

Министерство образования Республики Беларусь

**Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»**

Кафедра «Вычислительные методы и программирование»

В.Л.Бусько, А.Г.Корбит, Т.М.Кривоносова

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Конспект лекций

ЧАСТЬ 1

для студентов всех специальностей и форм обучения БГУИР

Минск 2006

СОДЕРЖАНИЕ

ЧАСТЬ 1	6
1. Основные понятия и определения.....	6
1.1. Структура персональной ЭВМ	6
1.2. Размещение данных и программ в памяти ПЭВМ	6
1.3. Программные модули	7
1.4. Ошибки.....	8
1.5. Функциональная и модульная декомпозиции.....	8
1.6. Файловая система хранения информации	9
1.7. Операционная система	10
2. Понятие алгоритмов и способы их описания.....	10
2.1. Свойства алгоритмов	11
2.2. Способы описания алгоритмов.....	11
2.3. Основные символы схемы алгоритма.....	12
3. Базовые элементы языка Си.....	13
3.1. Алфавит языка.....	13
3.2. Лексемы.....	13
3.3. Идентификаторы и ключевые слова	13
3.4. Знаки операций.....	14
3.5. Литералы (константы)	14
3.6. Комментарии	14
4. Базовые типы объектов	15
4.1. Простейшая программа	15
4.2. Основные типы данных.....	16
4.3. Декларация объектов	17
4.4. Данные целого типа (int)	17
4.5. Данные символьного типа (char)	18
4.6. Данные вещественного типа (float, double).....	18
5. Константы в программах.....	18
5.1. Целочисленные константы.....	18
5.2. Константы вещественного типа	19
5.3. Символьные константы	19
5.4. Строковые константы	20
6. Обзор операций	20
6.1. Операции, выражения.....	20
6.2. Арифметические операции	20
6.3. Операции присваивания	21
6.4. Сокращенная запись операции присваивания	21
6.5. Преобразование типов операндов арифметических операций	22
6.6. Операция приведения типа	23
6.7. Операции сравнения	23

6.8. Логические операции.....	23
6.9. Побитовые логические операции, операции над битами	24
6.10. Операция «,» (запятая).....	26
7. Обзор базовых инструкций языка Си	26
7.1. Стандартная библиотека языка Си.....	26
7.2. Стандартные математические функции	26
7.3. Функции вывода данных	27
7.4. Функции ввода информации.....	28
7.5. Ввод - вывод потоками	29
8. Синтаксис операторов языка Си	29
8.1. Условные операторы	30
8.2. Условная операция «? :».....	31
8.3. Оператор выбора альтернатив (переключатель)	32
9. Составление циклических алгоритмов	33
9.1. Понятие цикла	33
9.2. Оператор с предусловием while	33
9.3. Оператор цикла с постусловием do - while	34
9.4. Оператор цикла с предусловием и коррекцией for.....	34
10. Операторы передачи управления	35
10.1. Оператор безусловного перехода goto.....	35
10.2. Оператор continue.....	36
10.3. Оператор break	36
10.4. Оператор return.....	36
11 . Указатели	37
11.1. Операции над указателями (косвенная адресация)	38
11.2. Ссылка	38
12. Массивы	39
12.1. Одномерные массивы	39
12.2. Многомерные массивы.....	40
12.3. Операция sizeof	40
12.4. Применение указателей.....	41
12.5. Указатели на указатели	42
13. Работа с динамической памятью	43
13.1. Пример создания одномерного динамического массива	45
13.2. Пример создания двумерного динамического массива	45
14. Строки в языке Си.....	45
14.1. Русификация под Visual	47
15. Функции пользователя	48
15.1. Декларация функции	48
15.2. Вызов функции.....	49
15.3. Операция typedef.....	50
15.4. Указатели на функции	51
16. Классы памяти и области действия объектов	53
16.1. Автоматические переменные.....	54
16.2. Внешние переменные	54

16.3. Область действия переменных	56
17. Структуры, объединения, перечисления	58
17.1. Структуры	58
17.2. Декларация структурного типа данных	58
17.3. Создание структурных переменных	59
17.4. Вложенные структуры	60
17.5. Массивы структур	61
17.6. Размещение структурных переменных в памяти	62
17.7. Объединения	63
17.8. Перечисления	64
18. Файлы в языке Си	65
18.1. Открытие файла	65
18.2. Закрытие файла	67
18.3. Запись - чтение информации	68
18.4. Текстовые файлы	69
18.5. Бинарные файлы	69
Литература	72
Приложение 1. Таблицы символов ASCII	73
Приложение 2. Операции языка Си	74
Приложение 3. Возможности препроцессора	76
Приложение 4. Интегрированная среда программирования Visual C++	80
1. Вид рабочего стола консольного приложения Visual C++	80
2. Создание нового проекта	81
3. Добавление к проекту файлов с исходным кодом	83
4. Компиляция, компоновка и выполнение проекта	84
5. Конфигурация проекта	86
6. Открытие существующего проекта	86
Приложение 5. Некоторые возможности отладчика Visual C++	87
1. Установка точки прерывания	87
2. Пошаговое выполнение программы	89
3. Проверка значений переменных во время выполнения программы	89
4. Окна Auto и Watch 1	90
5. Программные средства отладки	90
ЧАСТЬ 2	94
1. Системы счисления	94
1.1. Общие определения	94
1.2. Алгоритмы перевода из одной системы счисления в другую	94
2. Организация памяти и структуры данных	100
3. Линейные списки и рекурсия	107
4. Динамическая структура данных – СТЕК	113
5. Динамическая структура данных – ОЧЕРЕДЬ	120
6. Двухнаправленный линейный список	125
7. Построение обратной польской записи	135
8. Нелинейные структуры данных	143
9. Понятие хеширования	155

