти (в так называемой файловой кеш-памяти), что при выполнении многочисленных операций записи и чтения значительно ускоряет работу программы.

Для закрытия файла используется функция **fclose** с прототипом

```
int fclose(FILE *f);
```

В случае успеха функция **fclose()** возвращает ноль, при ошибке – отрицательное значение (точнее, константу – конец файла – **EOF**, определенную в системных заголовочных файлах как минус единица). При ошибке можно воспользоваться функцией **perror()**, чтобы напечатать причину ошибки. Отметим, что ошибка при закрытии файла – явление очень редкое (чего не скажешь в отношении открытия файла), так что анализировать значение, возвращаемое функцией **fclose()**, в общем-то, не обязательно. Пример использования функции **fclose()**:

```
FILE *f;
f = fopen("tmp.res", "wb"); // Открываем файл
«tmp.res»
if (f == 0) { // При ошибке открытия файла
   // Напечатать сообщение об ошибке
   perror("Не могу открыть файл для записи");
   exit(1); // завершить работу программы с кодом 1
}
// Закрываем файл
if (fclose(f) < 0) {
   // Напечатать сообщение об ошибке
   perror("Ошибка при закрытии файла");
}
```

Существует также функция **fcloseall()**, которая вызывается без аргументов и закрывает все открытые в программе потоки.

15.5 Файловый ввод/вывод в текстовом режиме

15.5.1 Форматный ввод/вывод: функции fscanf() и fprintf()

Функция **fscanf()** читает информацию из текстового файла и преобразует ее во внутреннее представление данных в памяти компьютера. Информация о количестве читаемых элементов, их типах и особенностях представления задается с помощью формата. В случае функции ввода формат – это строка, содержащая

описания одного или нескольких вводимых элементов. Форматы, используемые функцией **fscanf()**, аналогичны применяемым функцией **scanf()**. Каждый элемент формата начинается с символа процента «**%**».

Наиболее часто используемые при вводе форматы:

- «**%d**» – целое десятичное число типа **int** (**d** – от **decimal**);
- «**%lld**» – целое десятичное число типа **long long int** (**ll** – от **long long**);
- «**%lf**» – вещественное число типа **double** (**lf** – от **long float**);
- «**%Lf**» – вещественное число типа **long double**;
- «**%c**» – один символ типа **char**;
- «**%s**» – ввод строки. Из входного потока выделяется слово, ограниченное пробелами или символами перевода строки «**\n**». Слово помещается в массив символов. Конец слова отмечается нулевым байтом.

Прототип функции **fscanf()** выглядит следующим образом:

```
int fscanf(FILE *f, const char *format, ...);
```

Многоточие здесь означает, что функция имеет переменное число аргументов, большее двух, и что количество и типы аргументов, начиная с третьего, произвольны. На самом деле, фактические аргументы, начиная с третьего, должны быть указателями на вводимые переменные.

Несколько примеров использования функции **fscanf()**:

```
int n, m; double a; char c; char str[256];
FILE *f;
...
fscanf(f, "%d", &n); // Ввод целого числа
fscanf(f, "%lf", &a); // Ввод вещественного числа
fscanf(f, "%c", &c); // Ввод одного символа
fscanf(f, "%s", str); // Ввод строки (выделяется
// очередное слово из входного потока)
fscanf(f, "%d%d", &n, &m); // Ввод двух целых чисел
```

Функция **fscanf()** возвращает число успешно введенных элементов. Таким образом, возвращаемое значение всегда меньше или равно количеству процентов внутри форматной строки (которое равно числу фактических аргументов минус **2**).

Функция **fprintf()** используется для форматного вывода в файл. Данные при выводе преобразуются в их текстовое представление в соответствии с форматной строкой. Ее отличие от форматной строки, используемой в функции ввода **fscanf()**, за-

ключается в том, что она может содержать не только форматы для преобразования данных, но и обычные символы, которые записываются без преобразования в файл. Форматы, как и в случае функции **fscanf()**, начинаются с символа процента «**%**». Они аналогичны форматам, используемым функцией **fscanf()**. Небольшое отличие заключается в том, что форматы функции **fprintf()** позволяют также управлять представлением данных, например, указывать количество позиций, отводимых под запись числа, или количество цифр после десятичной точки при выводе вещественного числа.

Некоторые типичные примеры форматов для вывода:

- «**%d**» – вывод целого десятичного числа;
- «**%10d**» – вывод целого десятичного числа, для записи числа отводится **10** позиций, запись при необходимости дополняется пробелами слева;
- «**%lf**» – вывод вещественного числа типа **double** в форме с фиксированной десятичной точкой;
- «**%.3lf**» – вывод вещественного числа типа **double** с печатью трёх знаков после десятичной точки;
- «**%12.3lf**» – вывод вещественного числа типа **double** с тремя знаками после десятичной точки, под число отводится **12** позиций;
- «**%c**» – вывод одного символа;
- «**%s**» – конец строки, т.е. массива символов, конец строки задается нулевым байтом.

Прототип функции **fprintf()** выглядит следующим образом:

int fprinf(FILE *f, const char *format, ...);

Многоточие, как и в случае функции **fscanf()**, означает, что функция имеет переменное число аргументов. Количество и типы аргументов, начиная с третьего, должны соответствовать форматной строке. В отличие от функции **fscanf()**, фактические аргументы, начиная с третьего, представляют собой выводимые значения, а не указатели на переменные.

Пример. Для примера рассмотрим небольшую программу, выводящую данные в файл «**tmp.dat**» (листинг 15.1).

Листинг 15.1

```
#include <stdio.h>  // Описания функций ввода вывода
#include <math.h>  // Описания математических функций
#include <string.h> // Описания функций работы
                    // со строками
int main()
```

```
{
   int n = 4, m = 6; double x = 2.;
   char str[256] = "Print test";
   FILE *f = fopen("tmp.dat", "wt"); // Открыть файл
   if (f == 0) {                     // для записи
      perror("Не могу открыть файл для записи");
      return 1; // Завершить программу с кодом ошибки
   }
   fprintf(f, "n=%d, m=%d\n", m, n);
   fprintf(f, "x=%.4lf, sqrt(x)=%.4lf\n", x, sqrt(x));
   fprintf(f, "Строка \"%s\" содержит %d симво-
лов.\n", str, strlen(str));
   fclose(f); // Закрыть файл
   return 0;  // Успешное завершение программы
}
```

В результате выполнения этой программы в файл «**tmp.dat**» будет записан следующий текст:

```
n=6, m=4
x=2.0000, sqrt(x)=1.4142
Строка "Print test" содержит 10 символов.
```

В последнем примере форматная строка содержит внутри себя двойные апострофы. Это специальные символы, выполняющие роль ограничителей строки, поэтому внутри строки их надо экранировать (т.е. защищать от интерпретации как специальных символов) с помощью обратной косой черты ****, которая, напомним, в системе **Unix** и в языке **Си** выполняет роль защитного символа. Отметим также, что мы воспользовались стандартной функцией **sqrt()**, вычисляющей квадратный корень числа, и стандартной функцией **strlen()**, вычисляющей длину строки.

15.5.2 Понятие потока ввода или вывода

В операционной системе Unix и в других системах, использующих идеи системы Unix (например, **MS DOS** и **MS Windows**), применяется понятие потока ввода или вывода. Поток представляет собой последовательность байтов. Различают потоки ввода и вывода. Программа может читать данные из потока ввода и выводить данные в поток вывода. Программы можно запускать в конвейере, когда поток вывода первой программы является потоком ввода второй программы и т.д. Для запуска двух программ в конвейере используется символ вертикальной черты **|** между именами программ в командной строке. Например, командная строка **ab | cd | ef** означает, что поток вывода программы **ab** направляется на вход программе **cd**, а поток вывода программы

cd – на вход программе **ef**. По умолчанию, потоком ввода для программы является клавиатура, поток вывода назначен на терминал (или, как говорят программисты, на консоль). Потоки можно переправлять в файл или из файла, используя символы больше **>** и меньше **<**, которые можно представлять как воронки.

Например, командная строка **abcd > tmp.res** перенаправляет выходной поток программы **abcd** в файл «**tmp.res**», т.е. данные будут выводиться в файл вместо печати на экране терминала. Соответственно, командная строка **abcd < tmp.dat** заставляет программу **abcd** читать исходные данные из файла «**tmp.dat**» вместо ввода с клавиатуры.

Командная строка **abcd < tmp.dat > tmp.res** перенаправляет как входной, так и выходной потоки: входной назначается на файл «**tmp.dat**», выходной – на файл «**tmp.res**».

В **Си** работа с потоком не отличается от работы с файлом. Доступ к потоку осуществляется с помощью переменной типа **FILE ***. В момент начала работы Си-программы открыты три потока:

- **stdin** – стандартный входной поток – по умолчанию назначен на клавиатуру;
- **stdout** – стандартный выходной поток – по умолчанию назначен на экран;
- **srderr** – стандартный поток ошибок – по умолчанию назначен на экран.

Переменные **stdin**, **stdout** и **stderr** являются глобальными и описаны в стандартном заголовочном файле «**stdio.h**».

Операции файлового ввода-вывода могут использовать эти потоки, например, строка **fscanf(stdin, "%d", &n)** вводит значение целочисленной переменной **n** из входного потока. Строка **fprintf(stdout, "n = %d\n", n)** выводит значение переменой **n** в выходной поток. Строка

fprintf(stderr, "Ошибка при открытии файла\n");

выводит указанный текст в поток **stderr**, используемый обычно для печати сообщений об ошибках. Функция **perror()** также выводит сообщения об ошибках в поток **stderr**.

По умолчанию, стандартный выходной поток и выходной поток для печати ошибок назначены на экран терминала. Однако операция перенаправления вывода в файл **>** действует только на стандартный выходной поток. Например, в результате выполнения командной строки **abcd > tmp.res** обычный вывод программы **abcd** будет записываться в файл «**tmp.res**», а сообщения об ошибках по-прежнему будут печататься на экране

терминала. Для того чтобы перенаправить в файл «**tmp.log**» стандартный поток печати ошибок, следует использовать командную строку **abcd 2> tmp.log**. Между двойкой и символом **>** не должно быть пробелов! Двойка здесь означает номер перенаправляемого потока. Стандартный входной поток имеет номер **0**, стандартный выходной поток – номер **1**, стандартный поток печати ошибок – номер **2**. Данная команда перенаправляет только поток **stderr**, поток **stdout** по-прежнему будет выводиться на терминал. Можно перенаправить потоки в разные файлы:

abcd 2> tmp.log > tmp.res

Таким образом, существование двух разных потоков вывода позволяет при необходимости отделить мух от котлет, т.е. направить нормальный вывод и вывод информации об ошибках в разные файлы.

15.5.3 Функции scanf() и printf() ввода/вывода в стандартные потоки

Поскольку ввод из стандартного входного потока, по умолчанию назначенного на клавиатуру, и вывод в стандартный выходной поток, по умолчанию назначенный на экран терминала, используются особенно часто, библиотека функций ввода-вывода **Си** предоставляет для работы с этими потоками функции **scanf()** и **printf()**. Они отличаются от функций **fscanf()** и **fprintf()** только тем, что у них отсутствует первый аргумент, означающий поток ввода или вывода. Строка

scanf(format, ...); *// Ввод из стандартного входного потока*
эквивалентна строке

fscanf(stdin, format, ...); *// Ввод из потока* ***stdin***

Аналогично, строка

printf(format, ...); *// Вывод в стандартный выходной поток*
эквивалентна строке

fprintf(stdout, format, ...); *// Вывод в поток* ***stdout***

15.5.4 Функции текстового преобразования sscanf() и sprintf()

Стандартная библиотека ввода-вывода **Си** предоставляет две замечательные функции **sscanf()** и **sprintf()** ввода и вывода не в файл или поток, а в строку символов (т.е. массив байтов), расположенную в памяти компьютера. Мнемоника назва-

ний функций следующая: в названии функции **fscanf()** первая буква **f** означает файл (**file**), т.е. ввод производится из файла; соответственно, в названии функции **sscanf()** первая буква **s** означает строку (**string**), т.е. ввод производится из текстовой строки. (Последняя буква **f** в названиях этих функций означает форматный). Первым аргументом функций **sscanf()** и **sprintf()** является строка (т.е. массив символов, ограниченный нулевым байтом), из которой производится ввод или в которую производится вывод. Эта строка как бы стоит на месте файла в функциях **fscanf()** и **fprintf()**.

Функции **sscanf()** и **sprintf()** удобны для преобразования данных из текстового представления во внутреннее и обратно. Например, в результате выполнения фрагмента

```
char txt[256] = "-135.76";
double x;
sscanf(txt, "%lf", &x);
```

текстовая запись вещественного числа, содержащаяся в строке **txt**, преобразуется во внутреннее представление вещественного числа, результат записывается в переменную **x**. Обратно, при выполнении фрагмента

```
char txt[256];
int x = 12345;
sprintf(txt, "%d", x);
```

значение целочисленной переменной **x** будет преобразовано в текстовую форму и записано в строку **txt**, в результате строка будет содержать текст «**12345**», ограниченный нулевым байтом.

Для преобразования данных из текстового представления во внутреннее в стандартной библиотеке **Си** имеются также функции **atoi()** и **atof()** с прототипами

```
int atoi(const char *txt);    // текст => int
double atof(const char *txt); // текст => double
```

Функция **atoi()** преобразует текстовое представление целого числа типа **int** во внутреннее. Соответственно, функция **atof()** преобразует текстовое представление вещественного числа типа **double**. Мнемоника имен следующая:

- **atoi** означает «**character to integer**»;
- **atof** означает «**character to float**».

В последнем случае **float** следует понимать как плавающее, т.е. вещественное, число, имеющее тип **double**, а вовсе не **float**! Тип **float** является атавизмом и практически не используется.

Прототипы функций **atoi()** и **atof()** описаны в стандартном заголовочном файле «**stdlib.h**», а не «**stdio.h**», поэтому при их использовании надо подключать этот файл:

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

Вообще-то, это можно делать всегда, поскольку «**stdlib.h**» содержит описания многих полезных функций, например, функции завершения программы **exit()**, генератора случайных чисел **rand()** и др.

Отметим, что аналогов функции **sprintf()** для обратного преобразования из внутреннего в текстовое представление в стандартной библиотеке **Си** нет.

15.6 Файловый ввод/вывод в бинарном режиме

15.6.1 Функции бинарного чтения и записи fread() и fwrite()

После того как файл открыт, можно читать информацию из файла или записывать информацию в файл. Рассмотрим функции **бинарного** чтения и записи **fread()** и **fwrite()**. Они называются бинарными потому, что не выполняют никакого преобразования информации при вводе или выводе: информация хранится в файле как последовательность байтов ровно в том виде, в котором она хранится в памяти компьютера.

Функция чтения **fread()** имеет следующий прототип:

```
size_t fread(
   char *buffer,    // Массив для чтения данных
   size_t elemSize, // Размер одного элемента
   size_t numElems, // Число элементов для чтения
   FILE *f          // Указатель на структуру FILE
);
```

Здесь **size_t** определен как беззнаковый целый тип в системных заголовочных файлах. Функция пытается прочесть **numElems** элементов из файла, который задается указателем **f** на структуру **FILE**, размер каждого элемента равен **elemSize**. Функция возвращает реальное число прочитанных элементов, которое может быть меньше, чем **numElems**, в случае конца файла или ошибки чтения. Указатель **f** должен быть возвращен функцией **fopen()** в результате успешного открытия файла.

Пример. Использование функции **fread()** представлено в листинге 15.2.

Листинг 15.2

```
#include <stdio.h>

int main()
{
   FILE *f;
   double buff[100];
   size_t res;
...
   f = fopen("tmp.dat", "rb"); // Открываем файл
   if (f == 0)
   { // При ошибке открытия файла напечатать
     // сообщение об ошибке
      perror("Не могу открыть файл для чтения");
      exit(1);  // завершить работу с кодом 1
   }
// Пытаемся прочесть 100 вещественных чисел из файла
   res = fread(buff, sizeof(double), 100, f);
   // res равно реальному количеству прочитанных чисел
   printf("Результат: %d",res);
   return 0;
}
```

В этом примере файл «**tmp.dat**» открывается на чтение как бинарный, из него читается **100** вещественных чисел размером **8** байт каждое. Функция **fread()** возвращает реальное количество прочитанных чисел, которое меньше или равно, чем **100**.

Функция **fread()** читает информацию в виде потока байтов и в неизменном виде помещает ее в память. Следует различать **текстовое** представление чисел и их **бинарное представление!** В приведенном выше примере числа в файле должны быть записаны в **бинарном виде**, а не в виде текста. Для текстового ввода чисел надо использовать функции **ввода по формату**.

Внимание! Открытие файла как текстового с помощью функции **fopen()**, например,

```
FILE *f = fopen("tmp.dat", "rt");
```

вовсе не означает, что числа при вводе с помощью функции **fopen()** будут преобразовываться из **текстовой** формы в **бинарную!** Из этого следует только то, что в операционных системах, в которых строки текстовых файлов разделяются парами символов «**\r\n**» (они имеют названия **CR** и **LF** – возврат каретки и продергивание бумаги, **Carriage Return** и **Line Feed**), при вводе такие пары символов заменяются на один символ «**\n**»

(продергивание бумаги). Обратно, при выводе символ «**\n**» заменяется на пару «**\r\n**». Такими операционными системами являются **MS DOS** и **MS Windows**. В системе **Unix** строки разделяются одним символом «**\n**» (отсюда проистекает обозначение «**\n**», которое расшифровывается как **new line**). Таким образом, внутреннее представление текста всегда соответствует системе Unix, а внешнее – реально используемой операционной системе. Отметим также, что создатели операционной системы компьютеров Apple Macintosh выбрали, чтобы жизнь не казалась скучной, третий, отличный от двух предыдущих, вариант: текстовые строки разделяются одним символом «**\r**» возврат каретки!

Такое представление текстовых файлов восходит к тем уже далеким временам, когда еще не было компьютерных мониторов и для просмотра текста использовались электрифицированные пишущие машинки или посимвольные принтеры. Текстовый файл фактически представлял собой программу печати на пишущей машинке и, таким образом, содержал команды возврата каретки и продергивания бумаги в конце каждой строки.

Функция бинарной записи в файл **fwrite()** аналогична функции чтения **fread()**. Она имеет следующий прототип:

```
size_t fwrite(
   char *buffer,    // Массив записываемых данных
   size_t elemSize, // Размер одного элемента
   size_t numElems, // Число записываемых элементов
   FILE *f          // Указатель на структуру FILE
);
```

Функция возвращает число реально записанных элементов, которое может быть меньше, чем **numElems**, если при записи произошла ошибка – например, не хватило свободного пространства на диске.

Пример. Использование функции **fwrite()** представлено в листинге 15.3.

Листинг 15.3

```
#include <stdio.h>

int main()
{
   FILE *f;
   double buff[100];
   size_t res;
```

```
    f = fopen("tmp.res", "wb"); // Открываем файл
                                «tmp.res»
    if (f == 0)
    { // При ошибке открытия файла напечатать
      // сообщение об ошибке
        perror("Не могу открыть файл для записи");
        exit(1); // Завершить работу программы с кодом 1
    }
    // Записываем 100 вещественных чисел в файл
    res = fwrite(buff, sizeof(double), 100, f);
    // В случае успеха res = 100
    printf("Результат: %d",res);
    return 0;
}
```

15.6.2 Пример: подсчет числа символов и строк в текстовом файле

В качестве содержательного примера использования рассмотренных выше функций файлового ввода приведем программу, которая подсчитывает число символов и строк в текстовом файле.

Программа сначала вводит имя файла с клавиатуры. Для этого используется функция **scanf()** ввода по формату из входного потока, для ввода строки применяется формат «**%s**». Затем файл открывается на чтение как бинарный (это означает, что при чтении не будет происходить никакого преобразования разделителей строк). Используя в цикле функцию чтения **fread()**, мы считываем содержимое файла порциями по **512** байтов, каждый раз увеличивая суммарное число прочитанных символов. После чтения очередной порции сканируется массив прочитанных символов и подсчитывается число символов «**\n**» продергивания бумаги, которые записаны в концах строк текстовых файлов как в системе **Unix**, так и в **MS DOS** или **MS Windows**. В конце закрывается файл и печатается результат (листинг 15.4).