9.1. Хеш-функция и хеш-таблица.....	156
9.2. Схемы хеширования	159
10. Элементы теории погрешностей	162
10.1. Методы реализации математических моделей	163
10.2. Источники погрешностей.....	164
10.3. Приближенные числа и оценка их погрешностей	164
10.4. Прямая задача теории погрешностей.....	165
10.5. Обратная задача теории погрешностей	166
10.6. Понятия устойчивости, корректности и сходимости	166
11. Вычисление интегралов	167
11.1. Формулы численного интегрирования	168
11.2. Формула средних	168
11.3. Формула трапеций	169
11.4. Формула Симпсона.....	169
11.5. Схема с автоматическим выбором шага по заданной точности	170
12. Методы решения нелинейных уравнений	171
12.1. Итерационные методы уточнения корней.....	172
12.2. Метод Ньютона	172
12.3. Метод секущих.....	172
12.4. Метод Вегстейна	173
12.5. Метод парабол.....	173
12.6. Метод деления отрезка пополам	174
13. Решение систем линейных алгебраических уравнений	177
13.1. Прямые методы решения СЛАУ	178
13.2. Метод Гаусса	179
13.3. Метод прогонки.....	181
13.4. Метод квадратного корня.....	183
13.5. Итерационные методы решения СЛАУ	188
13.6. Метод простой итерации	190
14. Поиск и сортировка.....	191
14.1. Последовательный поиск в массиве	191
14.2. Барьерный последовательный поиск	193
14.3. Сортировка вставками (включениями).....	195
14.4. Двоичный поиск	197
14.5. Сортировка выбором	200
14.6. Алгоритмы сортировки выбором	201
14.7. Сортировка обменом	204
14.8. Быстрая сортировка	210
15. Обработка статистических данных	213
Используемая литература.....	219

ЧАСТЬ 1

1. Основные понятия и определения

1.1. Структура персональной ЭВМ

Персональные ЭВМ содержат клавиатуру, системный блок и дисплей. Схема ПЭВМ представлена на рис. 1.

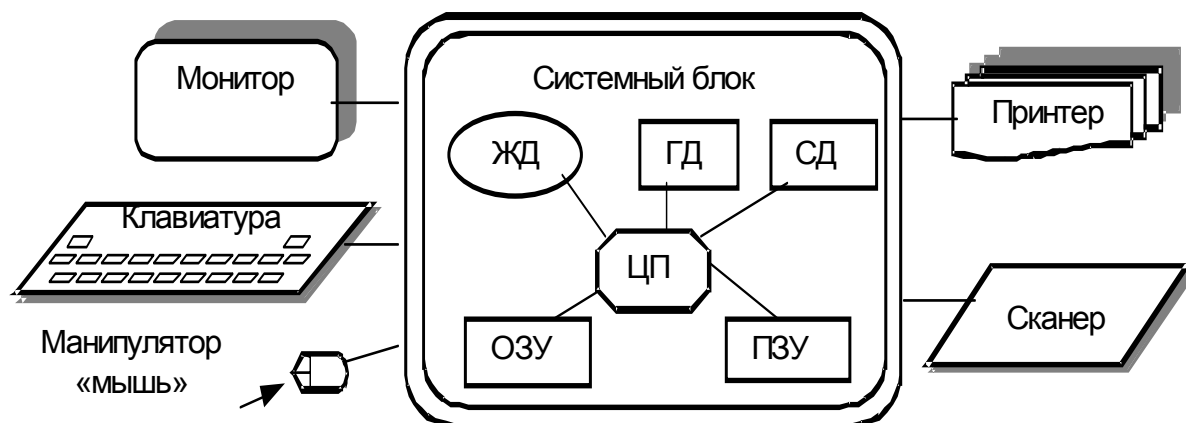


Рис.1. Схема ПЭВМ

В системном блоке ПЭВМ содержатся:

- **центральный процессор** (ЦП), который осуществляет управление работой и выполнение расчетов по программе;
- **оперативное запоминающее устройство** (ОЗУ), в котором во время работы компьютера располагаются выполняемые программы (при выключении компьютера – очищается);
- **постоянное запоминающее устройство** (ПЗУ), содержащее программы, необходимые для запуска компьютера;
- **жесткий магнитный диск** (ЖД), получивший название винчестер;
- **дисковод** (ГД) для сменных, гибких магнитных дисков (дискет);
- **CD-Rom** (СД) – устройство чтения компакт-дисков.

В системный блок встроены электронные схемы, управляющие работой различных устройств, входящих в состав компьютера. К системному блоку подключаются дисплей (монитор) для отображения информации, клавиатура для ввода данных и команд, устройство для визуального управления – «мышь», печатающее устройство – принтер, устройство для считывания и ввода информации – сканер.

1.2. Размещение данных и программ в памяти ПЭВМ

Данные и программы во время работы ПЭВМ размещаются в оперативной памяти, которая представляет собой последовательность пронумерованных ячеек. По указанному номеру процессор находит нужную ячейку, поэтому номер ячейки называется ее адресом. Минимальная адресованная ячейка состоит из 8 двоичных

позиций, т.е. в каждую позицию может быть записан либо 0, либо 1. Объем информации, который помещается в одну двоичную позицию, называется **битом**. Объем информации, равный 8 битам, называется **байтом**.

В одной ячейке из 8 двоичных разрядов помещается объем информации в 1 байт, поэтому объем памяти принято оценивать количеством байт (2^{10} байт = 1024 байт = 1 Кб, 2^{10} Кб = 1048576 байт = 1 Мб).

При размещении данных производится их запись с помощью нулей и единиц – кодирование, при котором каждый символ заменяется последовательностью из 8 двоичных разрядов в соответствии со стандартной кодовой таблицей (ASCII). Например, *D* (код – 68) → 01000100; *F* (код – 70) → 00100110; 4 (код – 52) → 00110100.

При кодировании числа (коды) преобразуются в двоичное представление, например,

$$2 = 1 \cdot 2^1 + 0 \cdot 2^0 = 10_2; 5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 101_2; 256 = 1 \cdot 2^8 = 100000000_2.$$

С увеличением числа количество разрядов для его представления в двоичной системе резко возрастает, поэтому для размещения большого числа выделяется несколько подряд расположенных байт. В этом случае адресом ячейки является адрес первого байта, один бит которого выделяется под знак числа.

Программа – это последовательность **команд** (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке.