Листинг 15.4

```
// Файл «wc.cpp»
// Подсчет числа символов и строк в текстовом файле

#include <stdio.h>  // Описания функций ввода-вывода
#include <stdlib.h> // Описание функции exit()

int main()
```

```
{
   char fileName[256]; // Путь к файлу
   FILE *f;          // Структура, описывающая файл
   char buff[512];    // Массив для ввода символов
   size_t num;        // Число прочитанных символов
   int numChars = 0;  // Суммарное число символов = 0
   int numLines = 0;  // Суммарное число строк = 0
   int i;             // Переменная цикла

   printf("Введите имя файла: ");
   scanf("%s", fileName);

   f = fopen(fileName, "rb");
   // Открываем файл на чтение
   if (f == 0)
   { // При ошибке открытия файла напечатать
     // сообщение об ошибке
      perror("Не могу открыть файл для чтения");
      exit(1);  // Закончить работу программы с кодом 1
                //  ошибочного завершения
   }
   while ((num = fread(buff, 1, 512, f)) > 0)
   {
      // Читаем блок из 512 символов
      // num - число реально прочитанных символов
      // Цикл продолжается, пока num > 0
      numChars += num; // Увеличиваем число символов
      // Подсчитываем число символов перевода строки
      for (i = 0; i < num; ++i)
         if (buff[i] == '\n')
            ++numLines; // Увеличиваем число строк
   }
   fclose(f);
   // Печатаем результат
   printf("Число символов в файле = %d\n",
numChars);
   printf("Число строк в файле = %d\n", numLines);
   return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы: она применяется к собственному тексту, записанному в файле «**wc.cpp**».

```
Введите имя файла: wc.cpp
Число символов в файле = 1635
Число строк в файле = 50
```

15.7 Другие полезные функции ввода-вывода

Стандартная библиотека ввода-вывода **Си** содержит ряд других полезных функций ввода-вывода. Отметим некоторые из них.

Посимвольный ввод-вывод

int fgetc(FILE *f);	ввести символ из потока **f**
int fputc(int c, FILE *f);	вывести символ в поток **f**

Построковый ввод-вывод

char *fgets(char *line,int size, FILE *f);	ввести строку из потока **f**
char *fputs(char *line, FILE *f);	вывести строку в поток **f**

Позиционирование в файле

int fseek(FILE *f, long offset, int whence);	установить текущую позицию в файле **f**
long ftell(FILE *f);	получить текущую позицию в файле **f**
int feof(FILE *f);	проверить, достигнут ли конец файла **f**

Функция **fgetc()** возвращает код введенного символа или константу **EOF** (определенную как минус единицу) в случае конца файла или ошибки чтения. Функция **fputc()** записывает один символ в файл. При ошибке **fputc()** возвращает константу **EOF** (т.е. отрицательное значение), в случае удачи – код выведенного символа **c** (неотрицательное значение).

В качестве примера использования функции **fgetc()** перепишем рассмотренную ранее программу **wc**, подсчитывающую число символов и строк в текстовом файле (листинг 15.5).

Листинг 15.5

```
// Файл «wc1.cpp»
// Подсчет числа символов и строк в текстовом файле
// с использованием функции чтения символа fgetc()

#include <stdio.h>  // Описания функций ввода-вывода

int main()
{
   char fileName[256]; // Путь к файлу
   FILE *f;          // Структура, описывающая файл
   int c;            // Код введенного символа
   int numChars = 0;  // Суммарное число символов = 0
   int numLines = 0;  // Суммарное число строк = 0

   printf("Введите имя файла: ");
   scanf("%s", fileName);
```

```
    f = fopen(fileName, "rb"); // Открываем файл
    if (f == 0)
    { // При ошибке открытия файла напечатать
      // сообщение об ошибке
       perror("Не могу открыть файл для чтения");
       return 1; // закончить работу программы с кодом 1
    }
    while ((c = fgetc(f)) != EOF)
    { // Читаем символ. Цикл продолжается, пока c ≠ -1
      // (конец файла)
       ++numChars; // Увеличиваем число символов
       // Подсчитываем число символов перевода строки
       if (c == '\n')
          ++numLines; // Увеличиваем число строк
    }
    fclose(f);
    // Печатаем результат
    printf("Число символов в файле = %d\n",
numChars);
    printf("Число строк в файле = %d\n", numLines);
    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы **wc1** в применении к собственному тексту, записанному в файле «**wc1.cpp**»:

```
Введите имя файла: wc1.cpp
Число символов в файле = 1334
Число строк в файле = 44
```

Функция **fgets()** с прототипом

```
char *fgets(char *line, int size, FILE *f);
```

выделяет из файла или входного потока **f** очередную строку и записывает ее в массив символов **line**. Второй аргумент **size** указывает размер массива для записи строки. Максимальная длина строки на единицу меньше, чем **size**, поскольку всегда в конец считанной строки добавляется нулевой байт. Функция сканирует входной поток до тех пор, пока не встретит символ перевода строки «**\n**» или пока число введенных символов не станет равным **(size-1)**. Символ перевода строки «**\n**» также записывается в массив непосредственно перед терминирующим нулевым байтом. Функция возвращает указатель **line** в случае успеха или нулевой указатель при ошибке или конце файла.

Построчный вывод выполняется функцией **fputs()**:

char *fputs(char *line, FILE *f);

которая передает данные из массива символов с адресом **line** в файл **f** до тех пор, пока не будет обнаружен завершающий строку нулевой байт.

Функция **fputs()** возвращает отрицательное значение, обычно **EOF**, если обнаружена ошибка при выводе данных. Значение, которое возвращается после успешного вызова функции, зависит от реализации, но обычно отличается от **EOF**.

Слияние файлов с помощью построчного ввода-вывода (листинг 15.6).

Листинг 15.6

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

int main()
{
   FILE *sp, *tp;
   char *line;

   if ((sp=fopen("1.dat","r"))==NULL)
   {
      fprintf(stderr, "Ошибка при открытии файла
1.dat\n");
      exit(1);
   }
   if ((tp=fopen("2.dat","a"))==NULL)
   {
      fprintf(stderr, "Ошибка при открытии файла
2.txt\n");
      exit(1);
   }
   while (fgets(line,sizeof(line)-1,sp))
      fputs(line,tp);
   fclose(sp);
   fclose(tp);
   return 0;
}
```

Вместо считывания по одному символу за прием эта программа считывает полную строку (до 80 символов). Функция **fputs()** записывает в файл символы строки **line** до тех пор, пока не обнаружит нулевой байт.

При выполнении файловых операций исполняющая система поддерживает указатель текущей позиции в файле. При чтении или записи **n** байтов указатель текущей позиции увеличивается на **n**; таким образом, чтение или запись происходят последовательно. Библиотека ввода-вывода **Си** предоставляет, однако, возможность нарушать эту последовательность путем позиционирования в произвольную точку файла. Для этого используется стандартная функция **fseek()** с прототипом

int fseek(FILE *f, long offset, int whence);

Первый аргумент **f** функции определяет файл, для которого производится операция позиционирования. Второй аргумент **offset** задает смещение в байтах, оно может быть как положительным, так и отрицательным. Третий аргумент **whence** указывает, откуда отсчитывать смещение. Он может принимать одно из трех значений, заданных как целые константы в стандартном заголовочном файле «**stdio.h**»:

- **SEEK_CUR** – смещение отсчитывается от текущей позиции;
- **SEEK_SET** – смещение отсчитывается от начала файла;
- **SEEK_END** – смещение отсчитывается от конца файла.

Например, фрагмент

fseek(f, 0, SEEK_SET);

устанавливает текущую позицию в начало файла.

Фрагмент

fseek(f, -4, SEEK_END);

устанавливает текущую позицию в четырех байтах перед концом файла.

Наконец, фрагмент

fseek(f, 12, SEEK_CUR);

продвигает текущую позицию на **12** байтов вперед.

Отметим, что смещение может быть положительным даже при использовании константы **SEEK_END** (т.е. при позиционировании относительно конца файла): в этом случае при следующей записи размер файла соответственно увеличивается.

Функция возвращает нулевое значение в случае успеха и отрицательное значение **EOF** (равное **-1**) при неудаче – например, если указанное смещение некорректно при заданной операции или если файл или поток не позволяет выполнять прямое позиционирование.

Узнать текущую позицию относительно начала файла можно с помощью функции **ftell()** с прототипом

```
long ftell(FILE *f);
```

Функция **ftell()** возвращает текущую позицию (неотрицательное значение) в случае успеха или отрицательное значение **-1** при неудаче (например, если файл не разрешает прямое позиционирование).

Наконец, узнать, находится ли текущая позиция в конце файла, можно с помощью функции **feof()** с прототипом

```
int feof(FILE *f);
```

Она возвращает ненулевое значение (т.е. истину), если конец файла достигнут, и нулевое значение (т.е. ложь) в противном случае. Например, в следующем фрагменте в цикле проверяется, достигнут ли конец файла, и, если нет, считывается очередной байт:

```
FILE *f;
int c;
...
while (!feof(f)) {    // цикл пока не конец файла
   c = fgetc(f); //  прочесть очередной байт
   ...
}                     // конец цикла
```

15.8 Низкоуровневый ввод/вывод

Самый низкий уровень ввода/вывода не предусматривает ни какой-либо буферизации, а по существу является непосредственным обращением к операционной системе. Весь ввод и вывод осуществляется двумя функциями: **read()** и **write()**. Первым аргументом обеих функций является дескриптор файла. Вторым аргументом является буфер в вашей программе, откуда или куда должны поступать данные. Третий аргумент – это число подлежащих пересылке байтов. Обращения к этим функциям имеют вид:

```
n_read=read(fd,buf,n);
n_written=write(fd,buf,n);
```

При каждом обращении возвращается счетчик байтов, указывающий фактическое число переданных байтов. При чтении возвращенное число байтов может оказаться меньше, чем запрошенное число. Возвращенное нулевое число байтов означает конец файла, а «**-1**» указывает на наличие какой-либо ошибки.

При записи возвращенное значение равно числу фактически записанных байтов; несовпадение этого числа с числом байтов, которое предполагалось записать, обычно свидетельствует об ошибке.

Количество байтов, подлежащих чтению или записи, может быть совершенно произвольным. Двумя самыми распространенными величинами являются «1», что означает передачу одного символа за обращение (т.е. без использования буфера), и «512», что соответствует физическому размеру блока на многих периферийных устройствах. Этот последний размер будет наиболее эффективным, но даже ввод или вывод по одному символу за обращение не будет слишком дорогим.

Пример. Копирование ввода на вывод. В **Unix**-подобной системе эта программа будет копировать что угодно куда угодно, потому что ввод и вывод могут быть перенаправлены на любой файл или устройство (листинг 15.7).

Листинг 15.7

```
#include<stdio.h>

#define BUFSIZE 512

void main() /*copy input to output*/
{
   char buf[BUFSIZE];
   int n;
   while((n=read(0,buf,BUFSIZE))>0)
      write(1,buf,n);
}
```

Если размер файла не будет кратен **BUFSIZE**, то при очередном обращении к **read()** будет возвращено меньшее число байтов, которые затем записываются с помощью **write()**; при следующем после этого обращении к **read()** будет возвращен нуль. Выход осуществляется по нажатию сочетания клавиш «**Ctrl+Z**».

15.8.1 Открытие, создание, закрытие и удаление

Во всех случаях, если только не используются определенные по умолчанию стандартные файлы ввода, вывода и ошибок, вы должны явно открывать файлы, чтобы затем читать из них или писать в них. Для этой цели существуют две функции: **open()** и **creat()**.

Функция **open()** весьма сходна с функцией **fopen()**, рассмотренной выше, за исключением того, что вместо возвраще-

ния указателя файла она возвращает дескриптор файла, который является просто целым типа **int**.

```
int fd;
fd=open(name,rwmode);
```

Как и в случае **fopen()**, аргумент **name** является символьной строкой, соответствующей внешнему имени файла. Однако аргумент, определяющий режим доступа, отличен: **rwmode** равно: **0** – для чтения, **1** – для записи, **2** – для чтения и записи. Если происходит какая-то ошибка, функция **open()** возвращает «**-1**»; в противном случае она возвращает неотрицательный дескриптор файла.

Попытка открыть файл, который не существует, является ошибкой. Функция **creat()** предоставляет возможность создания новых файлов или перезаписи старых. В результате обращения:

```
fd=creat(name,pmode);
```

возвращает дескриптор файла, если оказалось возможным создать файл с именем **name**, и «**-1**» в противном случае. Создание файла, который уже существует, не является ошибкой: **creat()** усечет его до нулевой длины.

Если файл ранее не существовал, то **creat()** создает его с определенным режимом защиты, специфицируемым аргументом **pmode**. В системе файлов **Unix**-подобных систем с файлом связываются девять битов защиты информации, которые управляют разрешением на чтение, запись и выполнение для владельца файла, для группы владельцев и для всех остальных пользователей. Таким образом, трехзначное восьмеричное число наиболее удобно для записи режима защиты. Например, число **0755** свидетельствует о разрешении на чтение, запись и выполнение для владельца и о разрешении на чтение и выполнение для группы и всех остальных.

Существует ограничение (обычно **15-25**) на количество файлов, которые программа может иметь открытыми одновременно. В соответствии с этим любая программа, собирающаяся работать со многими файлами, должна быть подготовлена к повторному использованию дескрипторов файлов. Процедура **close()** прерывает связь между дескриптором файла и открытым файлом и освобождает дескриптор файла для использования с другим файлом. Завершение выполнения программы через **exit()** или в результате возврата из головной функции приводит к закрытию всех открытых файлов.

Функция удаления **unlink(filename)** удаляет из системы файл с именем **filename** (Точнее, удаляет имя **filename**, файл удаляется, если на него не остается ссылок под другими именам).

15.8.2 Произвольный доступ – lseek()

Обычно при работе с файлами ввод и вывод осуществляется последовательно: при каждом обращении к функциям **read()** и **write()** чтение или запись начинаются с позиции, непосредственно следующей за предыдущей обработанной. Но при необходимости файл может читаться или записываться в любом произвольном порядке. Обращение к системе с помощью функции **lseek()** позволяет передвигаться по файлу, не производя фактического чтения или записи. В результате обращения

```
lseek(fd,offset,origin);
```

текущая позиция в файле с дескриптором **fd** передвигается на позицию **offset** (смещение), которая отсчитывается от места, указываемого аргументом **origin** (начало отсчета). Последующее чтение или запись будут теперь начинаться с этой позиции. Аргумент **offset** имеет тип **long**; **fd** и **origin** имеют тип **int**. Аргумент **origin** может принимать значения **0**, **1** или **2**, указывая на то, что величина **offset** должна отсчитываться соответственно от начала файла, от текущей позиции или от конца файла. Например, чтобы дополнить файл, следует перед записью найти его конец:

```
lseek(fd,0l,2);
```

чтобы вернуться к началу, можно написать:

```
lseek(fd,0l,0);
```

Обратите внимание на аргумент **0l**; его можно было бы записать и в виде **(long) 0**.

Функция **lseek()** позволяет обращаться с файлами примерно так же, как с большими массивами, только ценой более медленного доступа.

Пример. Функция, считывающая любое количество байтов, начиная с произвольного места в файле (листинг 15.8).

Листинг 15.8