Команда размещается в комбинированной ячейке следующим образом: в первом байте – код операции (КОП), которую необходимо выполнить над содержимым ячеек; в одной, двух или трех ячейках (операндах команды) по 2 (4) байта – адреса ячеек (A1, A2, A3), над которыми нужно выполнить указанную операцию. Номер первого байта называется адресом команды. Последовательность из этих команд называется **программой в машинных кодах** (рис. 2).

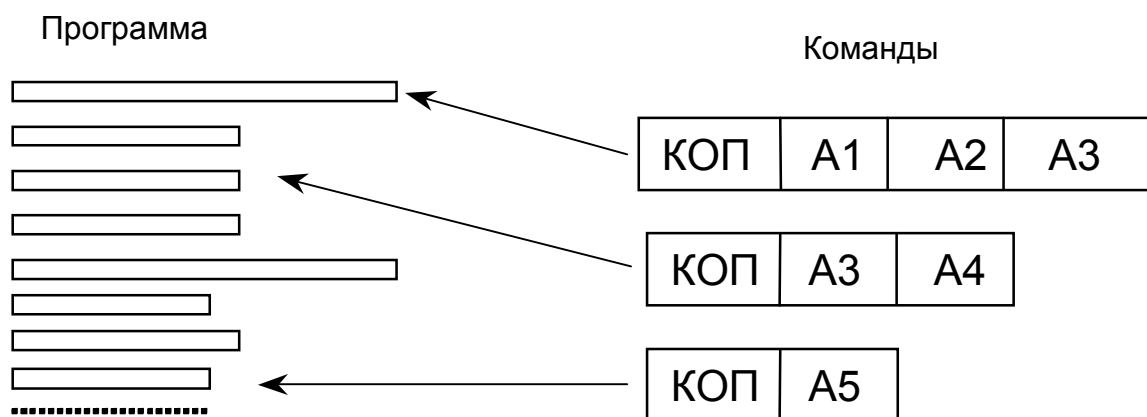


Рис. 2. Схема программы в машинных кодах

1.3. Программные модули

Программа записывается на **языке высокого уровня**, наиболее удобном для реализации алгоритма решения определенного класса задач. Исходный текст программы, введенный с помощью клавиатуры в память компьютера, – **исходный модуль** (в Си – расширение *.cpp).

Транслятор – программа, осуществляющая перевод текстов с одного языка на другой, т.е. с входного языка системы программирования на машинный язык ЭВМ. Одной из разновидностей транслятора является **компилятор**, обеспечивающий перевод программ с языка высокого уровня (приближенного к человеку) на язык более низкого уровня (близкий к ЭВМ), или машинозависимый язык.

Интерпретатор выполняет созданную программу путем одновременного анализа и реализации предписанных действий, при использовании отсутствует разделение на две стадии – перевод и выполнение.

Большинство трансляторов языка Си – компиляторы.

Результат обработки исходного модуля компилятором – **объектный модуль** (расширение **.obj*), это незавершенный вариант машинной программы, т.к., например, к нему должны быть присоединены модули стандартных библиотек. Здесь компилятор (*Compiler*) – вид транслятора, представляющий программу-переводчика исходного модуля в язык машинных команд.

Исполняемый (абсолютный, загрузочный) модуль создает вторая специальная программа – «компоновщик». Ее еще называют редактором связей (*Linker*). Она и создает модуль, пригодный для выполнения на основе одного или нескольких объектных модулей.

Загрузочный модуль (расширение **.exe*) – это программный модуль, представленный в форме, пригодной для выполнения.

1.4. Ошибки

Ошибки, допускаемые при написании программ, разделяются на синтаксические и логические.

Синтаксические ошибки – нарушение формальных правил написания программы на конкретном языке, обнаруживаются на этапе трансляции и могут быть легко исправлены.

Логические ошибки – ошибки алгоритма и семантические, которые могут быть исправлены только разработчиком программы. Причина ошибки алгоритма – несоответствие построенного алгоритма ходу получения конечного результата сформулированной задачи. Причина семантической ошибки – неправильное понимание смысла (семантики) операторов языка.

1.5. Функциональная и модульная декомпозиции

Для большинства задач алгоритмы их решения являются довольно большими и громоздкими. При программировании нужно стараться получить программу удобочитаемую, высокоэффективную и легко модифицируемую, для чего производят декомпозицию сложного алгоритма поставленной задачи, т.е. разбивают его на более простые подзадачи, затем декомпозицию подзадач и т.д.

Основной прием – разбивка алгоритма на отдельные функции и/или модули, используя функциональную и/или модульную декомпозиции.

Функциональная декомпозиция – метод разбивки большой программы на отдельные функции, т.е. общий алгоритм – на отдельные шаги, которые потом оформляют в виде отдельных функций.

Алгоритм декомпозиции можно представить следующим образом:

- программу создать в виде последовательности более мелких действий;
- каждую детализацию подробно описать;
- каждую детализацию представить в виде абстрактного оператора, который должен однозначно определять нужное действие, и в конечном итоге эти абстрактные действия заменятся на группы операторов выбранного языка программирования.

При этом надо помнить, что каждая детализация – это один из вариантов решения, и поэтому необходимо проверить, что:

- решение частных задач приводит к решению общей задачи;
- построенная декомпозиция позволяет получать команды, легко реализуемые на выбранном языке программирования.

Единица компиляции в языке Си – отдельный файл (модуль). **Модульная декомпозиция** – разбиение программы на отдельные файлы, каждый из которых решает конкретную задачу и облегчает процесс ее работы. Кроме того, код программы, разделенный на файлы, позволяет части этого кода использовать в других программах.

1.6. Файловая система хранения информации

Для размещения информации и программ на различных устройствах, необходимых пользователю, была разработана концепция файлов.

Под **файлом** понимается поименованное на внешнем носителе место (запоминающее устройство, диск и т.п.), отведенное для размещения и (или) чтения некоторой информации. При этом файл может быть пустым, т.е. место отведено, поименовано, а информация отсутствует. Информация, помещенная в файл, получает имя этого файла.