```
/*читать n байтов с позиции pos в buf */
int get(int fd, long pos,char *buf,int n)
{
   lseek(fd,pos,0);   /*переход на позицию pos */
   return (read(fd,buf,n));
}
```

16 СТРУКТУРЫ ДАННЫХ И АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

В основе современных информационных технологий лежат методы формализации знаний, относящихся к той или иной предметной области. Формализованные знания включают выявленные в результате анализа предметной области совокупности данных, под которыми понимаются предметы, факты, явления, идеи, события, процессы и т.п. и связи между ними (понятийные знания), а также описание алгоритмов выполняемых над данными действий (алгоритмические знания). В дальнейшем при построении информационных моделей среди множества взаимосвязей данных отбираются наиболее информативные, наиболее полно отражающие состояние предметной области. В конечном итоге получается упрощенная модель предметной области, в которой более простые (элементарные) данные объединяются по определенным признакам в более сложные и определяется набор выполняемых над ними действий. В программировании для представления подобных объединений данных используются структуры данных.

Структуры данных – это сложные данные, построенные из более простых или элементарных данных (возможно различных типов), объединенных с помощью определенных языковых конструкций.

Элементарные данные (элементарный объект данных) – логически неделимая поименованная порция данных.

Различают **функциональные**, **рекурсивные** и **теоретико-множественные** структуры данных.

16.1 Функциональные структуры данных

Функциональную структуру данных (ФСД) математически можно определить как отображение (соответствие) из некоторого множества **A**, называемого областью расположения структуры данных, в другое множество **B**, называемое областью значений структуры данных.

К функциональным структурам данных (ФСД) относятся *многомерные массивы*, *записи* и *файлы*.

В общем случае *файл* как совокупность записей, имеющих общую область использования, следует отнести к многоуровневым ФСД, т.е. ФСД, элементами которых являются ФСД, так же, как и массив массивов, массив записей или запись с полями регулярного или комбинированного типа.

16.2 Рекурсивные структуры данных

Рекурсивные структуры данных (РСД) определяются как структуры данных, в основе представления которых лежит понятие графа.

Граф – математическая модель связей между объектами произвольной природы, представляющая собой набор точек на плоскости (вершины графа), которые могут соединяться направленными или ненаправленными линиями (ребра графа). Если ребра графа являются направленными, т.е. для каждой вершины можно указать, какие ребра в нее входят и какие выходят, то граф называется *ориентированным*, в противном случае – *неориентированным*. Две вершины графа соединены *путем*, если из одной вершины можно попасть в другую, пройдя по ребрам. Граф называется *связным*, если любые две его вершины соединены некоторым путем. Состоящий из различных ребер замкнутый путь называется *циклом*. Ребро, соединяющее вершину с ней самой, называется *петлей*.

К РСД относятся различные списковые структуры, деревья, сетевые и фреймовые структуры и т.д.

Список можно определить как ориентированный или неориентированный граф, в любую вершину которого входит одно или два ребра.

Списковая структура данных – иерархическая система хранения данных в памяти ЭВМ, в которой данные рассматриваются как списки, элементы которых в свою очередь также могут быть списками (такие элементы называются подсписками) и т.д. Элементы списков и подсписков могут располагаться в памяти произвольно, но каждый из элементов содержит указатель на место расположения следующего и/или предыдущего элемента.

Дерево в теории графов – ориентированный или неориентированный граф без циклов с выделенной вершиной (корень дерева), из которой существует путь в любую другую вершину, причем этот путь – единственный. В каждую вершину дерева входит только одно ребро, а выходить может несколько. Различные виды деревьев широко используются в теории алгоритмов, теории оптимизации, математической статистике, теории надежности, теории распознавания образов и т.д.

Сетевая структура данных – структура, которая может быть представлена ориентированным графом, в любую вершину которого может входить более одного ребра. Сетевые структуры используются в различных приложениях для представления **n**-арных связей между объектами, т.е. когда данным одного типа

ставится в соответствие множество данных другого типа и наоборот. Сетевая модель данных представляет собой наиболее универсальную модель структурирования фактографических данных.

Фреймовая структура данных (от англ. **frame** – каркас, рамка) – структура, используемая для представления знаний в ЭВМ. Фрейм можно представить в виде графа, вершины которого распределены по уровням. Верхний уровень фрейма фиксирован и содержит знания, всегда истинные для ситуаций, которые представляет данный фрейм. На нижележащих уровнях расположены так называемые слоты, которые могут заполняться подходящими данными в процессе «приспособления» фрейма к конкретной ситуации. Родственные фреймы могут связываться в систему фреймов, системы фреймов – в информационно-поисковую сеть.

16.3 Теоретико-множественные структуры данных

Теоретико-множественные структуры данных (ТМСД) – наиболее общий тип структур данных, основанный на математическом понятии множества. В математике под множеством понимается любое объединение в одно целое некоторых определенных и различных между собой объектов, находящихся в определенных отношениях между собой или с элементами других множеств.

16.4 Абстрактные типы данных

В процедурных языках программирования, таких как **Си**, достаточно широко представлены встроенные структурные типы данных, относящиеся к ФСД, при этом допустимые структуры можно представить как композиции нескольких порождающих типов структур: элемент, массив, запись и файл. Построение структур данных других типов сопряжено с разработкой соответствующих абстрактных типов данных.

Абстрактный тип данных (АТД) – тип данных, рассматриваемый независимо от способов его представления или реализации средствами языка программирования и представляющий собой математическую модель взаимосвязи объектов некоторой предметной области с определенными для них основными операциями.

Решение любой прикладной задачи в программировании с применением того или иного АТД начинается с выбора соответствующей математической модели такой, например, как отображение, граф или множество. Далее разрабатывается АТД с под-

ходящим набором операций, и алгоритм решения задачи описывается в терминах этого АТД. Как правило, такое описание выполняется на неформальном языке (псевдоязыке), представляющем собой смесь фраз естественного языка и базовых управляющих структур того языка программирования, на котором в конечном итоге будет представлен разрабатываемый алгоритм.

Следующий этап – реализация алгоритма на конкретном языке программирования, т.е. переход от абстрактно-логического представления алгоритма в терминах разработанного АТД к представлению в терминах типов данных, операторов и структур, допустимых в данном языке программирования. Для этого необходимо, во-первых, отобразить элементы АТД на одну из синтаксических конструкций языка и, во-вторых, реализовать на этом языке разработанные для данного АТД операции.

Для представления элементов АТД используются структурные типы данных, а объединение элементов АТД в структуры производится путем создания поименованных групп элементов или с помощью ссылок, реализуемых посредством указателей, когда каждый элемент АТД содержит не только определенную порцию данных, но и указание на место расположения связанных с ним элементов.

В рамках одной и той же математической модели можно рассматривать разные операции, поэтому любой АТД фактически определяется своим набором операций. Выбор тех или иных операций будет зависеть от поставленной задачи и способа структурирования элементов АТД, но в большинстве случаев можно говорить о типовых операциях, включающих операции поиска, вставки и удаления элементов.

16.5 Списковые структуры

16.5.1 АТД «Список»

Список как структура данных представляет собой набор однотипных элементов, связанных друг с другом посредством ссылок.

По количеству ссылок у элементов списки делятся на **односвязные** (одна ссылка – на предыдущий или последующий элемент) и **двусвязные** (две ссылки – на предыдущий и последующий элементы). И те и другие могут быть **линейными** или **кольцевыми**. В линейном списке первый и последний элементы не связаны друг с другом, а в кольцевом списке первый элемент может ссылаться на последний и/или наоборот. В практике программирования наиболее часто используются двусвязные

кольцевые списки, в которых последний элемент ссылается на первый как на последующий, а первый элемент – на последний как на предыдущий.

Списки, как и ФСД, можно объединять, образуя более сложные структуры того же типа, т.е. можно создавать многоуровневые структуры данных типа списка списков. Такие объединения списков имеют иерархическую структуру и поэтому называются **иерархическими списками.**

Иерархические списки применимы в тех случаях, когда взаимосвязи между реальными объектами имеют иерархическую структуру, а именно: для решения задач структурного анализа сложных систем, в частности для описания организационной структуры социальных систем, для разработки различного рода каталогов, справочных систем и т.п.

Списковые структуры представляют собой модели взаимосвязей объектов реальной среды, объединяемых по наличию некоторого общего для всех объектов свойства. Обладая множеством различных свойств, такие объекты могут одновременно входить в несколько объединений. Если представлять объекты в виде элементов списков, то это означает, что одни и те же элементы могут включаться в несколько списков одновременно. Например, на базе одного и того же множества записей, содержащих анкетные данные граждан, вставших на учет в службе занятости с целью поиска работы, можно сформировать несколько списков: по профессии, по уровню квалификации, по стажу работы в той или иной отрасли, по районам города, которым отдается предпочтение при выборе места работы, и т.д.

Избавиться от неизбежного в таких случаях избыточного дублирования данных позволяют **ассоциативные списки**. Совокупность списков, создаваемых на базе одного общего набора записей, называется **информационной сетью** и представляет собой наиболее общую форму отображения множества взаимосвязей реальных объектов.

Для представления элементов АТД «Список» используются записи, в общем случае состоящие из обязательных информационной и ссылочной частей и необязательной справочной части. В информационной части содержатся поля, значения которых описывают определенные свойства элементов списка. Среди этих полей, как правило, выделяется одно или несколько полей, которые однозначно идентифицируют данный элемент. Такие поля называются ключевыми или просто – ключом. В ссылочной части указываются адреса следующего и/или предыдущего эле-

мента списка. Справочная часть предназначена для хранения дополнительной информации об элементах списка.

В начале списка обычно располагается особый головной элемент, содержащий сведения о списке в целом, например, количество элементов, минимальное и максимальное значения ключа и т.д., и, по крайней мере, ссылку на первый элемент списка.

К основным операциям над списками относятся:

- поиск заданного элемента списка;
- включение нового элемента в список;
- исключение элемента из списка.

16.5.2 Линейные списки

Рассмотрим реализацию линейного односвязного списка, в котором информационная часть элементов списка состоит из некоторого поля, тип которого определен как **VAL**, и ссылку, направленную таким образом, что каждый элемент, кроме последнего, ссылается на последующий. Для определенности примем в качестве типа **VAL** стандартный тип **int**.

```
typedef int VAL;
/* Определение некоторого типа данных VAL */
```

Тип элементов списка можно определить следующим образом:

```
typedef struct item
{
   VAL val;      /* Поле значения элемента */
   struct item *next;     /* Ссылочная часть */
} ITEM;
```

В структуре головного элемента списка обязательно присутствует ссылочное поле, указывающее на первый элемент списка. Кроме того, головной элемент может содержать дополнительные поля, облегчающие работу со списком: например, количество элементов в списке, ссылку на последний элемент списка и др. Определим структуру головного элемента, состоящую из ссылочного поля, указывающего на первый элемент списка, и счетчика, хранящего количество элементов списка в текущий момент времени:

```
typedef struct
{
   unsigned num; /* Количество элементов в списке */
   ITEM *first;  /* Ссылка на первый элемент */
} HITEM;
```

Более наглядно список можно изобразить следующим образом, указанным на рисунке 16.1.

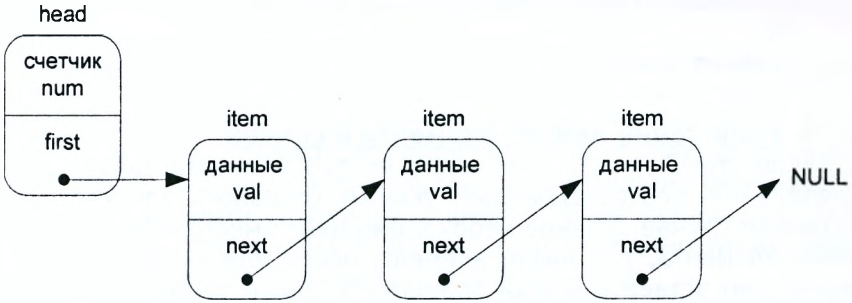


Рисунок 16.1 – Линейные списки

В памяти элементы списка располагаются так, как указано на рисунке 16.2.

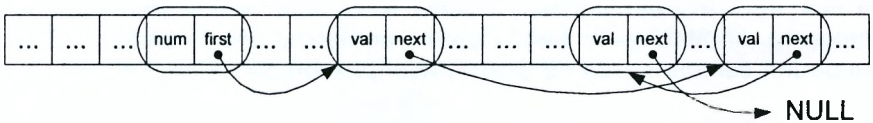


Рисунок 16.2 – Расположение линейных списков в памяти

Теперь рассмотрим функции, реализующие операции над элементами списка описанной выше структуры:

1. Создание пустого списка

Пустой список – это список, не содержащий никаких элементов, кроме головного. Таким образом, чтобы создать пустой список, надо создать его головной элемент с нулевой ссылкой (листинг 16.1).

Листинг 16.1

```
/* Создание пустого списка.
   Параметры: нет.
   Возвращаемое значение:
   при успешном завершении – указатель на созданный
   головной элемент списка,
   иначе – NULL */

HITEM *EmptyList()
{
   HITEM *head = NULL;
   if ((head = (HITEM *) malloc(sizeof(HITEM))) != NULL)
```

```
    {
      head->num = 0;
      head->first = NULL;
    }
    return head;
}
```

2. Включение нового элемента в список

Если список пуст, включение нового элемента сводится к определению соответствующей ссылки головного элемента. В противном случае сначала необходимо найти место в списке для нового элемента, т.е. найти элемент, после или перед которым необходимо вставить новый элемент. В общем случае расположение элементов в списке может быть произвольным. Если же элементы списка упорядочены в порядке убывания или возрастания их значений, каждый новый элемент займет вполне определенное место, а поиск элементов в таком списке будет выполняться несколько быстрее. После того, как место для нового элемента определено, необходимо вставить его в список. Эта операция выполняется по схеме, изображенной на рисунке 16.3.

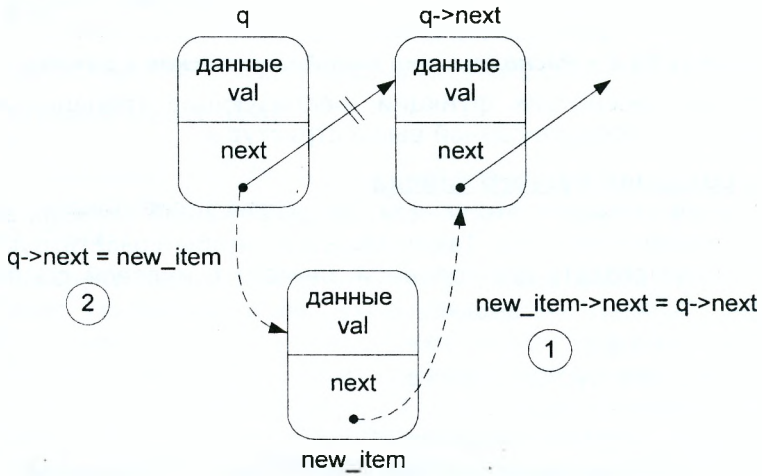


Рисунок 16.3 – Схема вставки элемента в линейный односвязный список

Действия на рисунке 16.3 пронумерованы. Если их выполнять в другом порядке, будет потерян указатель на вторую часть списка, и она окажется недоступна (ситуация, известная как "потеря хвоста").

Здесь указатель **q** является указателем на текущий элемент списка, **q->next** – следующий элемент. С помощью вспомога-

тельного указателя **new_item** – создается новый элемент, в ссылочной части которого записывается адрес элемента **q->next**, а ссылка элемента **q** перенаправляется на вновь созданный элемент.

Если новый элемент должен стать первым элементом списка, в роли элемента **q** будет выступать головной элемент. Если новый элемент добавляется в конец списка, его ссылочная часть устанавливается в **NULL**-значение, поскольку элемент **q->next** отсутствует. И в том и в другом случае необходимо переопределить счетчик **num** головного элемента. Если добавление элемента происходит в начало списка, то необходимо также переопределить ссылочное поле головного элемента.

В листинге 16.2 представлен исходный код функции, реализующей операцию включения элемента в список, элементы которого располагаются в порядке возрастания их значений.

Листинг 16.2

```
/* Включение элемента в упорядоченный список.
   Параметры: h – указатель на головной элемент;
   new_val – значение нового элемента;
   Возвращаемое значение:
   при успешном завершении – указатель на новый
   элемент списка, иначе – NULL */

ITEM *add_item (HITEM *h, VAL new_val)
{
   ITEM *q = h->first, *new_item;
   if ((new_item = (ITEM *) malloc(sizeof(ITEM))) ==
NULL)
      return NULL;
   new_item->val = new_val);
   if (h->num == 0)   /* Если список пустой */
   {
      h->first = new_item;
      new_item->next = NULL;
   }
   else    /* Если список не пустой */
   {     /* Поиск места для нового элемента */
      while ((q->next->val < new_val) || (q->next ==
NULL))
         if (q->val == new_val)
            return NULL;
         else q=q->next;
      new_item->next = q->next;
```

```
        q->next = new_item;
    }
    h->num++;
    return new_item;
}
```

3. Поиск элемента в списке

Если список упорядочен, то для поиска элемента по заданному значению нет необходимости просматривать все элементы списка. Просмотр элементов продолжается до тех пор, пока не встретится элемент, значение которого больше, чем искомое значение при упорядоченности элементов по возрастанию (или меньше искомого – при упорядоченности по убыванию). Реализация операции поиска для упорядоченного списка приведена в листинге 16.3.

Листинг 16.3

```
/* Поиск элемента в упорядоченном списке.
   Параметры: h – указатель на головной элемент;
   search_val – значение искомого элемента.
   Возвращаемое значение:
   при успешном завершении – указатель на элемент
   списка, иначе – NULL */

ITEM *search_item(HITEM *h, VAL search_val)
{
    ITEM *q=h->first;
    if ((q == NULL) || (q->val > search_val)
        return NULL;    /* Элемента нет в списке */
    do
    {
        if (q->val == search_val)
            return q;
        q = q->next;
    } while ((q != NULL) && (q->val < search_val));
    return NULL;
}
```

Поиск в неупорядоченном списке, как и поиск по значениям полей элементов списка, которые в общем случае не являются уникальными для всего списка, возможен лишь путем полного перебора элементов списка.

Если нужно найти не одно, а все значения, равные заданному значению или из указанного диапазона, то такая задача называется выборкой значений из списка.

4. Исключение элемента из списка

Для исключения некоторого элемента из списка необходимо в ссылочной части элемента, предшествующего удаляемому, указать адрес элемента, следующего за удаляемым, освободив при этом блок памяти, занимаемый удаляемым элементом (рис. 16.4).

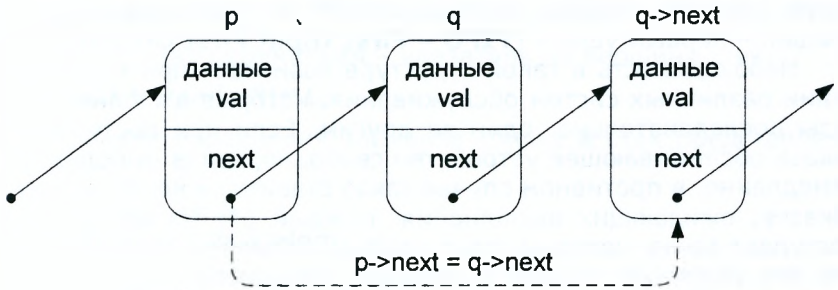


Рисунок 16.4 – Схема удаления элемента из линейного односвязного списка

Исходный код функции, реализующей удаление элемента из упорядоченного списка, приведен в листинге 16.4.

Листинг 16.4

```
/* Удаление элемента из упорядоченного списка.
   Параметры: h – указатель на головной элемент;
   del_val – значение удаляемого элемента.
   Возвращаемое значение:
   при успешном завершении – 1, иначе – 0 */
int del_item(HITEM *h, VAL del_val)
{
   ITEM *q, *p;
   q = p = h->first;
   while (q->val != del_val)
   {
      p = q;
      if ((q == NULL) || (q->val > del_val))
         return 0;
      q = q->next;
   }
   p->next = q->next;
   free(q);
   return 1;
}
```

16.5.3 Очередь, стек и дек

Частными случаями линейного односвязного списка являются очередь, стек и дек.

Очередь – это список, в котором новый элемент добавляется в конец списка, а удаляется всегда только первый элемент. Таким образом, очередь функционирует по принципу «первым пришел – первым ушел» **(FIFO – First Input First Output)**.

Необходимость в такой структуре возникает при моделировании различных систем обслуживания, которые выполняют заказы последовательно, один за другим. Если при поступлении заказа обслуживающее устройство свободно, заказ выполняется немедленно, в противном случае заказ ставится в конец очереди заказов, ожидающих выполнения. Каждый раз на выполнение поступает заказ, который стоит первым в очереди заказов, а после его удаления первым в очереди становится следующий за ним заказ. Если заказы поступают нерегулярно, длина очереди будет изменяться во времени.

Над элементами очереди обычно выполняются операции двух видов:

- выборка с одновременным удалением из очереди первого ее элемента;
- добавление нового элемента в конец очереди.

Одной из разновидностей очередей являются **очереди с приоритетами.** Очередь с приоритетами отличается от обычной очереди тем, что элементы имеют дополнительный параметр – приоритет, который и определяет порядок их удаления из очереди. Из очереди с приоритетами удаляется не первый элемент, а элемент, у которого в данный момент времени наивысший приоритет.

Стек (или магазин) представляет собой список, упорядоченный по времени поступления элементов таким образом, что извне доступен только последний из записанных в список элементов, который называется вершиной стека. Другими словами, стек функционирует по принципу «последним пришел – первым ушел» **(LIFO – Last Input First Output)**.

Дек – структура, обладающая свойствами очереди и стека, т.е. очередь, в которой элементы можно добавлять как в конец, так и в начало.

Пример работы со стеком.

Так как извне доступна только вершина стека, то основными операциями для стека являются:

- выборка с одновременным удалением вершины стека;
- добавление нового элемента в вершину стека.

Объявление элемента стека:

```
typedef struct node
{
   char elem;
   struct node *next;
} Stack;
```

Глобальная переменная – указатель на вершину стека:

```
Stack* Head=NULL;
```

Функция добавления элемента списка в стек представлена в листинге 16.5.

Листинг 16.5

```
void Push(char element)
{
   Stack *q;
   q = (Stack*) malloc(sizeof(Stack));
   if(q != NULL)
   {
      q->elem = element;
      q->next = Head;
      Head=q;
      printf("Push elem %c \n",element);
   }
   else
      puts("Error! Not free memory!");
}
```

Функция удаления элемента списка из стека представлена в листинге 16.6.

Листинг 16.6

```
char Pop()
{
   char a=Head->elem;
   Stack *q=Head;
   Head=Head->next;
   free(q);
   printf("Pop elem %c \n",a);
   return a;
}
```

Допустим, необходимо посчитать формулу, заданную строкой. Форма записи алгебраических выражений, которая знакома нам со школы, называется инфиксной (**in-fix**). Для этой формы

записи характерно то, что знаки операций располагаются между операндами, а в самой записи могут присутствовать скобки. При построении трансляторов произвольных алгебраических выражений, заданных в инфиксной форме, возникает ряд сложностей, а именно:

- последовательность выполнения операций зависит от скобочной структуры выражения;
- внутри скобок последовательность выполнения операций зависит от их приоритета.

От этих сложностей можно избавиться, представив выражение в постфиксной форме (**post-fix**), которую еще называют обратной польской записью. Выражение в постфиксной форме не имеет скобок, что позволяет не обращать внимания на приоритеты, связанные с ними. А за счет того, что все операнды расположены перед знаком операции, выражение можно вычислить последовательно, за один проход слева направо, что позволяет не учитывать приоритеты знаков операций.

Посмотрим несколько примеров записи выражения в инфиксной и постфиксной форме.

Инфиксная форма записи	Постфиксная форма записи
7 + (5 - 2) * 4	7 5 2 - 4 * +
8 / (2 + 1) * 3	8 2 1 + / 3 *
8 / ((2 + 1) * 3)	8 2 1 + 3 * /

Если посмотреть внимательно, можно заметить, что операнды в постфиксной форме не меняют своего относительного расположения, а вот порядок следования операций и функций меняется как по отношению к операндам, так и по отношению друг к другу.

Правила перевода следующие:

1. Ввести в стек левую скобку **'('**.
2. Добавить в конец строки с инфиксным выражением правую скобку **')'**.
3. Пока стек не пуст, считываем символы из строки, содержащей инфиксное выражение, слева направо и выполняем следующие действия:
   а) если текущий символ – левая скобка, помещаем ее в стек;
   б) если текущий символ – правая скобка, извлекаем знаки операций из стека и вставляем их в результирующую строку, до тех пор пока на вершине стека не появится левая скобка. Извлекаем левую скобку и отбрасываем ее;
   в) если текущий символ – знак операции, извлекаем знаки операций из стека (если они там есть), пока соот-

ветсвующее им операции имеют равный или более высокий приоритет по сравнению с текущей операцией. Вставляем извлеченные знаки операций в результирующую строку, а текущий символ – в стек;

г) если текущий символ – цифра, копируем его в результирующую строку.

Будем использовать только операции **+ - * /**. Примеры перевода выражений из инфиксной формы в постфиксную.

Исходная инфиксная строка	Стек	Результат
«4+2*7»		
4+2*7	пуст	пуст
4+2*7)	(	пуст
+2*7)	(	4
2*7)	(+	4
*7)	(+	4 2
7)	(+ *	4 2
)	(+ *	4 2 7
	(+	4 2 7 *
	(	4 2 7 * +
	пуст	4 2 7 * +
«3*5+1»		
3*5+1	пуст	пуст
3*5+1)	(	пуст
*5+1)	(	3
5+1)	(*	3
+1)	(*	3 5
1)	(+	3 5 *
)	(+	3 5 * 1
	(	3 5 * 1 +
	пуст	3 5 * 1 +
«(5-1)*4-2»		
(5-1)*4-2	пуст	пуст
(5-1)*4-2)	(	пуст
5-1)*4-2)	((	пуст
-1)*4-2)	((	5
1)*4-2)	((-	5
)*4-2)	((-	5 1
*4-2)	(	5 1 -
4-2)	(*	5 1 -
-2)	(*	5 1 - 4
2)	(-	5 1 - 4 *
)	(-	5 1 - 4 * 2
	(	5 1 - 4 * 2 -
	пуст	5 1 - 4 * 2 -

Дополнительные функции для рассчета формулы.

Функция проверки, является ли символ знаком операции, представлена в листинге 16.7.

Листинг 16.7

```
int IsOperation(char ch)
{
   if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
      return 1;
   return 0;
}
```

Функция возврата приоритета операции представлена в листинге 16.8.

Листинг 16.8

```
int Prior(char oper)
{
   if(oper=='+' ||oper=='-' )
     return 1;
   if(oper=='*' ||oper=='/')
      return 2;
   return 0;
}
```

Вид главной функции программы представлен в листинге 16.9.

Листинг 16.9

```
void main()
{
   char infix[100],postfix[100],q;
   int i=0,j=0,len;
   gets(infix);
   Push('(');    /* 1.Ввести в стек левую скобку '(' */
   len=strlen(infix);
   infix[len]=')';
   /*2. Добавить в конец строки с инфиксным
        выражением правую скобку ')' */
   infix[len+1]='\0';
   while(Head!=NULL)          /* Пока стек не пуст */
   {
      for(;infix[i]!= '\0';i++)   /* считываем символы из
            строки, содержащей инфиксное выражение,
            слева направо */
      {
         if(infix[i]=='(')
         /* Если текущий символ – левая скобка */
            Push('(');            /* помещаем ее в стек.*/
```

```
        else
           if(infix[i]==')')
           /* Если текущий символ – правая скобка */
           /* извлекаем знаки операций из стека и
              вставляем их в результирующую строку
              до тех пор, пока на вершине стека
              не появится левая скобка */
              for(;(q=Pop())!= '(';j++)
                 postfix[j]=q;
           else
              if(IsOperation(infix[i]))
              /* Если текущий символ – знак операции */
              {
                 do
                 {
                    q=Pop();
                  /* извлекаем знаки операций из стека */
                    if(Prior(q)>=Prior(infix[i]))
    /* если операции имеют равный или более высокий
       приоритет по сравнению с текущей операцией,
       вставляем извлеченные знаки операций в
       результирующую строку */
                    {
                       postfix[j]=q;
                       j++;
                    }
                    else
                    {          /* иначе назад в стек*/
                       Push(q);
                       break;
                    }
                 }while(1);
                 /* Вставим текущий символ в стек */
                 Push(infix[i]);
              }
              else
              {
     /* Если текущий символ – цифра, копируем его
        в результирующую строку*/
                 postfix[j]=infix[i];
                 j++;
              }
     }
}
```

```
    /*Заканчиваем результирующую строку*/
      postfix[j]='\0';
      puts(postfix);
      return;
}
```

Теперь, имея постфиксную запись, легко вычислить выражение. Пока не конец строки, считываем символы слева направо:

- Если текущий символ – цифра, поместим ее значение в стек (значение цифры равно значению ее символа в таблице символов **ASCII** минус значение символа **'0'**)
- Если текущий символ – знак операции, извлекаем два верхних элемента из стека в переменные **a** и **b**. Производим вычисление выражения **(b** операция **a)** (т.е. **b+a** или **b-a** или **b/a** или **b*a**). Результат вычисления кладем в стек.

Например: «4+2*7», то постфиксная форма будет **«4 2 7 * +»**

Исходная постфиксная строка	Стек	Результат
«4 2 7 * +»		
4 2 7 * +	пуст	0
2 7 * +	4	0
7 * +	4 2	0
* +	4 2 7	0
+	4 2	a=7
+	4	b=2
+	4	2*7=14
+	4 14	14
	4	a=14
	пуст	b=4
	пуст	4+14=18
	пуст	18

Функция подсчета формулы представлена в листинге 16.10.

Листинг 16.10

```
int calculate(char postfix[])
{
   int a=0,b=0,ind=0,rez=0;
      for(;postfix[ind]!=NULL;ind++)
      {
         switch(postfix[ind])
         /*проверяем, чем является элемент*/
         {
            case '+':b=Pop();a=Pop();
                     rez=a+b; Push(rez);break;
```

```
            case '-':b=Pop();a=Pop();
                    rez=a-b; Push(rez);break;
            case '*':b=Pop();a=Pop();
                    rez=a*b; Push(rez);break;
            case '/':b=Pop();a=Pop();
                    rez=a/b; Push(rez);break;
            default: Push(postfix[ind]-'0');break; /*иначе
это число*/
          }
        }
    return rez;
}
```

Можно добавить в функцию **main()** вычисление выражения следующим образом:

```
void main()
{
   int rezult;
   ..................
   puts(postfix);
   rezult=calculate(postfix);
   printf("rezult = %d",rezult);
   return;
}
```

16.5.4 Двусвязные списки

Линейный список называется списком с двумя связями или двусвязным списком, если каждый элемент хранения имеет два компонента указателя (ссылки на предыдущий и последующий элементы линейного списка), как показано на рисунке 16.5.

В программе двусвязный список описывается следующим образом:

```
typedef struct item
{
   VAL val;      /* Поле значения элемента */
   struct item *next;
   /* Ссылка на следующий элемент */
   struct item *prev;
   /* Ссылка на предыдущий элемент */
} ITEM;
```

Функция включения элемента в упорядоченный двусвязный список представлена в листинге 16.11.

Листинг 16.11

```
/* Включение элемента в упорядоченный список.
   Параметры: h – указатель на головной элемент;
   new_val – значение нового элемента;
   Возвращаемое значение:
   при успешном завершении – указатель
   на новый элемент списка,
   иначе – NULL */

ITEM *add_item (HITEM *h, VAL new_val)
{
   ITEM *q = h->first, *new_item;
   if ((new_item = (ITEM *) malloc(sizeof(ITEM))) ==
NULL)
      return NULL;
   new_item->val = new_val);
   if (h->num == 0)   /* Если список пустой */
   {
      h->first = new_item;
      new_item->next = NULL;
      new_item->prev = NULL;
   }
   else    /* Если список не пустой */
   {     /* Поиск места для нового элемента */
      while ((q->next->val < new_val) || (q->next ==
NULL))
         if (q->val == new_val)
            return NULL;
         else q=q->next;
      q->next->prev=new_item;
      new_item->next = q->next;
      new_item->prev = q;
      q->next = new_item;
   }
   h->num++;
   return new_item;
}
```

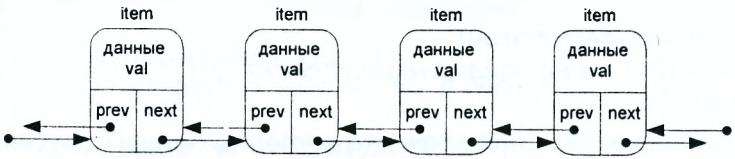


Рисунок 16.5 – Линейный двусвязный список

16.5.5 Иерархические списки

Иерархический список представляет собой несколько подсписков, ранжированных по уровням, так что элементы подсписка уровня **0** являются головными элементами (или содержат ссылки на них) для подсписков уровня **1**, те, в свою очередь, являются головными элементами для подсписков уровня **2** и т.д. (рисунок 16.6).

Таким образом, элементы иерархического списка, кроме справочной, информационной и ссылочной частей, соответствующих уровню данного подсписка, должны также содержать компоненты головного элемента подсписка нижележащего уровня (или ссылку на него).

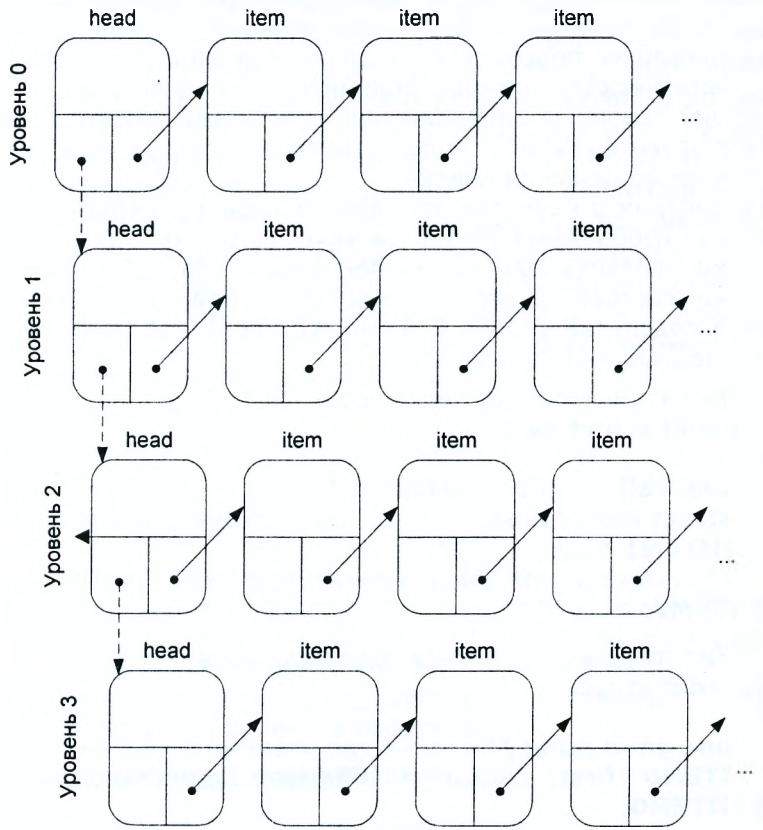


Рисунок 16.6 – Структура трехуровневого иерархическогосписка (сплошная линия – подсписок уровня 0, штриховая линия – подсписок уровня 1, пунктирная линия – подсписок уровня 2)

При реализации АТД «иерархический список» особого внимания требуют две его характеристики: количество уровней и количество разных типов подсписков, расположенных на одном и том же уровне. Если количество уровней велико и/или на каждом уровне могут располагаться несколько типов подсписков, это не только усложнит программный код, но и приведет к существенному увеличению времени выполнения основных операций. Однако нередко объекты, расположенные на одном уровне в иерархических моделях данных, относятся к одному и тому же типу.

Поскольку в иерархическом списке в общем случае могут быть объединены несколько подсписков разных типов с разными типами элементов, в число полей головного элемента каждого подсписка необходимо ввести указатели на головные элементы нижележащих подсписков (у элементов данного подсписка может быть несколько типов подчиненных подсписков), а для реализации основных операций над элементами придется определить столько функций, сколько типов подсписков содержится в данном иерархическом списке.

Рассмотрим в качестве примера реализацию иерархического списка, изображенного на рисунке 16.6, полагая, что все подсписки являются однонаправленными и что одноуровневые подсписки состоят из однотипных элементов. Тип головного элемента подсписка уровня **0** и тип его элементов можно определить следующим образом:

```
/* Тип элементов подсписка уровня 0 */
typedef struct item
{
   VAL val;   /* Поле значения */
   struct item *next;  // Ссылка на следующий элемент
   HITEM1 *sub;
   /* Ссылка на головной элемент подсписка */
} ITEM0;

/* Тип головного элемента подсписка уровня 0 */
typedef struct
{
   unsigned num; /* количество элементов подсписка */
   ITEM0 *first;  // Ссылка на первый элемент подсписка
} HITEM0;
```

Аналогично определяются типы **HITEM1**, **ITEM1** и **HITEM2**, **ITEM2** для подсписков уровней **1** и **2** с учетом того, что элементы последнего не будут иметь подсписков.

Для реализации операций вставки и удаления элементов в иерархическом списке можно использовать те же функции, что и для простых списков, но для поиска элемента уже недостаточно указать его значение, т.к. необходимо сначала найти подсписок, в котором находится данный элемент. Для этого надо знать путь к искомому элементу в виде последовательности значений ключей элементов, в которой начальное значение представляет собой ключ элемента подсписка уровня **0**, а каждое последующее значение является ключом элемента из соответствующего нижележащего подсписка. Назовем такую последовательность ключей, включая ключ самого элемента, **спецификацией элемента иерархического списка** по аналогии со спецификацией файла в иерархической файловой подсистеме.

Количество значений, входящих в спецификацию элемента, зависит от того, на каком уровне расположен данный элемент, и варьируется от **1** для элементов подсписка уровня **0** до величины, равной количеству уровней в иерархическом списке, для элементов подсписков, расположенных на последнем уровне. Поэтому функцию, реализующую операцию поиска элемента по его спецификации, определим как функцию с переменным числом параметров (листинг 16.12), в качестве которых используются адреса входящих в спецификацию значений, упорядоченные по возрастанию номера уровня (**0**,**1**,**2** и т.д.). При вызове функции список необязательных фактических параметров должен заканчиваться значением **NULL**.

Листинг 16.12

```
/* Поиск элемента в иерархическом списке.
   Параметры: h – указатель на головной элемент
   подсписка уровня 0;
   список необязательных параметров,
   заканчивающийся значением NULL.
   Возвращаемое значение: при успешном завершении –
   указатель на головной элемент подсписка, в котором
   содержится искомый элемент, иначе – NULL.
   Макроопределения va_start, va_arg, va_end и тип
   va_list определены в stdarg.h */

void *search_item(HITEM0 *h,...)
{
   va_list ap, arg, p = h, q; int i = 0;
   va_start(ap, h);
   while ((arg = va_ arg(ap, va_list)) != NULL)
```

```
        switch (i++) /* Количество меток case должно
                        быть равно количеству уровней
                        в иерархическом списке */
        {case 0:
                { if ((q = (*h.srch)(h, *(KEY0 *) arg))
                     == NULL)
                    return NULL;
                break;}
         case 1:
                { ITEM1 * h1 = ((ITEM1 *)q)->sub;
                  if (((p = h1) == NULL) ||
                      ((q = (*h1.srch)(h1, *(KEY1 *)
                           arg)) == NULL))
                    return NULL;
                break;}
         case 2:
                { ITEM2 * h2 = ((ITEM2 *)q)->sub;
                  if (((p = h2) == NULL) ||
                      ((q = (*h2.srch)(h2, * (KEY2 *)
                           arg)) == NULL))
                    return NULL;
                break;}
         default: return NULL;
         /* Слишком много параметров */
        }
    va_end(ap);
    return p;
}
```

16.5.6 Ассоциативные списки

Информационная сеть состоит из нескольких ассоциативных списков, которые организуются на основе одного общего или базового списка. Коль скоро одна и та же запись может входить в несколько списков одновременно, элементы базового списка должны содержать несколько ссылочных частей по количеству списков, в которые они могут включаться. Если количество ассоциативных списков фиксировано, то при реализации соответствующего абстрактного типа данных ссылочные части элементов базового списка и головные элементы ассоциативных списков могут быть организованы в виде массивов. В противном случае следует обеспечить возможность изменения количества соответствующих ссылочных полей элементов базового списка, что возможно только в одном случае – если эти поля объединены в подсписок.

Таким образом, в общем случае информационная сеть может быть создана на базе двухуровневого иерархического списка, в котором в качестве подсписка первого уровня выступает базовый список, а подсписки второго уровня состоят из ссылочных полей ассоциативных списков. Головные элементы ассоциативных списков удобно представить также в виде элементов специально созданного для этого списка.

В отношении основных операций и их реализации ассоциативные списки ничем не отличаются от рассмотренных выше линейных или кольцевых списков, поэтому ниже приведены только примеры определения основных типов данных для информационной сети с переменным количеством ассоциативных списков:

```
/* Тип элементов базового списка */
typedef struct item
{
   KEY key;      /* Ключевое поле */
   VAL val;      /* Поле значения */
   struct item *pre, *suc;    /* Основные ссылочные поля
*/
   PTR *psub;
   /* Указатель на первый элемент подсписка
      ссылочных полей ассоциативных списков */
} BASE_ITEM;

/* Тип элементов подсписка ссылочных частей ассоциативных списков */
typedef struct ptr
{
   struct ptr *pre, *suc;
   PASS *head;
   /* Ссылка на головной элемент соответствующего
      ассоциативного списка */
} PTR;