За работу с файлами в компьютере отвечают специальные программы, набор которых называется **файловой системой**, основные функции которой – предоставить пользователю средства для работы с данными.

Имя, которое присваивается файлу, может иметь тип, называемый «расширение». Имя и тип разделяются точкой. При отсутствии типа точка необязательна.

Для более удобного размещения файлов введены каталоги.

Каталог (папка) – это группа файлов на одном носителе, имеющий общее имя. Если каталог вложен внутрь другого каталога, он является **подкаталогом**. Такая вложенность может быть многократной и тогда образуется иерархическая структура хранения данных.

Внешним носителям присваиваются имена. Для дисков, например, имена обозначаются одной буквой – a:, b:, c:,... При этом на одном винчестере для удобства размещения файлов может быть организовано несколько логических дисков с разными именами.

Маршрут (путь) файла. При сложной структуре хранения файлов разные файлы могут иметь одинаковые имена и быть расположены в разных каталогах (дисках), поэтому для точной идентификации (указания) файла необходимо кроме

имени указывать путь к файлу, т.е. место на диске и цепочку подкаталогов, где он находится. Например:

c:\bc31\doc\lec.doc или *d:\work\prog.cpp*.

Для работы с файлами обычно используют специальные программы, такие, как *FAR*, *WinCom* и *Проводник*.

1.7. Операционная система

Вся работа компьютера осуществляется под управлением специальных программ, называемых операционной системой (ОС). С точки зрения пользователя ОС – это набор системных команд, задавая которые можно потребовать от ПЭВМ выполнения многих полезных процедур и действий.

Часть программ ОС предназначена для управления процессом выполнения задач. Группа программ так называемого администратора системы позволяет следить за работой пользователей в рамках системы. Важное место занимает блок программ, обеспечивающих обмен сообщениями между пользователями сети.

Удобства, предоставляемые пользователю, зависят от качества ОС, которые постоянно развиваются. В настоящее время наибольшее распространение имеют ОС WindowsXX и LinuxXX.

2. Понятие алгоритмов и способы их описания

Решение задачи на ЭВМ можно разбить на следующие этапы:

- математическая или информационная формулировка задачи;
- выбор метода (численного) решения поставленной задачи;
- построение алгоритма решения поставленной задачи;
- запись построенного алгоритма, т.е. написание текста программы;
- отладка программы – процесс обнаружения, локализации и устранения возможных ошибок;
- выполнение программы – получение требуемого результата.

Понятие алгоритма занимает центральное место в современной математике и программировании.

Алгоритмизация – сведение задачи к последовательным этапам действий, так чтобы результаты предыдущих действий использовались при выполнении следующих.

Числовой алгоритм – детально описанный способ преобразования числовых входных данных в выходные при помощи математических операций. Существуют нечисловые алгоритмы, которые используются в экономике и технике, в различных научных исследованиях.

В общем, **алгоритм** – строгая и четкая система правил, определяющая последовательность действий над некоторыми объектами и после конечного числа шагов приводящая к достижению поставленной цели.

2.1. Свойства алгоритмов

Дискретность – значения новых величин (данных) вычисляются по определенным правилам из других величин с уже известными значениями.

Определенность (детерминированность) – каждое правило из системы однозначно, а данные однозначно связаны между собой, т.е. последовательность действий алгоритма строго и точно определена.

Результативность (конечность) – алгоритм решает поставленную задачу за конечное число шагов.

Массовость – алгоритм разрабатывается так, чтобы его можно было применить для целого класса задач, например, алгоритм вычисления определенных интегралов с заданной точностью.

2.2. Способы описания алгоритмов

Наиболее распространенными способами описания алгоритмов являются словесное и графическое описания алгоритма.

Словесное описание алгоритма рассмотрим на конкретном примере: необходимо найти корни квадратного уравнения $ax^2 + bx + c = 0$ ($a \neq 0$):

- 1) вычислить $D = b^2 - 4 \cdot a \cdot c$;
- 2) если $D < 0$, перейти к 4;
- 3) вычислить корни уравнения $x_1 = (-b + \sqrt{D}) / (2 \cdot a)$; $x_2 = (-b - \sqrt{D}) / (2 \cdot a)$;
- 4) конец.

Здесь алгоритм описан с помощью естественного языка, а объекты обработки, являющиеся числами, обозначены буквами.

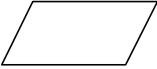


Графическое описание алгоритма – это представление алгоритма в виде схемы, состоящей из последовательности блоков (геометрических фигур), каждый из которых отображает содержание очередного шага алгоритма. Внутри фигур кратко записывают выполняемое действие. Такую схему называют блок-схемой алгоритма.

Правила изображения фигур сведены в единую систему документации (ГОСТ 19.701-90), по которой – это схема данных, отображающая путь данных при решении задачи и определяющая этапы их обработки.

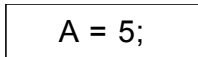

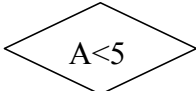
Схема содержит: *символы данных* (могут отображать тип носителя данных); *символы процесса*, который нужно выполнить над данными; *символы линий*, указывающих потоки данных между процессами и носителями данных; *специальные символы* (для удобства чтения схемы).

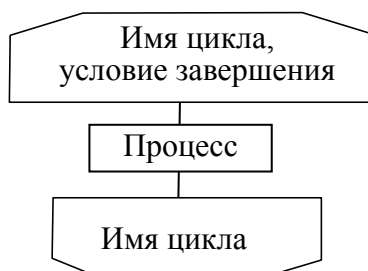
2.3. Основные символы схемы алгоритма

Символы ввода-вывода данных:

-  – данные ввода/вывода (носитель не определен);
-  – ручной ввод данных с устройства любого типа, например, с клавиатуры;
-  – отображение данных в удобочитаемой форме на устройстве, например, дисплее.

Символы процесса:

-  – **процесс** – отображение функции обработки данных, приводящей к изменению значения указанного объекта;
-  – **предопределенный процесс** – отображение группы операций, которые определены в другом месте, например в подпрограмме;
-  – **решение** – отображение функции, имеющей один вход и ряд альтернативных выходов, из которых только один может быть активирован после анализа условия, указанного внутри этого символа.