/* Тип элементов списка головных элементов ассоциативных списков */
typedef struct pass
{
   struct pass *pre, *suc;
   BASE_ITEM *first, *last;
   /* Ссылки на первый и последний элементы
      ассоциативного списка */
} PASS;
```

16.6 Деревья

16.6.1 АТД «Дерево»

Дерево представляет собой совокупность элементов (узлов), между которыми установлены иерархические отношения типа «предок-потомок» («родитель-сын»), причем каждый из этих элементов может иметь несколько истинных потомков (сыновей) и только одного истинного предка, кроме одного элемента, называемого **корнем дерева,** у которого нет предка.

Истинный потомок или **истинный предок –** узел, являющийся потомком или предком для некоторого узла, кроме него самого. В дереве только корень не имеет истинного предка, но может быть множество узлов, не имеющих истинных потомков. Каждый из элементов дерева является одновременно и потомком и предком для самого себя, поэтому один узел также является деревом. Дерево без узлов называется **нулевым** или **пустым** деревом.

Лист дерева – узел, не имеющий истинных потомков.

Путь из одного узла дерева в другой есть упорядоченная последовательность узлов таких, что каждый последующий узел является потомком предыдущего.

Длина пути – число, на единицу меньшее количества узлов, составляющих этот путь.

Высота узла – длина самого длинного пути из этого узла до какого-либо листа дерева.

Высота дерева – высота его корня.

Глубина узла – длина пути от корня дерева до этого узла.

Неупорядоченное дерево – дерево, в котором порядок следования потомков узлов не учитывается и может быть произвольным.

Упорядоченное дерево – дерево с определенным порядком следования потомков одного узла. Обычно потомки узла упорядочиваются слева направо так, что если узлы u_1, u_2 являются прямыми потомками одного и того же узла и первый лежит левее второго, то считается, что все потомки узла u_1 лежат левее любого потомка узла u_2.

Каждый узел дерева можно сопоставить с некоторым значением, которое называется **меткой узла** и служит для его идентификации в дереве. Деревья, у которых узлы имеют метки, называются **помеченными деревьями.**

Поддерево дерева – любой элемент в дереве вместе со всеми своими потомками. Пусть в некотором дереве **T** узлы с метками u_1, u_2,..., u_m являются прямыми потомками узла с мет-

кой u_0, тогда u_0 можно рассматривать как корень некоторого дерева T_0, являющегося поддеревом дерева **T**, а **u (i=1,2,...,m)** – как корни деревьев T_i, каждое из которых является одновременно поддеревом дерева T_i и дерева **T**.

Для таких структур как деревья одной из наиболее часто выполняемых процедур является обход всех узлов дерева в некоторой последовательности и вывод полученного списка узлов.

Существует **три способа обхода всех узлов дерева**:

- прямой порядок;
- симметричный порядок;
- обратный порядок.

Их рекурсивные алгоритмы можно объединить в одном описании:

1. Если дерево пустое, то занести в список обхода пустую запись и перейти к п.4.
2. Если дерево состоит из одного узла, то в список обхода занести этот узел и перейти к п.4.
3. Если дерево состоит из корня **R**, у которого **k** прямых потомков, являющихся корнями поддеревьев T_i **(i=1,2,...,k)**, то:
   - при **прямом обходе дерева** первым выбирается корень **R**, затем -последовательно в прямом порядке все узлы поддеревьев T_1, T_2 и т.д.;
   - при **симметричном обходе дерева** сначала выбираются все узлы поддерева T_1 в симметричном порядке, затем – корень **R** и далее последовательно в симметричном порядке все узлы поддеревьев T_2, T_3 и т.д.;
   - при **обратном обходе дерева** сначала выбираются последовательно в обратном порядке все узлы поддеревьев T_1, T_2 и т.д., а затем, т.е. в последнюю очередь, – корень **R**.
4. Закончить обход узлов дерева.

Графически порядок обхода дерева изображается в виде непрерывного контура, проходящего вдоль всех частей дерева, как показано на рисунке 16.7, где в качестве примера изображен контур **прямого обхода** дерева:

$$R, u_1, u_3, u_6, u_4, u_2, u_5, u_7, u_8, u_9.$$

При **симметричном обходе** дерева, изображенного на рисунке 16.7, получим такую последовательность меток узлов:

$$u_6, u_3, u_1, u_4, R, u_7, u_5, u_8, u_9, u_2,$$

а при **обратном обходе**:

$$u_6, u_3, u_4, u_1, u_7, u_8, u_9, u_5, u_2, R.$$

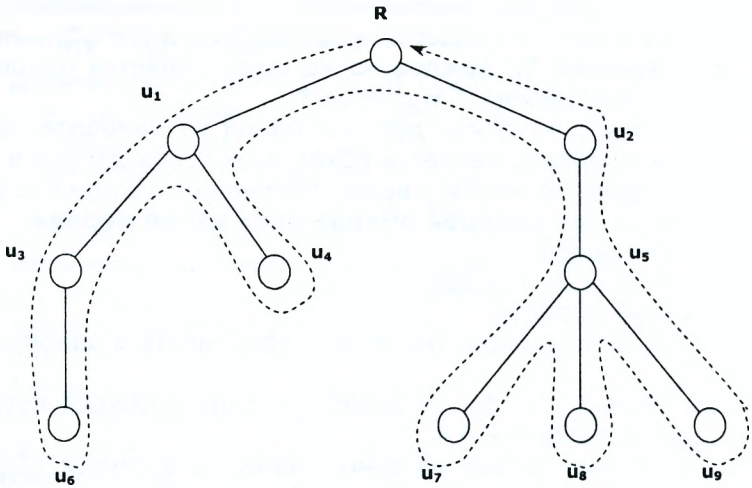


Рисунок 16.7 – Прямой (штриховая линия) способ обхода дерева

Так же, как элементы списков, узлы деревьев представляются в виде записей, поля которых подразделяются на справочную, информационную и ссылочную части. Состав информационной и ссылочной частей может варьироваться в зависимости от типа дерева и средств реализации АТД, но метка узла, как правило, относится к информационной части.

Общепринятым является **представление деревьев посредством связанных списков потомков,** когда каждый узел является одновременно головным элементом списка своих прямых потомков (для листьев этот список будет пустым) и, за исключением корня, входит в список потомков своего прямого предка.

Такое представление деревьев равносильно представлению посредством иерархического списка, в котором подсписок уровня **0** состоит из одного узла – корня дерева, а подсписки уровня **1** и всех последующих уровней формируются из прямых потомков узлов предыдущего уровня. Количество уровней в этой иерархии определяется как максимальная глубина узлов дерева плюс **1**.

16.6.2 Двоичные деревья

На практике используются главным образом деревья особого вида, называемые двоичными (бинарными).

Двоичное (бинарное) дерево – упорядоченное ориентированное дерево, у которого любой узел имеет не более двух прямых потомков, называемых левым и правым сыном.

Можно определить двоичное дерево и рекурсивно:

• пустая структура является двоичным деревом;

• дерево – это корень и два связанных с ним двоичных дерева, которые называют левым и правым поддеревом .

Двоичные деревья упорядочены, то есть различают левое и правое поддеревья. Типичным примером двоичного дерева является генеалогическое дерево (родословная). В других случаях двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья. Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево.

На рисунке 16.8 даны деревья **а)** и **б)** являются **строго двоичными**, а **в)** и **г)** – нет.

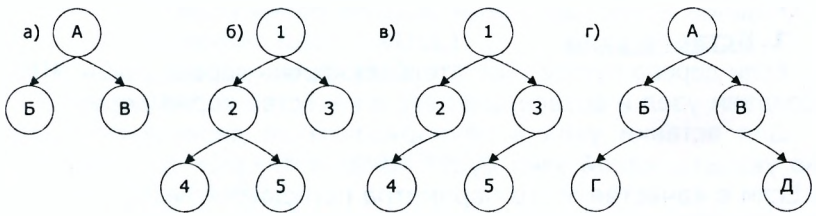


Рисунок 16.8 – Двоичные деревья

Полным двоичным деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

На рисунке 16.8 выше только дерево **а)** является **полным двоичным деревом**.

16.6.3 Деревья двоичного поиска

Деревья двоичного поиска (ДДП) – двоичные деревья, обладающие следующими свойствами:

• для меток узлов определено отношение порядка больше/меньше;

• метки всех узлов имеют уникальные значения;

• метка любого узла (кроме листьев) больше метки любого его потомка из левого поддерева, но меньше метки любого потомка из правого поддерева.

Соотношение между метками узлов ДДП и метками их потомков приводит к тому, что список узлов ДДП, полученный при

симметричном обходе, будет отсортирован в порядке возрастания меток узлов. Отсюда в частности следует, что крайний слева лист в ДДП имеет минимальную метку, а крайний справа лист – максимальную.

Каждый узел дерева имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним. Для двоичного дерева таких ссылок будет две – ссылка на левого сына и ссылка на правого сына. Рассмотрим **основные операции** для **ДДП**, определив тип элементов как (предполагая, что полезными данными для каждого узла является одно целое число):

```
typedef struct treeNode
{
   int data;                    /* Поле метки */
   struct treeNode *left,       /* Ссылка на левого сына */
                   *right;      /* Ссылка на правого сына */
} TREENODE;
```

1. Вставка узла

Если дерево пустое (указатель на корень дерева равен **NULL**), то создаем узел и возвращаем его в качестве корня дерева.

Для вставки узла в неупорядоченное дерево необходимо явно указать, какой узел будет прямым предком для нового узла. Если в качестве этого параметра передается **NULL**, то считается, что предком должен стать корень дерева.

В функции, исходный код которой представлен в листинге 16.13, осуществляется вставка узла в упорядоченное двоичное дерево, поэтому в функцию передается только указатель на корень дерева и метка нового узла.

Листинг 16.13

```
/*Вставка узла в дерево.
   Параметры:
   tree – указатель на указатель на корень дерева;
   value – метка нового узла.
   Возвращаемое значение:
   при успешном завершении – указатель
   на корень дерева, иначе – NULL.*/

TREENODE *insertNode(TREENODE *tree, int value)
{
   TREENODE
*newNode=(TREENODE*)malloc(sizeof(TREENODE));
   TREENODE *root=tree;
```

```
    /* создаем новый узел */
    if (newNode != NULL)
    {
       newNode->data = value;
       newNode->left = NULL;
       newNode->right = NULL;
    }
    else
       puts("Error!");
    /* если дерево пустое */
    if (root == NULL)
       return newNode;
    /* если дерево не пустое */
    while (root != NULL)
    {
       /* вставка узла в дерево */
       if (value < root->data)
       { /* если число меньше, нужно двигаться влево */
          if (root->left != NULL)
             root = root->left;
          else
          {
             root->left = newNode;
             break;
          }
       }
       else
          if (value > root->data)
          { // если число больше, нужно двигаться вправо
             if (root->right != NULL)
                root = root->right;
             else
             {
                root->right = newNode;
                break;
             }
          }
          else
          {
             puts("Clone");
             break;
          }
    }
    return tree;
}
```

2. Обход узлов дерева в прямом, симметричном или обратном порядке

Очевидной является реализация любого способа обхода дерева, исходя из описания их рекурсивного алгоритма, т.е. в виде рекурсивной функции.

Напомним, что существует **три** различных способа обхода деревьев (см. п. 16.6.1). Функция, исходный код которой представлен в листинге 16.14, реализует **симметричный** способ обхода дерева, функция в листинге 16.15 – **прямой** способ обхода дерева, а **обратный** способ обхода дерева представлен функцией в листинге 16.16.

Листинг 16.14

```
/*Симметричный способ обхода дерева.
  Параметры: tree – указатель на корень дерева;
  Функция осуществляет вывод на экран меток узлов
  дерева */

void inOrder(TREENODE *tree)
{
   if (tree != NULL)
   {
      inOrder(tree->left);
      printf("%3d ",tree->data);
      inOder(tree->right);
   }
}
```

Листинг 16.15

```
/*Прямой способ обхода дерева.
  Параметры: tree – указатель на корень дерева;
  Функция осуществляет вывод на экран меток узлов
   дерева */

void preOrder(TREENODE *tree)
{
   if (tree != NULL)
   {
      printf("%3d ",tree->data);
      preOrder(tree->left);
      preOder(tree->right);
   }
}
```

Листинг 16.16

```
/*Обратный способ обхода дерева.
  Параметры: tree – указатель на корень дерева;
  Функция осуществляет вывод на экран меток узлов
  дерева */

void postOrder(TREENODE *tree)
{
   if (tree != NULL)
   {
      postOrder(tree->left);
      postOder(tree->right);
      printf("%3d ",tree->data);
   }
}
```

Нерекурсивный вариант операции обхода узлов дерева предполагает использование вспомогательной памяти, в которой хранились бы метки или адреса узлов, составляющих путь от корня дерева до текущего узла. Для организации такой вспомогательной памяти удобно использовать стек, элементы которого содержат ссылки на соответствующие им узлы дерева.

3. Поиск узла по его метке

Необходимым условием корректности постановки задачи о поиске узла по его метке является уникальность меток всех узлов дерева. Если это условие не выполняется, необходимы дополнительные критерии поиска, например, метка прямого предка и/или одного из сыновей (листинг 16.17).

Листинг 16.17

```
/*Поиск узла в дереве.
  Параметры: tree – указатель на корень дерева; q – указатель на текущий узел; value – метка искомого узла.
  Возвращаемое значение:
  при успешном завершении – указатель на искомый узел
  иначе – NULL*/

TREENODE *searchNode(TREENODE *tree, int value)
{
   TREENODE *q = tree;
   while (q != NULL)
   {
      if (q->data == value)
         break;     /* узел найден */
      else
```

```
        {
            if (value < q->data) /* если число меньше, нужно
                                    двигаться влево */
                q = q->left;
            else /* если число больше, нужно двигаться вправо */
                q = q->right;
        }
    }
    if (q == NULL)
    {
        puts("No founded!");
        return NULL;  /* узел не найден */
    }
    puts("Founded!");
    return q;         /* узел найден */
}
```

4. Удаление узла

Последовательность действий при выполнении операции удаления узла из ДДП зависит от того, сколько сыновей имеет удаляемый узел:

- если удаляемый узел является листом, то для его удаления достаточно обнулить соответствующую ссылку его истинного предка;
- если удаляемый узел имеет единственного сына, то последний должен стать истинным потомком прямого предка удаляемого узла;
- если удаляемый узел имеет двух сыновей, то его следует заменить узлом с подходящей меткой, причем у последнего должно быть не более одного сына.

Поскольку метка узла в ДДП больше метки любого его потомка из левого поддерева и меньше метки любого его потомка из правого поддерева, при удалении узла с двумя сыновьями подходящим для замены узлом будет либо узел с максимальной меткой из левого поддерева, либо с минимальной – из правого поддерева удаляемого узла. Следовательно, чтобы найти такой узел, надо перейти от удаляемого узла сначала к его левому/правому сыну, а затем – по ссылкам правого/левого сына пока не будет найден узел с нулевой правой/левой ссылкой. Информационную часть найденного таким образом узла следует сохранить во вспомогательном буфере, а сам узел удалить. Далее остается лишь восстановить содержимое буфера вместо информационной части удаляемого узла с двумя сыновьями (лист. 16.18).

Листинг 16.18

```
/*Удаление узла из дерева.
  Параметры: tree – указатель на указатель на корень
  дерева; q – указатель на текущий узел, s1, s2, s – ука-
  затели на сыновей; max_node – указатель для поиска
  максимального элемента в левом поддереве;
  value – метка удаляемого узла.
  Возвращаемое значение:
  при успешном завершении – указатель на корень
  дерева, иначе – NULL.*/
TREENODE* deleteNode(TREENODE* tree, int value)
{
  TREENODE* q=tree;
  TREENODE* parent=NULL;
  TREENODE *s1,*s2,*s;
  TREENODE *max_node;
  int tmp;
  while(q!=NULL)
  {
    if(q->data==value)
      break;        /* нашли узел и выходим из цикла */
    else
    {
      parent=q;
      /* текущий узел – это отец для следующего */
      if(value<q->data)
      /* если число меньше, нужно двигаться влево */
        q=q->left;
      else
       /* если число больше, нужно двигаться вправо */
        q=q->right;
    }
  }     /* вышли из цикла поиска элемента */
  if(q==NULL)
  {
    puts("Not founded!");
    return tree;
 /* узел не найден и возвращаем указатель на дерево */
  }
  s1=q->left;
  s2=q->right;
  if(s1==NULL && s2==NULL)
```

```
/* если удаляемый узел лист */
{
   if(parent!=NULL)   /* не корень дерева */
   { /* обнуляем у отца ссылку на удаляемый узел */
      if(parent->left==q)
         parent->left=NULL;
      else parent->right=NULL;
   }
   else   /* удаление корня дерева */
   {
      free(q);
      return NULL;  /* дерево пустое */
   }
}
else
   if(s1==NULL || s2==NULL)
   /* если только один сын у удаляемого узла */
   {
      s=(s1==NULL)?s2:s1;   /* находим сына */
      if(parent!=NULL)          /*не корень дерева */
      {
      /* обнуляем у отца ссылку на удаляемый узел */
         if(parent->left==q)
            parent->left=s;
         else
            parent->right=s;
   }
   else    /* удаление корня дерева */
   {
      free(q);
      return s;  /* новый корень дерева */
   }
}
else    /* удаление узла, у которого два сына */
{
   /* удаление происходит замещением
      ищем в левом поддереве самый правый узел -
      он является самым максимальным из всех узлов
      правого поддерева - для замены удаляемого узла */
   max_node=q->left;
   while(max_node->right!=NULL)
      max_node=max_node->right;
```

```
        /* копируем значение максимального узла
           из правого поддерева */
        tmp=max_node->data;
        tree=deleteNode(tree,tmp);
        /* удаляем замещающий узел */
        /* копируем значение максимального узла
           из правого поддерева в удаляемый */
        q->data=tmp;
        return tree;
      }
      free(q);
      return tree;
}
```

В общем случае количество операций, их реализация и, соответственно, время выполнения зависят от типа дерева и выбранных средств реализации соответствующего АТД.

16.6.4 Деревья выражений

Дерево выражения – это дерево, в котором узлы помечены лексемами этого выражения. Рассмотрим этот тип деревьев на примере арифметических/логических выражений.

Деревья арифметических/логических выражений будем строить в соответствии со следующими правилами:

• Выражению вида $\mathbf{E_1 \delta E_2}$, где **δ** – знак бинарной операции, $\mathbf{E_1}$ и $\mathbf{E_2}$ – выражения, соответствует дерево, корень которого помечен знаком операции, а выражения-операнды образуют два его поддерева. Это означает, что если некоторый узел имеет метку в виде знака бинарной операции, например, **«+»**, а два его прямых потомка имеют метки, например, **x** и **y**, то вместе они представляют выражение **x+y**.

• Выражению вида **δE**, где **δ** – знак унарной операции, **E** – выражение, соответствует дерево, корень которого помечен знаком или мнемокодом операции (**exp()**, **sin()**, **sqrt()** и т.п.), первое поддерево состоит из одного узла, помеченного символом **«0»** для унарных операций **«+»** и **«-»** или **«пустой»** меткой для всех остальных унарных операций, а второе поддерево соответствует выражению-операнду. Различие между меткой **«0»** и **«пустой»** меткой заключается в том, что узлы с **«пустыми»** метками игнорируются при любом способе обхода дерева, а узлы с метками **«0»** – только при симметричном. При прямом и обратном обходе метки **«0»** позволят отличить (и корректно выполнить) унарные операции от бинарных, обозначаемых такими же знаками.

• Дерево выражения со скобками вида **(E)** строится на основе дерева выражения **E** так, что узел с меткой **«(»** будет прямым и единственным потомком узла, соответствующего первому операнду выражения **E**, а узел с меткой **«)»** – правым братом узла, являющегося корнем поддерева, соответствующего последнему операнду выражения **E**.

Таким образом, в дереве арифметического/логического выражения листья могут быть помечены идентификаторами переменных, константами и скобками, а внутренние узлы – идентификаторами унарных и бинарных операций. Например, дерево выражения **(c + 2*(d - 1))*ln(a + b)** будет выглядеть так, как показано на рисунке 16.9.

При обходе деревьев выражений тем или иным способом получаются соответствующие формы записи этих выражений: при прямом обходе – **префиксная форма записи**, при обратном обходе – **постфиксная форма записи**, а при симметричном обходе – **инфиксная форма записи**, которая совпадает с математической формой записи. Например, используя дерево выражения, приведенное на рисунке 16.9, можно легко получить префиксную

*** + c * 2 - d 1 ln + a b**

и постфиксную

c 2 d 1 - * + a b + ln *

формы записи этого выражения.

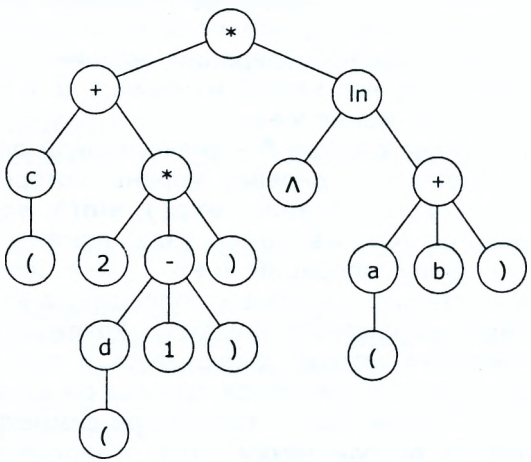


Рисунок 16.9 – Дерево выражения (c + 2*(d - 1))*ln(a + b) («пустая» метка обозначена символом «Λ»)

В префиксной и постфиксной формах записи выражений скобки не нужны, т.к. всегда можно корректно сформировать, причем единственным образом, выражения вида

<операция> <операнд1><операнд2>

или **<операнд1><операнд2><операция>**.

Инфиксная форма записи выражения возможна без скобок и со скобками. Следует отметить, что скобки нужны лишь тогда, когда необходимо получить инфиксную форму записи выражения, в остальных случаях их можно просто опустить.

С учетом сказанного, тип узлов дерева выражения можно определить как

```
typedef struct node
{
   char * label;        // Метка узла
   struct node  *s1;    // Ссылка на первого сына
                *s2;    // Ссылка на второго сына
                *s3;    // Ссылка на третьего сына
} NODE;
```

Особенностью деревьев выражений является то, что для их построения, как правило, требуется предварительный синтаксический и семантический анализ всего выражения, поэтому к основным операциям над деревьями выражений относятся:

- создание дерева выражения;
- обход узлов дерева выражения в прямом, симметричном и обратном порядке.

16.6.4.1 *Создание дерева выражения*

Поскольку любой из операндов выражения также является выражением, то для создания дерева выражения можно использовать рекурсивный подход. Ниже, в листинге 16.19, приведено описание на псевдоязыке рекурсивного алгоритма построения дерева арифметического выражения вида **f(x)**, т.е. выражения с одной переменной, в котором над операндами выполняются унарные или бинарные операции и операнды могут быть заключены в круглые скобки.

Листинг 16.19