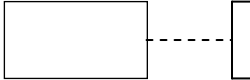


Граница цикла – начало и конец цикла,

или, наоборот, – условие завершения указывают в нижней границе.

Символы линий – отображают поток данных или управления. Линии – горизонтальные или вертикальные, имеющие только прямой угол перегиба. Стрелки – указатели направления не ставятся, если управление идет сверху вниз или слева направо.

Специальные символы

-  **Соединитель** – используется при обрыве линии и продолжении ее в другом месте (необходимо присвоить название).
-  **Терминатор** – вход из внешней среды или выход во внешнюю среду (начало или конец схемы программы).
-  **Комментарии.**

3. Базовые элементы языка Си

В языке Си фундаментальным понятием является *инструкция* (операция, оператор, функция), которая представляет собой описание определенного набора действий над некоторыми объектами. Объектам, над которыми выполняются эти действия, вместо номеров ячеек в памяти принято давать имена (идентификаторы), а содержимое ячеек называть переменными, или константами, в зависимости от того, изменяется значение в процессе работы или нет.

Таким образом, программа состоит из последовательности инструкций, оформленных в строгом соответствии с набором правил, составляющих синтаксис языка Си. Рассмотрим эти правила.

3.1. Алфавит языка

Каждому из множества значений, определяемых одним байтом (от 0 до 255), в таблице знакогенератора ЭВМ ставится в соответствие символ. По кодировке фирмы IBM символы с кодами от 0 до 127, образующие первую половину таблицы знакогенератора, построены по стандарту ASCII и одинаковы для всех компьютеров, вторая половина символов (коды 128 – 255) может отличаться и обычно используется для размещения символов национального алфавита, коды 176 – 223 отводятся под символы псевдографики и коды 240 – 255 – под специальные знаки (прил. 1).

Алфавит языка Си включает:

- буквы латинского алфавита и знак подчеркивания (код 95);
- арабские цифры от 0 до 9;
- специальные символы, смысл и использование которых будем рассматривать в соответствующих темах;
- пробельные (разделительные) символы: пробел, символы табуляции, перевода строки, возврата каретки, новая страница и новая строка.

3.2. Лексемы

Из символов алфавита формируются лексемы языка – минимальные значимые единицы текста в программе:

- идентификаторы;
- ключевые (зарезервированные) слова;
- знаки операций;
- константы;
- разделители (скобки, точка, запятая, пробельные символы).

Границы лексем определяются другими лексемами, такими, как разделители или знаки операций, а также комментариями.

3.3. Идентификаторы и ключевые слова

Идентификатор (ID) – это имя программного объекта (константы, переменной, метки, типа, функции, модуля и т.д.). В идентификаторе могут использо-

ваться латинские буквы, цифры и знак подчеркивания; первый символ ID – не цифра; пробелы внутри ID не допускаются.

Длина идентификатора определяется версией транслятора и редактора связей (компоновщика). Современная тенденция – снятие ограничений длины идентификатора.

При именовании объектов следует придерживаться общепринятых соглашений:

- ID переменной обычно пишется строчными буквами – *index*, а *Index* – это ID типа или функции, *INDEX* – константа;

- идентификатор должен нести смысл, поясняющий назначение объекта в программе, например, *birth_date* – день рождения, *sum* – сумма;

- если ID состоит из нескольких слов, как, например, *birth_date*, то принято либо разделять слова символом подчеркивания, либо писать каждое следующее слово с большой буквы – *BirthDate*.

В Си прописные и строчные буквы – различные символы. Идентификаторы *Name*, *NAME*, *name* – различные объекты.

Ключевые (зарезервированные) слова не могут быть использованы в качестве идентификаторов.

3.4. Знаки операций

Знак операции – это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Операции делятся на унарные, бинарные и тернарные, по количеству участвующих в них операндов.

3.5. Литералы (константы)

Когда в программе встречается некоторое число, например 21, то это число называется литералом, или литеральной константой. Константой, потому что мы не можем изменить его значение, и литералом, потому что оно буквально передает свое значение (от латинского *literal* – буквальный).

Константа является неадресуемой величиной, хотя реально она хранится в памяти машины, но нет никакого способа узнать ее адрес. Каждая константа имеет определенный тип.

3.6. Комментарии

Еще один базовый элемент языка программирования – *комментарий* – не является лексемой. Внутри комментария можно использовать любые допустимые на данном компьютере символы, поскольку компилятор их игнорирует.

В Си комментарии ограничиваются парами символов */** и **/*, а в C++ введен вариант комментария, который начинается символами *//* и заканчивается символом перехода на новую строку.

4. Базовые типы объектов

4.1. Простейшая программа

Программа, написанная на языке Си, состоит из одной или нескольких функций, одна из которых обязательно имеет идентификатор (имя) *main* – основная, главная. Ее назначение – управление всей работой программы (проекта). Данная функция, как правило, не имеет параметров и не возвращает результат, но наличие круглых скобок (как и для других функций) обязательно.

Общая структура программы на языке Си имеет вид

```
<директивы препроцессора>  
<определение типов пользователя – typedef>  
<описание прототипов функций>  
<определение глобальных переменных>  
<функции>
```

В свою очередь, функции имеют структуру

```
<класс памяти> <тип> <ID функции> (<список параметров>)  
{ – начало функции  
    код функции  
} – конец функции
```

Рассмотрим кратко основные части общей структуры программ.

Перед компиляцией программа обрабатывается препроцессором (прил. 3), который работает под управлением директив.

Препроцессорные директивы начинаются символом #, за которым следует наименование директивы, указывающее ее действие.

Препроцессор решает ряд задач по предварительной обработке программы, основной из которых является подключение к программе так называемых заголовочных файлов (обычных текстов) с декларацией стандартных библиотечных функций, использующихся в программе. Общий формат ее использования

```
#include <ID_файла.h>
```

где *h* – расширение заголовочных файлов.

Если идентификатор файла заключен в угловые скобки (< >), то поиск данного файла производится в стандартной директории, если – в двойные кавычки (" "), то поиск файла производится в текущей директории.

К наиболее часто используемым библиотекам относятся:

stdio.h – содержит стандартные функции файлового ввода-вывода;

conio.h – функции для работы с консолью (клавиатура, дисплей);

math.h – математические функции.

Второе основное назначение препроцессора – обработка макроопределений. Макроподстановка «определить» имеет общий вид

```
#define <ID> <строка>
```

Например: **#define** PI 3.1415927

– в ходе препроцессорной обработки программы идентификатор *PI* везде будет заменяться значением 3.1415927.

Рассмотрим пример, позволяющий понять простейшие приемы программирования на языке Си:

```
#include <stdio.h>
void main(void)
{
    // Начало функции main
    printf(" Высшая оценка знаний - 10 !");
    // Окончание функции main
}
```

Отличительным признаком функции служат скобки () после ее идентификатора, в которые заключается список параметров. Если параметры отсутствуют, указывают атрибут *void* – отсутствие значения. Перед ID функции указывается тип возвращаемого ею результата, так как функция *main* ничего не возвращает – в качестве результата – *void*.

Код функции представляет собой набор инструкций, каждая из которых оканчивается символом «;». В нашем примере одна инструкция – функция *printf*, выполняющая вывод данных на экран, в данном случае указанную фразу.

4.2. Основные типы данных

Данные в языке Си разделяются на две категории: простые (скалярные), будем их называть базовыми, и сложные (составные) типы данных.

Основные типы базовых данных: целый – *int* (integer), вещественный с одинарной точностью – *float* и символьный – *char* (character).

В свою очередь, данные целого типа могут быть короткими – *short*, длинными – *long* и беззнаковыми – *unsigned*, а вещественные – с удвоенной точностью – *double*.

Сложные типы данных – массивы, структуры – *struct*, объединения или смеси – *union*, перечисления – *enum*.

Данные целого и вещественного типов находятся в определенных диапазонах, т.к. занимают разный объем оперативной памяти, табл. 1.

Таблица 1

Тип данных	Объем памяти (байт)	Диапазон значений
char	1	–128 ...127
int	2	–32768...32767
short	2(1)	–32768...32767(–128...127)
long	4	–2147483648...2147483647
unsigned int	4	0...65535
unsigned long	4	0...4294967295
float	4	$3,14 \cdot 10^{-38} \dots 3,14 \cdot 10^{38}$
double	8	$1,7 \cdot 10^{-308} \dots 1,7 \cdot 10^{308}$

4.3. Декларация объектов

Все объекты, с которыми работает программа, необходимо декларировать, т.е. объявить компилятору об их присутствии. При этом возможны две формы декларации:

- описание, не приводящее к выделению памяти;
- определение, при котором под объект выделяется объем памяти в соответствии с его типом; в этом случае объект можно инициализировать, т.е. задать его начальное значение.

Кроме констант, заданных в исходном тексте, все объекты программы должны быть явно декларированы по следующему формату:

<атрибуты> <список ID объектов>;

элементы *списка* разделяются запятыми, а *атрибуты* – разделителями, например: *int i,j,k; float a,b;*

Объекты программы могут иметь следующие атрибуты:

<класс памяти> – характеристика способа размещения объектов в памяти (статическая, динамическая), определяет область видимости и время жизни переменной (по умолчанию *auto*), данные атрибуты будут рассмотрены позже;

<тип> – информация об объекте: объем выделяемой памяти, вид представления и допустимые над ним действия (по умолчанию *int*).

Класс памяти и тип – атрибуты необязательные и при их отсутствии (но не одновременно) устанавливаются по умолчанию.

Примеры декларации простых объектов:

int i,j,k; char r; double gfd;

Рассмотрим основные базовые типы данных более подробно.

4.4. Данные целого типа (*int*)

Тип *int* – целое число, обычно соответствующее естественному размеру целых чисел. Квалификаторы *short* и *long* указывают на различные размеры и определяют объем памяти, выделяемый под них (см. табл.1), например: *short x;*

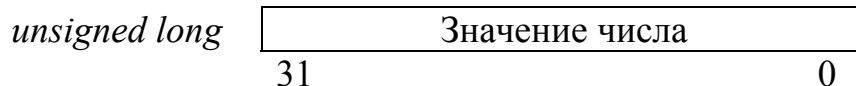
long x;

unsigned x = 8; – декларация с инициализацией числом 8.

Атрибут *int* в этих случаях может быть опущен.

Атрибуты *signed* и *unsigned* показывают, как интерпретируется старший бит числа – как знак или как часть числа:

<i>int</i>	Знак		Значение числа															
	15		14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	– номера бит
<i>unsigned int</i>	Значение числа																	
	15 0																	
<i>long</i>	Знак		Значение числа															
	31		30															0



Если указан только атрибут *int*, это означает *short signed int*.

4.5. Данные символьного типа (*char*)

Символьная переменная занимает в памяти один байт. Закрепление конкретных символов за кодами производится кодовыми таблицами.

Для ПЭВМ наиболее распространена таблица кодов ASCII – *American Standard Code for Information Interchange* (прил. 1). Данные типа *char* рассматриваются компилятором как целые, поэтому возможно использование *signed char* (по умолчанию) – символы с кодами от –128 до +127 и *unsigned char* – символы с кодами от 0 до 255.

Примеры: *char res, simv1, simv2;*
char let = 's'; – декларация с инициализацией символом *s*.

4.6. Данные вещественного типа (*float, double*)

Данные вещественного типа в памяти занимают: *float* – 4 байта; *double* – 8 байт; *long double* (повышенная точность) – 10 байт. Для размещения данных типа *float* обычно 8 бит выделено для представления порядка и знака и 24 бита под мантиссу, табл. 2.

Таблица 2

Тип	Точность (мантисса)	Порядок
<i>float</i>	7 цифр после запятой	± 38
<i>double</i>	15	± 308
<i>Long double</i>	19	± 4932

5. Константы в программах

Константы – объекты, *не подлежащие использованию в левой части оператора присваивания*, т.к. константа – неадресуемая величина. В языке Си константами являются:

- самоопределенные арифметические константы целого и вещественного типов, символьные и строковые данные;
- идентификаторы массивов и функций;
- элементы перечислений.