```
/*Создание дерева выражения.
  Параметры:
  str – адрес строки, в которой записано  выражение.
  Возвращаемое значение:
  при успешном завершении – указатель на корень
  сформированного дерева, иначе – NULL.
```

```
   Для вывода сообщения об ошибке используется макрос
   #define ERROR {
    printf("Ошибка синтаксиса выражения: %s", str);
    return NULL; } */
NODE *build_tree (char *str)
{
   NODE *R=NULL, *ptr, *f;
   /* R – указатель на корень дерева */
   chars, *q,* buf, *p;
   while (str != NULL)   /* Пока не конец строки */
   {
      s = *str;
      q = str;
_1:   if (s-буква)
      {
         if ((s == 'x') || (s == 'X'))
         /* Если переменная */
         {
            buf = malloc(2);
            strncpy(buf, str, 1);
         }
         else /* Если идентификатор унарной операции */
         {
            while (*q – буква и *q != '\0')
               q++;
            if (*q == '(' )
            {
               buf = malloc(q – str + 1);
               strncpy(buf, str, q – str);
               if (buf – не идентификатор унарной
                        операции)
                  ERROR
               else
               {
                  Найти адрес соответствующей скобки ")";
                  if (Адрес найден)
                     Записать адрес в p;
                  else
                     ERROR
               }
               else
                  ERROR
      }
```

```
      Создать новый узел с меткой buf и его адрес
      записать в ptr;
      if (strlen(buf) > 1)
      {
         Создать узел с "пустой" меткой и сделать его
         первым сыном узла ptr;
         buf=malloc(p - q + 2);
         strncpy(buf, q, p - q + 1);
         if ((ptr->s2 = build_tree(buf)) == NULL)
            return NULL;
      }
      if (R)
      {
         for (f = R; f->s2 != NULL; f = f->s2);
         if (f->s1 != NULL)
            ERROR
         f->s2=ptr;
      }
      else R=ptr;
      if (q == str)
         str++;
      else str = p + 1;
   }
_2:  if (s – цифра от 0 до 9)
      {
         float C;
         while (*q – один из символов десятичной записи
                     числа и *q!='\0')
            q++;
         buf = malloc(q – str);
         strncpy(buf, str, q - str - 1);
         if (sscanf(buf, "%d", C))
            Создать новый узел с меткой buf и его адрес
            записать в ptr;
         else
            ERROR
         if (R)    /* Если дерево не пустое */
         {
            for (f = R; f->s2 != NULL; f = f->s2);
            if (f->s1 != NULL)
               ERROR
            f->s2 = ptr;
         }
```

```
            else
               R = ptr;
            str = q;
         }
_3:      if (s – символ "+" или "-")
         {
            buf = malloc(2);
            strncpy(buf, str, 1);
            Создать новый узел с меткой buf и его адрес
            записать в ptr;
            if (R)    /* Если дерево не пустое */
            {
               for (f=R; f->s2!= NULL; f = f->s2);
               if (f->s1 != NULL)
                  ERROR
               ptr->s1 = R;
               R = ptr;
            }
            else      /* Унарная операция */
            {
               Создать узел с меткой "0" и сделать его
               первым сыном узла ptr;
               R=ptr;
            }
            str++;
         }
_4:      if (s - символ "*" или "/")
         {
            Сформировать строку buf из одного символа s;
            Создать новый узел с меткой buf и его адрес
            записать в ptr;
            if (R)    /* Если дерево не пусто */
            {
               for (f = R; f->s2 != NULL; f = f->s2);
               if (f->s1 != NULL)
                  ERROR
               if ((*R->label == '+') ||
                   (*R->label == '-') &&
                   strcmp(R->s1->label, "0"))
               {
                  ptr->s1 = R->s2;
                  R->s2 = ptr;
               }
```

```
            else
            {
               ptr->s1 = R;
               R = Ptr;
            }
         }
         else
            ERROR
         str++;
      }
_5:   if (s – символ "(" )
      {
         Найти адрес соответствующей скобки ")";
         if (Адрес найден)
            Записать найденный адрес в p;
         else
            ERROR
         buf = malloc(p - q);
         stmcpy(buf, q + 1, p - q -1);
         for (f = R; f->s2 != NULL; f = f->s2);
         if (f->s1 != NULL)
            ERROR
         ptr = f->s2 = build_tree(buf);
         if (ptr == NULL)
            return NULL;
         for(f = ptr; f->s1 != NULL; f = f->s1);
         Создать узел с меткой "(" и сделать его первым
         сыном узла f;
         for (f = ptr->s3; f->s1 != NULL; f = f->s1);
         Создать узел с меткой ")" и сделать его первым
         сыном узла f;
         if (R)    /* Если дерево не пустое */
         {
            for (f = R; f->s2 != NULL; f = f->s2);
            if (f->s1 == NULL)
               ERROR
            f->s2 = ptr;
         }
         else
            R = ptr;
         str=p+1;
      }
   } return R;
}
```

Рассмотрим пошаговое выполнение описанного выше алгоритма на примере выражения **1+2*x*x+exp((x+1)*(-2/x))**:

1-й шаг:

Считывается символ «**1**».

Дерево состоит из одного узла, он же является корнем дерева.

2-й шаг:

Считывается символ «**+**».

Узел с меткой «**+**» становится новым корнем дерева, а старый корень становится его первым сыном.

3-й шаг:

Считывается символ «**2**».

Узел с меткой «**2**» становится вторым сыном корневого узла (рисунок 16.10, а).

4-й шаг:

Считывается символ «*».

Поскольку текущий корень дерева помечен символом «+», узел с меткой «*» становится вторым сыном корневого узла (рисунок 16.10, б).

5-й шаг:

Считывается символ «**x**».

Поскольку у текущего корня дерева уже есть второй сын, узел с меткой «**x**» становится вторым сыном узла, помеченного символом «*» (рисунок 16.10, в).

6-й шаг:

Считывается символ «*».

Узел с меткой «*» становится вторым сыном корневого узла, замещая стоящий в этой позиции узел (рисунок 16.10, г).

7-й шаг:

Считывается символ «**x**».

Узел с меткой «**x**» становится вторым сыном узла с меткой «*», вставленного в дерево на предыдущем шаге (рис. 16.10, д).

8-й шаг:

Считывается символ «**+**».

Узел с меткой «**+**» становится корнем дерева, а старый корень – его первым сыном (рисунок 16.10, е).

9-й шаг:

Считывается строка «**exp**».

Считанная строка является идентификатором унарной операции, поэтому вновь созданный узел с такой же меткой становится вторым сыном текущего корня дерева. Первым сыном узла с меткой «**exp**» будет узел с «**пустой**» меткой, а вторым – корень поддерева выражения, являющегося аргументом операции (включая скобки), для построения которого производится первый рекурсивный вызов.

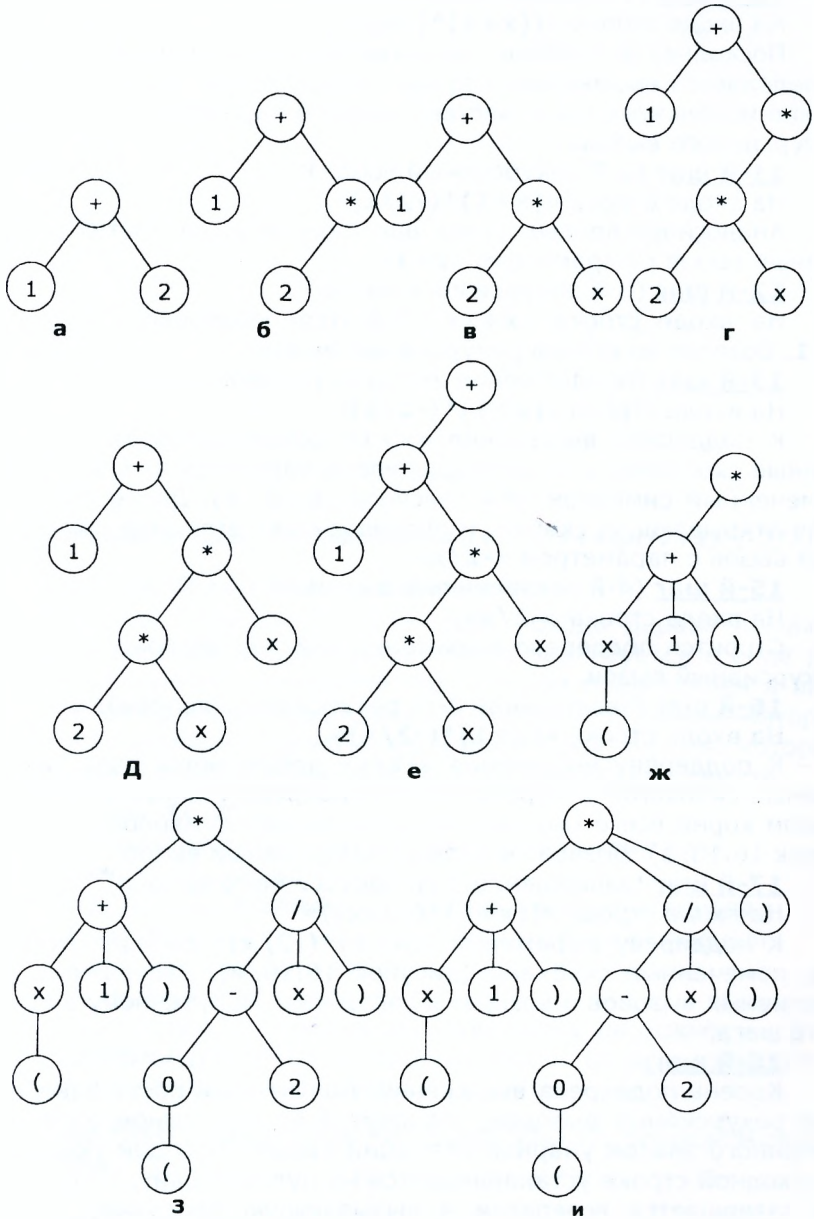


Рисунок 16.10 – Последовательность шагов при построении дерева выражения 1+2*x*x+exp((x+1)*(-2/x))

10-й шаг (1-й рекурсивный вызов):

На входе строка «**((x+1)*(-2/x))**».

Поскольку выражение начинается с открывающей скобки, определяется выражение, стоящее между скобками, и это выражение используется в качестве входного параметра для второго рекурсивного вызова.

11-й шаг (2-й рекурсивный вызов):

На входе строка «**(x+1)*(-2/x)**».

Аналогично предыдущему шагу производится третий рекурсивный вызов с параметром «**x+1**».

12-й шаг (3-й рекурсивный вызов):

На входе строка «**x+1**». Строится поддерево выражения **x+1**. Возврат во второй рекурсивный вызов.

13-й шаг (продолжение 2-го рекурсивного вызова):

На входе строка «**(x+1)*(-2/x)**».

К поддереву выражения «**x+1**» добавляются узлы, помеченные скобками, и в это поддерево добавляется новый корень, помеченный символом «*» (рисунок 16.10, ж). Далее считывается открывающая скобка, и производится четвертый рекурсивный вызов с параметром «**-2/x**».

15-й шаг (4-й рекурсивный вызов):

На входе строка «**-2/x**».

Строится поддерево выражения «**-2/x**». Возврат во второй рекурсивный вызов.

16-й шаг (завершение 2-го рекурсивного вызова):

На входе строка «**(x+1)*(-2/x)**».

К поддереву выражения «**-2/x**» добавляются узлы, помеченные скобками, и корень этого поддерева становится вторым сыном корня всего выражения, помеченного символом «*» (рисунок 16.10, з). Возврат в первый рекурсивный вызов.

17-й шаг (завершение 1-го рекурсивного вызова):

На входе строка «**((x+1)*(-2/x))**».

К поддереву выражения «**(x+1)*(-2/x)**» добавляются узлы, помеченные скобками (рисунок 16.10, и). Завершение рекурсивных вызовов и возврат к состоянию, полученному в конце **9**-го шага.

18-й шаг:

Корень поддерева выражения, полученный после завершения рекурсивных вызовов, становится вторым сыном узла, помеченного знаком унарной операции «**exp**». Текущий указатель в исходной строке устанавливается на нулевой байт, и процедура завершается возвратом в вызывающую программу адреса корня построенного дерева выражения (рисунок 16.11).

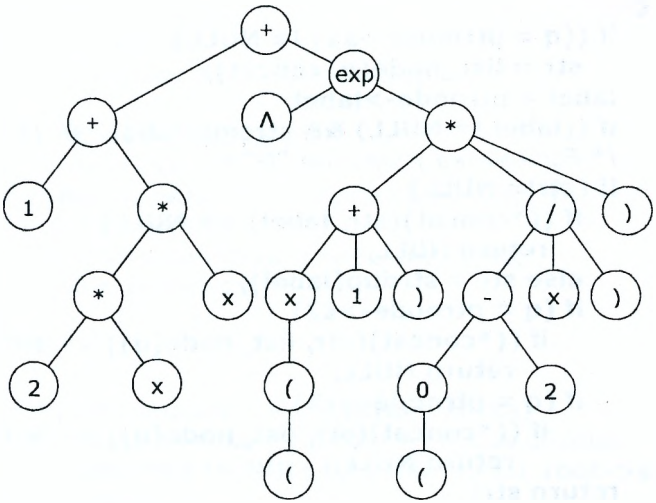


Рисунок 16.11 – Дерево выражения 1+2*x*x+exp((x+1)*(-2/x))

16.6.4.2 *Обход дерева выражения*

Алгоритм симметричного способа обхода дерева выражения, как, впрочем, и два других, легко реализуется в виде рекурсивной функции, исходный код которой представлен в листинге 16.20. Здесь же приведено определение функции, которая выполняет операцию конкатенации двух строк, одна из которых размещена в области динамического распределения памяти.

Листинг 16.20

```
/*Симметричный способ обхода дерева выражения
  (рекурсивный вариант).
  Параметры: ptrnode – указатель на корень дерева,
  concat – указатель на функцию, которая выполняет
  конкатенацию двух строк.
  Возвращаемое значение:
  при успешном завершении – указатель на строку,
  содержащую соответствующую форму записи исходного
  выражения, иначе – NULL.*/

char *list_node (NODE *ptrnode, char *(*concat)(char
* char *))
{
   NODE * q;
   char *str = NULL, * label;
   if (ptrnode != NULL)
```

```
{
   if ((q = ptrnode->s1) != NULL)
      str = list_node(q, concat);
   label = ptrnode->label;
   if ((label != NULL) && strcmp(label, "0"))
   /* Если метка узла – не "0" */
   if (str!= NULL)
      if ((*concat)(str, label) == NULL)
         return NULL;
      else str = strdup(label);
      if (q = ptrnode->s2)
         if ((*concat)(str, list_node(q)) == NULL)
            return NULL;
      if (q = ptrnode->s3)
         if ((*concat)(str, list_node(q)) == NULL)
            return NULL;
   return str;
}
```

```
/*Конкатенация двух строк.
  Параметры:
  str1, str2 – указатели на строки.
  Возвращаемое значение:
  указатель на результирующую строку (str1).
  Строка, адрес которой содержит параметр str1,
  должна размещаться в области динамического
  распределения памяти. */

char *concat(char *str1, char *str2)
{
   realloc(str1, strlen(str1) + strlen(str2) + 1);
   strcat(str1, str2);
   return str1;
}
```

В основе нерекурсивного варианта алгоритма симметричного способа обхода дерева выражения лежит использование вспомогательной памяти, организованной в виде стека. Если через **root** обозначить указатель на корень дерева, а через **top** – указатель на узел, соответствующий элементу, находящемуся в вершине стека, то алгоритм симметричного способа обхода дерева выражения можно представить на псевдоязыке следующим образом:

```
/* Начало процедуры */
if (root != NULL)
   Добавить root в вершину стека;
else
   Завершить процедуру;
root = root->s1;
while (top != NULL)    /* Пока стек не пустой */
   if (root != NULL)
   {  Добавить узел root в вершину стека;
      root = root->s1;
   }
   else
   {
      if (root->label – не "пустая" метка)
         Вывести метку узла root в список обхода;
      if (root->s3 != NULL) Добавить узел root->s3 в
вершину стека;
      if (root->s2 != NULL) root = root->s2;
      else root = NULL;
      Удалить элемент из стека;
   }
/* Конец процедуры */
```

Исходный код функции, реализующей описанный выше алгоритм, представлен в листинге 16.21. Здесь тип элементов стека определен как:

```
typedef struct item {
                        NODE * ptrnode;
            /* Указатель на узел дерева выражения */
                        struct item * ptr;
          /* Указатель на следующий элемент в стеке */
} ITEM;
```

а адреса функций, выполняющих операции вставки и удаления элементов стека, передаются в основную функцию посредством параметров.

Листинг 16.21

```
/*Симметричный способ обхода дерева выражения
  (нерекурсивный вариант).
  Параметры:
  root – указатель на корень дерева;
  concat – указатель на функцию, выполняющую
  конкатенацию двух строк;
```

__add__ – указатель на функцию, которая выполняет операцию вставки элемента в стек;
__del__ – указатель на функцию, которая выполняет операцию удаления элемента из стека.
Возвращаемое значение:
*при успешном завершении – указатель на строку, содержащую соответствующую форму записи исходного выражения, иначе – __NULL__. */*

```
char * list_node2(NODE * root,
                  char * (*concat)(char *, char *),
                  ITEM * (*add)(ITEM *, NODE *),
                  ITEM * (*del)(ITEM *))
{
   ITEM * top; /* Указатель на вершину стека */
   char * str = NULL;
   if (root != NULL)
      top = (*add)(top, root);
   else
      return NULL;
   root = root->s1;
   while (top != NULL)
      if (root != NULL)
      {
         top = (*add)(top, root);
         root = root->s1;
      }
      else
      {
         if ((root->label) && strcmp(root->label,"0"))
            if (str != NULL)
               concat(str, label);
            else
               str = strdup(label);
         if (root->s3 != NULL)
            top = (*add){top, root->s3);
         if (root->s2!=NULL)
            root = root->s2;
         else
            root = NULL;
         top = (*del)(top);
      }
}
```

16.6.5 N-арные деревья

В общем случае деревья могут иметь сыновей как больше двух, так и не иметь вовсе. Каждый узел содержит ссылку на правого брата, на первого и последнего сына и на родителя, кроме собственно метки (рисунок 16.12).

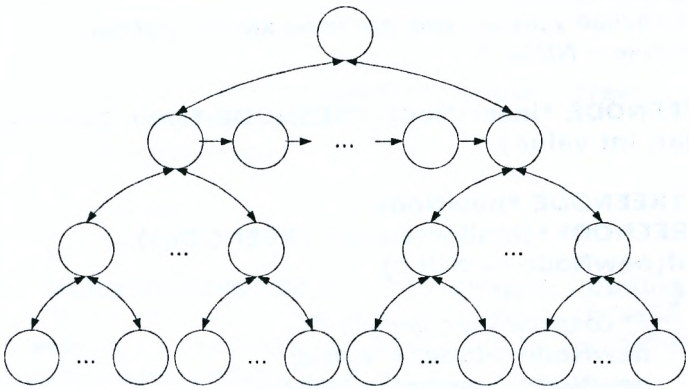


Рисунок 16.12 – Последовательность шагов при вставке узла

Каждый узел **N**-арного дерева при представлении посредством связанных списков потомков должен содержать, по меньшей мере, две ссылки: ссылку на следующий за ним узел в списке прямых потомков того же предка – такой узел называется правым братом – и ссылку на первый элемент списка прямых потомков. По-видимому, наиболее общим будет следующее **определение типа узлов дерева**:

```
typedef struct treeNode
{
   int label;                    /* метка узла */
   struct treeNode *parent;  /* указатель на отца */
   struct treeNode *brother;
   /* указатель на правого брата */
   struct treeNode *first;
   /* указатель на первого сына */
   struct treeNode *last;
   /* указатель на последнего сына */
} TREENODE;
```

Рассмотрим **основные операции** для **N**-арных деревьев:

1. Вставка узла

Функция добавления элемента в дерево представлена в листинге 16.22.

Листинг 16.22