5.1. Целочисленные константы

Общий формат: $\pm n$ (+ обычно не ставится).

Десятичные константы – последовательность цифр 0...9, первая из которых не должна быть 0. Например, 22 и 273 – обычные целые константы, если нуж-

но ввести длинную целую константу, то указывается признак $L(l) - 273L$ ($273l$). Для такой константы будет отведено – 4 байта. Обычная целая константа, которая слишком длинна для типа *int*, рассматривается как *long*.

Существует система обозначений для восьмеричных и шестнадцатеричных констант.

Восьмеричные константы – последовательность цифр от 0 до 7, первая из которых должна быть 0, например: $020 = 16$ – десятичное.

Шестнадцатеричные константы – последовательность цифр от 0 до 9 и букв от *A* до *F* (*a...f*), начинающаяся символами *0X* (*0x*), например: $0X1F$ ($0x1f$) = 31 – десятичное.

Восьмеричные и шестнадцатеричные константы могут также заканчиваться буквой $L(l) - long$, например, $020L$ или $0X20L$.

Примеры целочисленных констант:

1992	13, 777	1000L	– десятичные;
0777	00033	01 l	– восьмеричные;
0x123	0X00ff	0xb8000l	– шестнадцатеричные.

5.2. Константы вещественного типа

Данные константы размещаются в памяти по формату *double*, а во внешнем представлении могут иметь две формы:

1) с фиксированной десятичной точкой, формат записи: $\pm n.m$, где *n*, *m* – целая и дробная части числа;

2) с плавающей десятичной точкой (экспоненциальная форма): $\pm n.mE\pm p$, где *n*, *m* – целая и дробная части числа, *p* – порядок; $\pm 0.xxxE\pm p$ – нормализованный вид, например, $1,25 \cdot 10^{-8} = 0.125E-8$.

Примеры констант с фиксированной и плавающей точками:

1.0	-3.125	100e-10	0.12537e+13
-----	--------	---------	-------------

5.3. Символьные константы

Символьная константа – это символ, заключенный в одинарные кавычки: *'A'*, *'x'* (тип *char* → целое *int*).

Также используются специальные последовательности символов – управляющие (*escape*) последовательности, основные из них: $\backslash n$ – новая строка, $\backslash t$ – горизонтальная табуляция, $\backslash 0$ – нулевой символ (пусто).

При присваивании символьной переменной они должны быть заключены в апострофы. Константа $\backslash 0$, изображающая символ 0 (пусто), часто записывается вместо целой константы 0, чтобы подчеркнуть символьную природу некоторого выражения.

Текстовые символы непосредственно вводятся с клавиатуры, а специальные и управляющие – представляются в исходном тексте парами символов, например: $\backslash \backslash$ – обратный слеш; $\backslash '$ – апостроф; $\backslash ''$ – кавычки.

Примеры символьных констант: *'A'*, *'9'*, *'\$'*, *'\n'*, *'\72'*.

5.4. Строковые константы

Строковая константа представляет собой последовательность символов кода ASCII, заключенную в кавычки ("). Во внутреннем представлении к строковым константам добавляется нулевой символ '\0', называемый нуль-терминатор, отмечающий конец строки. Кавычки не являются частью строки, а служат только для ее ограничения. Строка в языке Си представляет собой массив, состоящий из символов. Внутреннее представление константы "01234\0ABCDEF": '0' '1' '2' '3' '4' '\0' 'A' 'B' 'C' 'D' 'E' 'F' '\0'

Примеры строковых констант:

"Система", "\n\t Аргумент \n", "Состояние \"WAIT\""

В конец строковой константы компилятор автоматически помещает нуль-символ, который не является цифрой 0, на печать не выводится, в таблице кодов ASCII имеет код = 0.

Например, строка " " – пустая строка (нуль-строка).

6. Обзор операций

6.1. Операции, выражения

Выражения используются для вычисления значений (определенного типа) и состоят из операндов, операций и скобок. Каждый операнд может быть, в свою очередь, выражением.

Знак операции – это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются.

Операции делятся на унарные, бинарные и тернарные – по количеству участвующих в них операндов; выполняются в соответствии с приоритетами – для изменения порядка выполнения операций используются круглые скобки.

Большинство операций выполняется слева направо, например, $a+b+c \rightarrow (a+b)+c$. Исключение: унарные операции, операции присваивания и тернарная условная операция (?:) – справа налево.

Полный список операций приводится в прил. 2.

6.2. Арифметические операции

Арифметические операции – *бинарные*, их обозначения:

+ (сложение); – (вычитание); / (деление, для *int* операндов – с отбрасыванием остатка); * (умножение); % (остаток от деления целочисленных операндов со знаком первого операнда – деление «по модулю»).

Операндами традиционных арифметических операций (+ – * /) могут быть константы, переменные, функции, элементы массивов, указатели, любые арифметические выражения.

Порядок выполнения операций:

- 1) выражения в круглых скобках;
- 2) функции (стандартные математические, функции пользователя);

3) операции $*$ / (выполняются слева направо);

4) операции $- +$ (слева направо).

Унарные операции $+, -$ (знак числа) обладают самым высоким приоритетом, определены только для целых и вещественных операндов, «+» носит только информационный характер, «-» меняет знак операнда на противоположный.

Таким образом, т.к. операции $*, /, \%$ обладают высшим приоритетом над операциями $+, -$, при записи сложных выражений нужно использовать общепринятые математические правила: $x+y \cdot z - \frac{a}{b+c} \leftrightarrow x+y \cdot z - a/(b+c)$.

6.3. Операции присваивания

Формат операции присваивания:

$\langle ID \rangle = \langle \text{выражение} \rangle;$

Присваивание значения в языке Си в отличие от традиционной интерпретации рассматривается как *выражение*, имеющее значение левого операнда после присваивания. Таким образом, присваивание может включать несколько операций присваивания, изменяя значения нескольких операндов, например:

int i, j, k ;

float x, y, z ;

...