```
/*Функция добавления элемента в дерево.
  Параметры: tree – указатель на корень дерева;
  par – указатель на отца для нового элемента;
  value – значение нового элемента.
  Возвращаемое значение:
  в случае успеха: указатель на корень дерева,
  иначе – NULL */

TREENODE *insertNode(TREENODE *tree, TREENODE
*par, int value)
{
  TREENODE *newNode =
(TREENODE*)malloc(sizeof(TREENODE));
  if(newNode != NULL)
  {
    /* создали узел дерева */
    newNode->label = value;
    newNode->parent = NULL;
    newNode->brother = NULL;
    newNode->first = NULL;
    newNode->last = NULL;
  }
  else
  {
    puts("Error!");
    return tree;
  }
  if(tree == NULL)      /* пустое дерево */
    return newNode;
    /* новый узел становится корнем дерева */
  if(par != NULL)
  /* если новый узел – не корень дерева */
  {
    newNode->parent = par;
    if(par->first != NULL)
    /* если у отца уже есть сыновья */
    {
      /* то новый узел становится последним сыном */
      par->last->brother = newNode;
      par->last = newNode;
    }
```

```
        else
        {
  /* иначе новый узел становится единственным сыном */
            par->first = par->last = newNode;
        }
     }
     else   /* если новый узел корень дерева */
     {
        newNode->first = newNode->last = tree;
        tree->parent = newNode;
        return newNode;
     }
     return tree;
}
```

2. Обход узлов дерева в симметричном порядке

Очевидной является реализация любого способа обхода дерева, исходя из описания их рекурсивного алгоритма, т.е. в виде рекурсивной функции, и не только.

Функция, исходный код которой представлен в лист. 16.23, реализует **симметричный** способ обхода дерева при помощи рекурсии.

Листинг 16.23

```
/*Функция обхода дерева в симметричном порядке.
  Параметры: tree – указатель на корень дерева.
  Возвращаемое значение:
  в случае успеха: указатель на корень дерева,
  иначе – NULL */

void inOrder(TREENODE *tree)
{
   if(tree != NULL)
   {
      inOrder(tree->first);     /* вывод первого сына */
      printf("%3d ",tree->label); /*вывод метки корня*/
      if(tree->first)                  /* если есть сыновья */
         for(tree = tree->first->brother;tree != NULL;
                tree = tree->brother)
           /* вывод всех остальных сыновей */
            inOrder(tree);
   }
}
```

3. Поиск узла в дереве по его метке

Необходимым условием корректности постановки задачи о поиске узла по его метке является уникальность меток всех узлов дерева. Если это условие не выполняется, необходимы дополнительные критерии поиска, например, метка прямого предка и/или первого сына.

В листинге 16.24 представлен исходный код рекурсивной функции, реализующей поиск узла по его метке.

Листинг 16.24

```
/*функция поиска узла.
  Параметры:
  tree – принимает указатель на корень дерева;
  value – метку искомого узла.
  Возвращаемое значение: в случае успеха – указатель на
  найденный узел, иначе – NULL. */

TREENODE *search(TREENODE *tree, int value)
{
   TREENODE* findNode = NULL;
   if(tree != NULL)
   {
      if(tree->label == value)
         return tree; /* если это искомый узел */
      /* иначе проверяем всех сыновей */
      for(tree = tree->first;tree! = NULL;tree = tree-
>brother)
      {
         findNode = search(tree,value);
         if(findNode != NULL)
            break;
         /* если нашли останавливаем поиск */
      }
   }
   return findNode;
}
```

4. Удаление узла из дерева

В общем случае декомпозиция задачи удаления узла приводит к двум подзадачам:

- исключение удаляемого узла из списка сыновей его прямого предка;
- размещение в дереве потомков удаляемого узла.

При удалении листа или корня дерева можно, очевидно, ограничиться решением лишь одной из указанных подзадач.

Исключение удаляемого узла из списка сыновей его прямого предка выполняется в соответствии с описанными выше алгоритмами для списков и не зависит от типа дерева. Вопрос о размещении потомков удаляемого узла, напротив, решается поразному в разных типах деревьев, но фактически всегда сводится к передаче потомков удаляемого узла другому узлу, например, прямому предку удаляемого узла или какому-нибудь другому узлу с подходящей меткой (см. листинг 16.25).

Листинг 16.25

```
/*Функция удаления элемента из дерева
  Параметры:
  tree – указатель на корень дерева;
  deleteNode – указатель на удаляемый элемент;
  newParent – указатель на приемного отца для
  сыновей удаляемого элемента.
  Возвращаемое значение:
  в случае успеха – указатель на корень дерева,
  иначе – NULL */

TREENODE *deleteNode(TREENODE *tree, TREENODE
*deleteNode,
                     TREENODE *newParent)
{
  TREENODE *node, *par, *tmp;
  /*проверка параметров*/
  if (deleteNode == NULL || tree == NULL || deleteNode
== newParent
        || newParent == NULL)
  {
    puts("Параметры не верные");
    return tree;
  }
  /* 1 шаг: исключить удаляемый узел, если он –
    не корень или узел newParent, если он должен
    стать корнем вместо удаляемого */
  if (deleteNode->parent != NULL)
  /* если удаляемый узел – не корень */
    node = deleteNode;
    /* будем удалять его из связей */
  else
```

```
{
   puts("Удаление корня дерева");
   /* дерево состоит из одного удаляемого элемента */
   if (tree->first == NULL)
   {
      puts("Удаление всего дерева");
      free(deleteNode);
      return NULL;
   }
   if (newParent == NULL)
   {
      puts("Нет приемного отца");
      return tree;
   }
    /*удаляемый узел – корень, нужно удалить
      приемного отца, т.к. он станет новым корнем*/
   node = newParent;
}
par = node->parent;
/*если удаляем первый элемент*/
if (node == par->first)
{
   par->first = node->brother;
   if (par->last == node)
      par->last = NULL)
}
else
{
   /*ищем узел левый брат удаляемого*/
   for(tmp = par->first;tmp->brother != node;
        tmp = tmp->brother);
   tmp->brother = node->brother;
   /*перекидываем ссылку*/
   /*если удаляемый был последним*/
   if (tmp->brother == NULL)
      par->last=tmp;
} /*2 шаг добавить список сыновей удаляемого узла
    приемному отцу*/
if (deleteNode->first != NULL)
{
   /*если есть сыновья у удаляемого*/
   if(newParent->first == NULL)
```

```
        /* у приемного отца нет сыновей */
           newParent->first = deleteNode->first;
           /* вставляем в начало */
        else
           /* вставляем в конец */
           newParent->last->brother = deleteNode->first;
        /* устанавливаем указатель на последнего сына */
        newParent->last = deleteNode->last;
        /* у всех сыновей удаляемого узла теперь
           отцом будет приемный */
        for(tmp = deleteNode->first;tmp != NULL;
            tmp = tmp->brother)
           tmp->parent = newParent;
     }
     if (deleteNode->parent == NULL) /* удаляли корень */
     {
        newParent->parent = NULL;
        /* приемный отец теперь корень */
        free(deleteNode);
        return newParent;
     }
     else
     {
        free(deleteNode);
        return tree;
     }
  }
```

5. Поиск правого брата

Поиск первого справа соседнего узла на том же уровне глубины или, просто, правого соседа может завершиться с одним из следующих результатов:

- Правым соседом может быть правый брат данного узла.
- Если узел **r** замыкает список сыновей своего прямого предка, правым соседом для него будет первый сын узла, который является правым соседом для прямого предка узла **r**. Например, на рисунке 16.7 правый сосед для узла u_4 – это узел u_5, который является первым сыном правого брата узла u_1 – прямого предка узла u_4. Налицо рекурсивная структура решаемой задачи – чтобы найти правого соседа узла, необходимо найти правого соседа для прямого предка этого узла.

• Правый сосед может отсутствовать вовсе, если, очевидно, узел является крайним справа узлом на своем уровне (узлы u_2, u_5, u_9 на рисунке 16.7).

В листинге 16.26 приведен исходный код рекурсивной функции, реализующей операцию поиска правого соседа для данного узла при условии, что адрес последнего известен и передается в функцию посредством единственного параметра.

Листинг 16.26

```
/*Функция удаления элемента из дерева
  Параметры: node – указатель на узел, для которого
  ищем соседа справа.
  Возвращаемое значение:
  в случае успеха – указатель на соседа,
  иначе – NULL */

TREENODE *right_neighb(TREENODE *node)
{
  TREENODE *findNode = NULL;
  if (node != NULL)
  {
    if(node->parent == NULL)
      return NULL;   /*у корня нет соседей*/
    if(node->brother != NULL)
      return node->brother;
      /*сосед справа – это брат*/
      /*будем искать среди двоюродых братьев*/
    findNode = right_neighb(node->parent);
    /*ищем дядю справа*/
    /*дяди справа нет, значит, нет и соседа справа*/
    if (findNode == NULL) return NULL;
    /*соседом будет первый сын дяди справа*/
    findNode = findNode->first;
  }
  return findNode;
}
```

16.6.6 Частично упорядоченные деревья

Частично упорядоченное дерево (ЧУД) – это двоичное дерево, обладающее следующими свойствами:

• для меток узлов определено отношение порядка больше/меньше;

• метка любого узла не превышает или не меньше (одно из двух) меток всех его потомков;

• новые узлы разрешается добавлять только на самом низшем уровне глубины.

Существенным отличием ЧУД от ДДП является то, что листья в ЧУД располагаются на самом низшем или предшествующем ему уровне, поэтому такие деревья всегда будут сбалансированными. Критерий сбалансированности в данном случае можно сформулировать так: двоичное дерево является сбалансированным тогда и только тогда, когда глубина любого его листа отличается от высоты дерева не более чем на **1**. Ясно, что данная формулировка эквивалентна приведенной выше.

Наиболее эффективны ЧУД для реализации очередей с приоритетами, в которых, как отмечалось выше, в любой момент времени преимущественным правом на обслуживание обладает элемент с максимальным приоритетом, а не первый, как в обычной очереди.

Рассмотрим в качестве примера очередь с приоритетами, которая возникает при распределении совместно используемых ресурсов вычислительной системы между несколькими одновременно выполняющимися процессами. Машинное время выделяется процессам квантами, поэтому, если некоторый процесс не завершился за один квант времени, обслуживание этого процесса может быть прервано, если приоритет другого процесса к этому моменту времени стал выше, чем приоритет обслуживаемого процесса.

Обычно процессы, для выполнения которых требуется меньше времени, имеют более высокие приоритеты на системные ресурсы, чем те процессы, для выполнения которых требуется значительное машинное время. Однако в таком случае более продолжительные процессы могут оказаться заблокированными, поэтому приоритет процесса определяется как функция времени:

$$\mathbf{P = C \cdot t_{exec} - t},$$

где $\mathbf{t_{exec}}$ – время, уже выделенное процессу, **t** – время, прошедшее с момента инициализации процесса. Константа **C** определяет процент машинного времени, выделяемого процессам в среднем за некоторый достаточно продолжительный промежуток времени, и должна быть несколько больше числа ожидаемых активных процессов (как правило, выбирается эмпирически найденное значение **C = 100**).

Таким образом, приоритет процесса тем выше, чем меньше значение величины **P**. Если процессу в течение длительного промежутка времени не выделяется машинное время, то его

приоритет растет (**P** становится большим отрицательным числом), и процесс, в конце концов, получает необходимые для его завершения машинные ресурсы.

Задача реализации очередей с приоритетами посредством ЧУД заключается в том, чтобы сформировать из элементов очереди с приоритетами сбалансированное двоичное дерево с частично упорядоченными узлами. Частичная упорядоченность такого дерева вытекает из требования, чтобы приоритет любого узла был не ниже приоритета его сыновей, а новый лист может быть добавлен только на самый нижний уровень справа от имеющихся на этом уровне листьев или, если этот уровень полностью заполнен, в начало нового уровня. Тогда корень дерева будет соответствовать элементу очереди с максимальным приоритетом.

Основными операциями в случае реализации очередей с приоритетами посредством ЧУД являются:

1. Поиск крайнего справа листа на последнем уровне

Чтобы найти крайний справа лист на последнем уровне, надо сначала найти крайний слева лист, так как этот лист в ЧУД всегда будет находиться на последнем уровне, и затем, выполняя рекурсивно операцию поиска первого справа соседа, найти лист, у которого нет соседа справа.

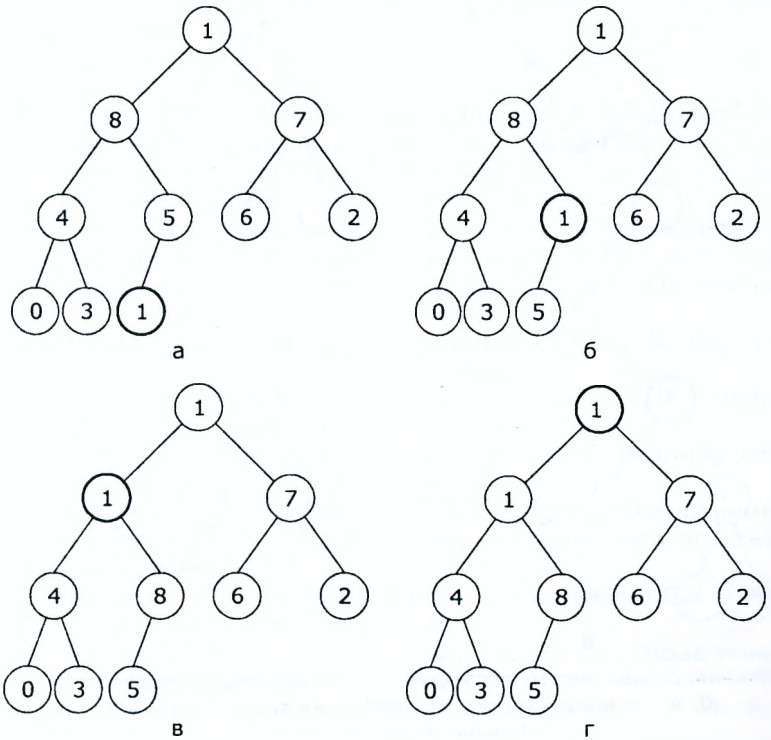
2. Вставка нового узла в дерево

Эта операция выполняется в два шага.

Сначала новый узел надо установить в первую свободную позицию справа на последнем уровне (рисунок 16.13, а) или крайнюю левую позицию на новом уровне, если последний уровень заполнен полностью (на полностью заполненном уровне располагается 2^i узлов, где **i = 0,1,2,...** – номер уровня). Если последний уровень заполнен полностью, то истинным предком для нового узла станет первый слева лист на последнем уровне глубины. В противном случае истинным предком нового узла станет либо истинный предок крайнего справа листа, либо первый справа сосед истинного предка крайнего справа листа, если крайний справа лист является левым или правым, соответственно, сыном своего истинного предка.

На втором шаге необходимо проверить, не нарушена ли частичная упорядоченность узлов дерева, т.е. не превышает ли приоритет нового узла приоритет его истинного предка. Если это действительно так, то новый узел и его истинный предок меняются местами (рисунок 16.13, б-г). Эта процедура выпол-

няется рекурсивно до тех пор, пока новый узел не займет место, в котором его приоритет не будет превышать приоритет его истинного предка или он не станет корнем дерева.

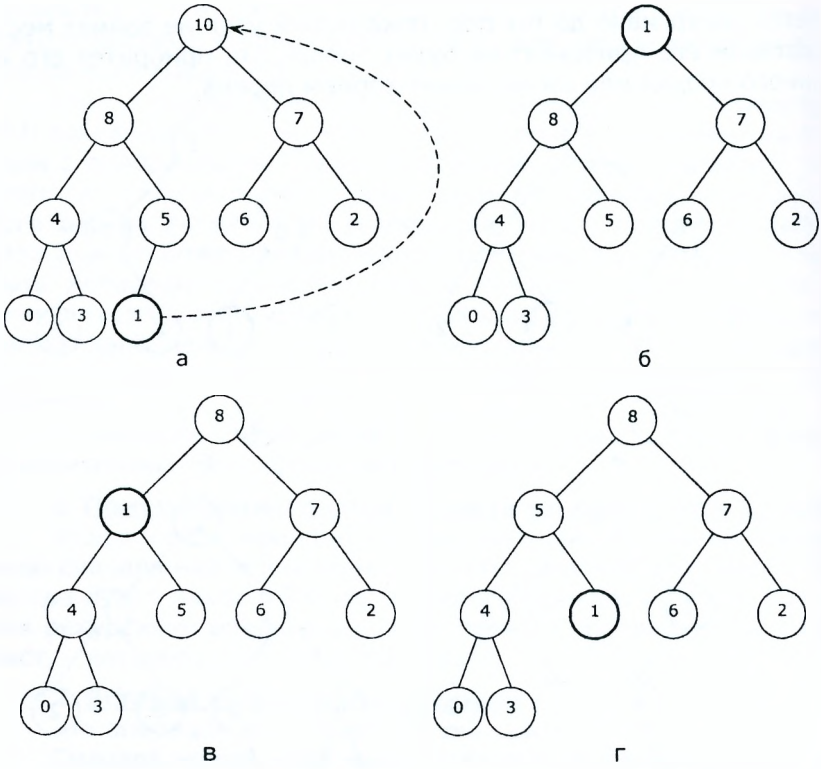


а – добавление нового узла на последний уровень;
б, в, г – новый узел «поднимается» вверх по ЧУД,
пока его метка превышает метку его истинного предка

Рисунок 16.13 – Последовательность шагов при вставке узла в ЧУД

3. Удаление корня дерева

Чтобы удалить корень ЧУД, следует найти крайний справа узел на последнем уровне (рисунок 16.14, а) и подставить его вместо корня (рисунок 16.14, б). Поскольку этот узел имеет один из самых низких приоритетов, его следует поменять местами с тем из сыновей корня, у которого больший приоритет (рисунок 16.14, в). Как и при вставке узла, процедура обмена узлов выполняется рекурсивно до тех пор, пока очередной узел не станет листом или его приоритет будет не ниже, чем у его сыновей (рисунок 16.14, г).



а – крайний справа лист на последнем уровне подставляется вместо корня ЧУД; б, в, г – новый корень «спускается» вниз, пока его метка меньше меток его сыновей

Рисунок 16.14 – Последовательность шагов при удалении корня ЧУД

Поскольку ЧУД всегда сбалансированы, время выполнения основных операций по порядку величины оказывается не хуже $\log_2 N$, где **N** – количество узлов, т.е. в худшем случае, а не в среднем как для ДДП.

ЛИТЕРАТУРА

1. Единая система программной документации – Схемы алгоритмов, программ, данных и систем – Условные обозначения и правила выполнения: ГОСТ 19.701-90.

2. Демидович, Е.М. Основы алгоритмизации и программирования. Язык Си. – СПб: БХВ-Петербург, 2008.

3. Костюк, Д.А. Методические указания по вычислительной практике «Основы программирования на языке Си» для студентов специальности I - 40 02 01 / Д.А. Костюк, Л.А. Клементьева, С.С. Дереченник. – Брест: БрГТУ, 2005. – Ч.1.

4. Костюк, Д.А. Методические указания по вычислительной практике «Основы программирования на языке Си» для студентов специальности I - 40 02 01 / Д.А. Костюк, Л.А. Клементьева, Д.О. Петров. – Брест: БрГТУ, 2007. – Ч.2.

5. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи. – СПб.: Невский диалект, 2004.

6. Вирт, Н. Алгоритмы и структуры данных. – СПб.: Невский Диалект, 2001.

7. Мороз, О.В. Абстрактные типы данных. Реализация на языке Си: пособие. – Брест: БрГТУ, 2005.

8. Касаткин, А.И. Профессиональное программирование на языке Си. Системное программирование: справочное пособие. – Мн.: Выш. шк., 1994. – 328 с.

9. Шмидский, Я.К. Программирование на языке Си. – Москва-С-Петербург-Киев: Диалектика, 2004.

10. Болски, М.И. Язык программирования Си: справочник. – М.: «Радио и связь», 1988.

11. Кочан, С. Программирование на языке Си. – М: Вильямс, 2007.

12. Голицына, О. Основы алгоритмизации и программирования / О. Голицына, И. Попов. – СПб., 2003.

13. Хусаинов, Б.С. Структуры и алгоритмы обработки данных. Примеры на языке Си. – М.: Финансы и статистика, 2004.

14. Кнут, Д.Э. Искусство программирования: учеб. пособие: в 3-х т. – М.: Вильямс, 2000.

15. Уоррен, Г.С. Алгоритмические трюки для программистов. – М.: Вильямс, 2004.

16. Криницкий, Н.А. Программирование и алгоритмические языки / Н.А. Криницкий [и др.] – М.: Наука, 1979.

17. Майоров, С.А. Электронные вычислительные машины. Введение в специальность: уч.пособие для вузов / С.А. Майоров, Г.И. Новиков. – М.: Высш. шк., 1982.

18. Матюшков, Л.П. Основы машинной математики / Л.П. Матюшков, А.А. Лихтарович – Минск: Нар. Асвета, 1988.

19. Янсен, Ф. Эпоха инноваций. – М.: ИНФРА-М, 2002.

ПРИЛОЖЕНИЯ

Математическая библиотека – math.h

Стандартная математическая библиотека **Си** включает набор функций, выполняющих элементарные математические операции. Библиотека подключается в самом начале текста программы с помощью заголовочного файла «**math.h**»:

#include <math.h>

Большая часть функций работает с типом **double**. Это значит, что они корректно работают и с типом **float**. Однако использование данных типа **long double** совместно с функциями математической библиотеки не допускается, так как в большинстве случаев приводит к ошибкам.

Наиболее часто употребляемые функции перечислены ниже.

int abs(int x) – вычисляет абсолютное значение целого аргумента.

double fabs(double x) – вычисляет абсолютное значение вещественного аргумента с двойной точностью.

long labs(long x) – вычисляет абсолютное значение длинного целого аргумента.

double acos(double x) – вычисляет **arccos(x)** в диапазоне **[0,п]**. Аргумент **x** должен быть в диапазоне **[-1,1]**.

double asin(double x) – вычисляет значение **arcsin(x)** в диапазоне **[-п/2, п/2]**. Значение аргумента **x** должно быть в диапазоне **[-1,1]**.

double atan(double x) – вычисляет значение **arctg(x)** в **[-п/2,п/2]**.

double atan2(double x, double y) – вычисляет значение **arctg(y/x)** в диапазоне **[-п,п]**. Использует знаки **x, y** для определения квадранта, которому принадлежит результат.

double atof(char* str) – преобразует строку по указателю **str** в целое значение. Строка может иметь ведущие пробелы, символы табуляции, знаки «**+**» или «**-**». Преобразование заканчивается на первой подходящей литере. Если первая же литера не подходит, результатом является **0**.

double ceil(double x) – возвращает значение удвоенной точности, равное наименьшему целому, превышающему **x**.

double cos(double x) – возвращается значение **cos(x)**. В случае ошибки возвращается **0**.

double cosh(double x) – возвращается значение **ch(x)**. В случае ошибки возвращается **0**.

double exp(double x), **long double exp(long double x)** – вычисляет экспоненту аргумента **x**.

double floor(double x) – возвращает наибольшее целое (в виде числа с удвоенной точностью), не превышающее **x**.

double fmod (double x, double y) – выдает в плавающем формате остаток от деления одного числа на другое. Она возвращает **x**, если он нуль, либо число **F** с тем же знаком, что у **x**, – такое, что **x = L·y + F** для некоторого целого **L** и **|F|<|y|**.

double frexp(double x, int *exponent) – представляет значение двойной точности **value** в виде мантиссы **x** и показателя **e**, так что **value=x·2^e**, **x>0.5** и **x≤1.0**. **e** запоминается по **eptr**, а значение **x** возвращается как результат.

```
#include <stdio.h>

void main(void)
{
   double mantissa, number;
   int exponent;
   number = 8.0;
   mantissa = frexp(number, &exponent);
   printf("The number %lf is ", number);
   printf("%lf times two to the ", mantissa);
   printf("power of %d\n", exponent);
}
```

double hypot(double x, double y) – вычисляет длину гипотенузы прямоугольного треугольника со сторонами **x**, **y**. Гипотенуза является значением, полученным вычислением $\sqrt{x^2+y^2}$.

double ldexp(double x, int exp) – вычисляет значение **x·2^e**.

double log(double x) – вычисляет натуральный логарифм **x**. Значение **x** должно быть больше **0**.

double log10(double x) – вычисляет десятичный логарифм **x** (**x** должен быть больше **0**).

int matherr(struct exception *e) – **matherr** вызывается при обнаружении ошибки функциями **math** библиотеки. Поль-

зователь может использовать библиотечный вариант этой функции или определить собственную одноименную процедуру в своей программе. Указатель на структуру **exception** должен поступать в процедуру пользователя, берущую обработку ошибки на себя. Возвращение **0** означает, что ошибка обработана правильно, **1** означает, что ошибка не может быть обработана.

double modf(double x, double *ptr) – вычисляет дробную и целую части значения аргумента **x** со знаком. Целая часть **x** со знаком запоминается в **ptr**. Возвращает дробную часть аргумента **x** со знаком.

double poly(double x, int deg, double coeff[]) – Вычисляет выражение вида: **(...(x*coeff[deg]+coeff[deg-1])*x...)*x+ +coef[0]**. Возвращает вычисленное значение.

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

double coeff[4];

void main()
{
   double x=1.2, y;
   coeff[0]=0.0;
   coeff[1]=1.0;
   coeff[2]=2.0;
   coeff[3]=3.0;
   y = poly(x,3,coeff);
   printf("Полином от %f равен %f\n", x, y);
}
```

double pow(double x, double y) – вычисляет значение **x** в степени **y**. Ошибка области определения имеет место в случае, если **x=0** и **y≤0** либо **x≤0** и **y** не целое.

double pow10(double x) – вычисляет значение **10** в степени **x**.

double sin(double x) – функция возвращает синус аргумента **x**. Значение **x** задается в радианах.

double sinh(double x) – функция возвращает гиперболический синус аргумента **x**. Значение **x** задается в радианах.

double sqrt(double x) – возвращает квадратный корень аргумента **x**. Если **x** отрицательное число, то возникает ошибка области определения.

double tan(double x) – вычисляет тангенс аргумента **x**, измеряемого в радианах.

double tanx(double x) – вычисляет гиперболический тангенс аргумента **x**, измеряемого в радианах.

Кроме перечисленных функций в **Си** доступны обычные арифметические операторы: сложения «+», вычитания «-», умножения «*», деления «/», получение остатка от деления нацело «%».

Математические константы, описываемые в математической библиотеке, приведены в таблице.

Таблица – Математические константы

Константа	Значение	Представление в Си	Константа	Значение	Представление в Си
e	2.718281828459045	M_E	$\pi/2$	1.570796326794897	M_PI_2
$\log_2 e$	1.442695040888963	M_LOG2E	$\pi/4$	0.785398163397448	M_PI_4
$\log_{10} e$	0.434294481903252	M_LOG10E	$1/\pi$	0.318309886183791	M_1_PI
$\ln 2$	0.693147180559945	M_LN2	$2/\pi$	0.636619772367581	M_2_PI
$\ln 10$	2.302585092994046	M_LN10	$2/\sqrt{\pi}$	1.128379167095513	M_2_SQRTPI
π	3.141592653589793	M_PI	$\sqrt{2}$	1.414213562383095	M_SQRT2
			$1/\sqrt{2}$	0.707106781186548	M_SQRT_2

Стандартная библиотека – stdlib.h

Библиотека подключается в самом начале текста программы с помощью заголовочного файла «**stdlib.h**»:

#include <stdlib.h>

int atoi(char *str), **long atol(char *str)** – преобразует строку по указателю **str** в вещественное число с двойной точностью (**atof**) или длинное (**atol**) значение. Строка может иметь ведущие пробелы, символы табуляции, знаки «+» или «-». Преобразование заканчивается на первой подходящей литере. Если первая же литера не подходит, результатом является **0**.

char *itoa(int value, char * str, int radix) – преобразует **value** в строку, заканчивающуюся нулем, используя **radix**. **radix** задает основание системы исчисления и должен быть в диапазоне **2..36**. Если **value** отрицательно и **radix** равен **10**, первой литерой **str** будет **'–'**. Результат записывается в строку

str, которая должна быть достаточно велика, чтобы результат поместился.

div_t div(int numerator, int denominator) – делит делимое на делитель, возвращая частное и остаток.

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

void main()
{
   div_t answ;
   int ina, inb;
   puts("Введите два целых:");
   scanf("%d %d", &ina, &inb);
   answ = div(ina, inb);
   printf("Частное=%d, и остаток=%d\n", answ.quot,
answ.rem);
}
```

ldiv_t ldiv(long int numerator, long int denominator) – делит **numerator** на **denominator**, возвращая частное и остаток. Если делитель нуль, программа завершается с сообщением об ошибке. Аргументы и возвращаемые значения имеют тип **long**.

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

void main()
{
   long numer, denom;
   ldiv_t result;
   numer = 3;
   denom = 2;
   printf("Делимое = %ld и делитель = is %ld\n", nu-
mer, denom);
   result = ldiv(numer, denom);
   printf("Частное = %ld\n", result.quot);
   printf("Остаток = %ld\n", result.rem);
}
```

char *ecvt(double val, int ndig, int *pdecpt, int *psign) – преобразует значение удвоенной точности в строку и возвращает указатель на строку. Число создаваемых цифр задает **ndig**. Число соответственно округляется. Позиция десятичной точки относительно первой (левой) цифры строки запоминается в ***pdecpt**. Если ***pdecpt** отрицательно, точка находится на соответствующее количество позиций левее начала строки. В ***psign**

запоминается **0**, если **val>0**, и отличное от **0** значение в противном случае. Строка записывается в статически выделенную память, которая используется совместно с **ecvt()** и обновляется при каждом обращении.

```
#include <stdio.h>
#include <stdlib.h>
char *buffer;
int dec, sign;
void main()
{
   buffer = ecvt(98.69138432, 5, &dec, &sign);
   printf("buffer = \"%s\", decimal = %d, sign =
%d\n", buffer, dec, sign);
}
```

void exit(int exitstatus) – вызывает все статические деструкторы, выталкивает буфера вывода, закрывает выходные файлы и возвращает управление **MS DOS** со статусом выхода **exitstatus**.

void _exit(int exitstatus) – не вызывает статические деструкторы и не освобождает буфера, а немедленно возвращает управление в **MS DOS**. **exitstatus** обычно **0** для указания на нормальное завершение программы. Значение, отличное от нуля, индицирует ошибку. Только младший байт **exitstatus** поступает к родительскому процессу. Статус выхода может быть использован посредством имени **ERRORLEVEL** в командных (**batch**) файлах.

void *malloc(unsigned numbytes) – возвращает указатель на размещенный блок памяти. Либо возвращает **NULL** при недостатке памяти или в случае, когда **numbytes=0**.

void *realloc(void *ptr, unsigned size) – изменяет размер ранее выделенного блока памяти, на который указывает **ptr**. Размер этого блока после обращения к **realloc** определяется параметром **size**. Если **size** равен **0**, блок освобождается и возвращается **NULL**. Если **ptr** равен **NULL**, то отводится (по **malloc**) **size** байтов памяти и возвращается указатель на этот массив памяти. Если для расширения текущего блока места не хватает, будет размещен новый блок, а текущий блок освободится. Текущие данные перепишутся в новый блок.

void perror(char *msg) – воспроизводит сообщение о последней ошибке, имевшей место при системном вызове или обращении к библиотечной функции, на стандартное устройство

вывода. Печатается аргумент **msg**, затем двоеточие, затем сообщение об ошибке.

int rand(void) – возвращает случайное число в диапазоне от **0** до **32767**.

void srand(unsigned seed) – инициализирует («засевает» начальным значением) генератор случайных чисел (**rand()**). Если **srand()** никогда не вызывалось, то подразумевается, что вызывалась **srand(1)**.

int system(char *string) – обеспечивает передачу строки **string** командному процессору **DOS**, как если бы эта строка была набрана за терминалом. Текущая программа ожидает конца исполнения команды, а затем возобновляет работу. Не существует способа для определения статуса завершения программ, запускаемых по **system**. Если требуется статус завершения, следует использовать одну из функций порождения.

Библиотека для работы со строками – string.h

Наиболее типичные операции над строковыми данными оформлены в виде функций стандартной библиотеки работы со строками, подключаемой к программе с помощью файла-заголовка «**string.h**»:

#include <string.h>

В качестве аргументов функций обычно выступают указатели на строки. Если результат работы функции сохраняется по соответствующему указателю, то для него должно быть предварительно зарезервировано место в памяти (в том числе и для нуль-терминатора, которым заканчивается строка).

Из набора функций библиотеки приведем некоторые наиболее часто используемые на практике.

char *strcpy(char* dst, char* src) – копирование строки, адресованной **src**, в область памяти, на которую указывает **dst**. Функция возвращает указатель на начало скопированной строки.

char *stpcpy(char* dst, char* src) – выполняет то же, что и предыдущая функция, но возвращает указатель на конец результирующей строки.

char *strncpy(char* dst, char* src, size_t n) – действие аналогично функции **strcpy**, но копируются только первые **n** символов строки **src** (или меньше, если длина строки меньше

n). Тип **size_t** – целочисленный тип, используемый стандартными функциями **Си** для хранения индексов массивов.

char *strdup(char* s) – выделяет память и копирует в нее содержимое строки **s**. Возвращает указатель на начало строки-копии или константу **NULL**, если выделение памяти завершилось неудачей.

char *strlwr(char *s), char *strupr(char *s) – преобразует все латинские символы строки **s** соответственно к нижнему и к верхнему регистру.

char *strrev(char *s) – меняет порядок следования символов строки на противоположный.

char *strcat(char *dst, char *src) – присоединяет строку **src** в конец строки **dst**. В результате **dst** содержит в себе две строки (но только один нуль-терминатор).

char *strncat(char *dst, char *src, size_t n) – аналогично предыдущей функции, но действие распространяется только на **n** первых символов строки **src**.

char *strset(char *s, int ch) – заполняет все позиции строки **s** символом **ch**.

char *strnset(char *s, int ch, size_t n) –заполняет первые **n** символов строки **s** символом **ch**.

int strcmp(const char *s1, const char *s2) – сравнивает строки, заданные константными указателями **s1** и **s2**, в лексикографическом порядке с учетом различия прописных и строчных букв. Возвращает нуль, если обе строки идентичны, значение меньше нуля, если строка **s1** расположена в упорядоченном по алфавиту «словаре» раньше, чем **s2**, или значение больше нуля в противном случае.

int stricmp(const char *s1, const char *s2) – то же, но без учета разницы между прописными и строчными латинскими буквами.

int strncmp(const char *s1, const char *s2, size_t n), strnicmp(const char* s1, const char *s2, size_t n) – аналогичные действия для первых **n** символов строк.

char *strchr(const char *s, int ch) – возвращает указатель на первое вхождение символа **ch** в строку **s**. Нуль-терминатор также участвует в поиске. Если поиск неудачен, возвращается **NULL**.

char *strrchr(const char *s, int ch) – то же, но возвращается указатель на последний совпавший символ в строке.

size_t strlen(char *s) – возвращает длину строки в байтах без учета нуль-терминатора.

char *strpbrk(const char *s1, const char *s2) – сканирует строку **s1**, сравнивая ее со всеми символами строки **s2**. При первом совпадении возвращает указатель на совпавший символ в строке **s1**, в противном случае возвращает **NULL**.

char *strstr(const char *s1, const char *s2) – находит место первого вхождения строки **s2** в строку **s1** и возвращает указатель на соответствующую позицию в **s1**.

size_t strspn(const char *s1, const char *s2) – возвращает длину сегмента строки **s1**, состоящего только из символов, входящих в строку **s2**. Нуль-терминатор не участвует в сравнении. Если строка **s1** начинается с символа, не совпадающего ни с одним из символов строки **s2**, возвращается **0**.

size_t strcspn(const char *s1, const char *s2) – возвращает длину сегмента строки **s1**, состоящего только из символов, не входящих в строку **s2**. Длина отсчитывается от начала строки **s1**.

char *strtok(char *s1, const char *s2) – выделяет лексему в строке **s1**. Под лексемой в данном случае понимается фрагмент строки **s1**, ограниченный любыми из символов, встречающихся в строке **s2**. Возвращается указатель на выделенную лексему или **NULL**, если лексема не найдена.

char *strerror(int err) – возвращает строковое представление сообщения об ошибке **errno**.

char *strsignal(int sig) – возвращает строковое представление сигнала **sig**.

void *memcpy(void *dest, const void *src, size_t n) – копирует **n** байт из области памяти **src** в **dest**, которые не должны пересекаться, в противном случае результат не определен (возможно как правильное копирование, так и нет).

void *memmove(void *dest, const void *src, size_t n) – копирует **n** байт из области памяти **src** в **dest**, которые в отличие от **memcpy()** могут перекрываться.

void *memchr(const void *s, char c, size_t n) – возвращает указатель на первое вхождение **c** в первых **n** байтах **s**, или **NULL**, если не найдено.

int memcmp(const void *s1, const void *s2, size_t n) – сравнивает первые **n** символов в областях памяти.

void *memset(void *s, int z, size_t) – заполняет область памяти одним байтом **z**.

Таблица наиболее часто используемых скан-кодов

Dec	Hex	Клавиша
8	8	Backspace
9	9	Tab
*13	D	Esc
*27	1B	Enter
*32	20	Space
59	3B	F1
60	3C	F2
61	3D	F3
62	3E	F4
63	3F	F5
64	40	F6
65	41	F7
66	42	F8
67	43	F9
68	44	F10
71	47	Home
72	48	↑
73	49	PgUp
75	4B	←
77	4D	→
79	4F	End
80	50	↓
81	51	PgDown
82	52	Insert
83	53	Delete
84	54	Shift+F1
85	55	Shift+F2
86	56	Shift+F3
87	57	Shift+F4
88	58	Shift+F5
89	59	Shift+F6

Dec	Hex	Клавиша
90	5A	Shift+F7
91	5B	Shift+F8
92	5C	Shift+F9
93	5D	Shift+F10
94	5E	Ctrl+F1
95	5F	Ctrl+F2
96	60	Ctrl+F3
97	61	Ctrl+F4
98	62	Ctrl+F5
99	63	Ctrl+F6
100	64	Ctrl+F7
101	65	Ctrl+F8
102	66	Ctrl+F9
103	67	Ctrl+F10
104	68	Alt+F1
105	69	Alt+F2
106	6A	Alt+F3
107	6B	Alt+F4
108	6C	Alt+F5
109	6D	Alt+F6
110	6E	Alt+F7
111	6F	Alt+F8
112	70	Alt+F9
113	71	Alt+F10
133	85	F11
134	86	F12
135	87	Shift+F11
136	88	Shift+F12
137	89	Ctrl+F11
138	8A	Ctrl+F12
139	8B	Alt+F11
140	8C	Alt+F12

* – Клавиши имеют только один код, остальные в составе имеют **2** кода, первый из которых **0**.

Научное издание

КОНСПЕКТ ЛЕКЦИЙ
ПО ДИСЦИПЛИНЕ
«ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ »

для студентов специальностей
1-40 02 01 «Вычислительные машины, системы и сети»
1-36 04 02 «Промышленная электроника»

Ответственный за выпуск: ***Грисевич Л.Н.***
Редактор: ***Боровикова Е.Л.***
Компьютерная вёрстка: ***Кармаш Е.Л.***
Корректор: ***Никитчик Е.В.***

ISBN 978-985-493-275-0

Лицензия № 02330/0549435 от 08.04.2009 г.
Подписано к печати 10.01.2014 г.
Формат 60×84 $^{1}/_{16}$. Бумага «Снегурочка».
Гарнитура «Arial Narrow». Усл. п. л. 13,5.
Уч.-изд. л. 14,5.
Тираж 70 экз. Заказ № 1308.
Отпечатано на ризографе Учреждения образования «Брестский государственный технический университет»
224017, Брест, ул. Московская, 267.