$i = j = k = 0;$ $\leftrightarrow k = 0, j = k, i = j;$

$x = i + (y = 3) - (z = 0);$ $\leftrightarrow z = 0, y = 3, x = i + y - z;$

Внимание! Левым операндом операции присваивания может быть только именованная либо косвенно адресуемая указателем переменная. Примеры недопустимых выражений:

а) присваивание константе: $2 = x + y;$

б) присваивание функции: $getch() = i;$

в) присваивание результату операции: $(i+1) = 2 + y;$

6.4. Сокращенная запись операции присваивания

В языке Си используются два вида сокращений записи операции присваивания:

а) вместо записи $v = v \# e;$

где $\#$ – арифметическая операция (операция над битовым представлением операндов), рекомендуется использовать запись $v \# = e;$

например, $i = i + 2;$ $\leftrightarrow i += 2;$ (**знаки операций без пробелов**);

б) вместо записи $x = x \# 1;$

где $\#$ – символы, обозначающие операцию инкремента (+), либо декремента (-), x – целочисленная переменная (переменная-указатель), рекомендуется использовать запись:

$##x;$ – префиксную или $x##;$ – постфиксную.

Если эти операции используются в чистом виде, то различий между постфиксной и префиксной формами нет. Если же они используются в выражении, то в префиксной форме ($##x$) сначала значение x изменится на 1, а затем будет использовано в выражении; в постфиксной форме ($x##$) сначала значение использу-

ется в выражении, а затем изменяется на 1. Операции над указателями рассмотрим позже.

Пример 1:

```
int i,j,k;
float x,y;
...
x* = y;
i+ = 2;
x/ = y+15;
--k;
--k;
j = i++;
j = ++i;
```

Смысл записи

```
x = x*y;
i = i+2;
x = x/(y+15);
k = k-1;
k = k-1;
j = i;    i = i+1;
i = i+1;  j = i;
```

Пример 2:

```
int n,a,b,c,d;
n = 2; a = b = c = 0;
a = ++n;
a+ = 2;
b = n++;
b- = 2;
c = --n;
c* = 2;
d = n--;
d% = 2;
```

Значения

```
n=3, a=3
a=5
b=3, n=4
b=1
n=3, c=3
c=6
d=3, n=2
d=1
```

6.5. Преобразование типов операндов арифметических операций

В операциях могут участвовать операнды различных типов, в этом случае они преобразуются к общему типу в порядке увеличения их "размера памяти", т.е. объема памяти, необходимого для хранения их значений. Поэтому неявные преобразования всегда идут от "меньших" объектов к "большим". Схема выполнения преобразований операндов арифметических операций:

```
short, char    → int    → unsigned    → long    → double
float          → double
```

Стрелки отмечают преобразования даже однотипных операндов перед выполнением операции, т.е. действуют следующие правила:

- значения типов *char* и *short* всегда преобразуются в *int*;
- если любой из операндов (*a* или *b*) имеет тип *double*, то второй преобразуется в *double*;
- если один из операндов *long*, то другой преобразуется в *long*.

Внимание! Результатом $1/3$ будет «0», чтобы избежать такого рода ошибок необходимо явно изменять тип хотя бы одного операнда, т.е. записывать, например: $1. / 3$.

Типы *char* и *int* могут свободно смешиваться в арифметических выражениях. Каждая переменная типа *char* автоматически преобразуется в *int*, что обеспечивает значительную гибкость при проведении определенных преобразований символов.

При присваивании значение правой части преобразуется к типу левой, который и является типом результата. И здесь необходимо быть внимательным, т.к. при некорректном использовании операций присваивания могут возникнуть неконтролируемые ошибки. Так, при преобразовании *int* в *char* старший байт просто отбрасывается.

Пусть имеются значения: *float* *x*; *int* *i*; тогда $x=i$; и $i=x$; приводят к преобразованиям, причем *float* преобразуется в *int* отбрасыванием дробной части.

Тип *double* преобразуется во *float* округлением.

Длинное целое преобразуется в более короткое целое и *char* посредством отбрасывания лишних бит более высокого порядка.

При передаче данных функциям также происходит преобразование типов: в частности, *char* становится *int*, а *float* – *double*.

6.6. Операция приведения типа

В любом выражении преобразование типов может быть осуществлено явно, для этого достаточно перед выражением поставить в скобках идентификатор соответствующего типа.

Вид записи операции: $(\text{тип}) \text{ выражение};$
ее результат – значение выражения, преобразованное к заданному типу.

Операция приведения типа вынуждает компилятор выполнить указанное преобразование, но ответственность за последствия возлагается на программиста. Рекомендуются использовать эту операцию в исключительных случаях, например:

float *x*;

int *n*=6, *k*=4;

$x=(n+k)/3;$ → $x=3$, т.к. дробная часть будет отброшена;

$x=(\text{float})(n+k)/3;$ → $x=3.333333$ – использование операции приведения типа позволяет избежать округления результата деления целочисленных операндов.

6.7. Операции сравнения

$==$	– равно или эквивалентно;	$!=$	– не равно;
$<$	– меньше;	$<=$	– меньше либо равно;
$>$	– больше;	$>=$	– больше либо равно.

Пары символов соответствующих операций разделять нельзя.

Общий вид операций отношений:

$\langle \text{выражение 1} \rangle \langle \text{знак операции} \rangle \langle \text{выражение 2} \rangle$

Общие правила:

- операндами могут быть любые базовые (скалярные) типы;
- значения *выражений* перед сравнением преобразуются к одному типу;
- результат операции отношения – значение 1, если отношение истинно, или 0 в противном случае (ложно). Следовательно, операция отношения может использоваться в любых арифметических выражениях.

6.8. Логические операции

Логические операции (в порядке убывания относительного приоритета) и их обозначения:

$!$	– отрицание (логическое НЕТ);
$\&\&$	– конъюнкция (логическое И);
$ $	– дизъюнкция (логическое ИЛИ).

Общий вид операции отрицания:

$!\langle \text{выражение} \rangle$

Общий вид операций конъюнкции и дизъюнкции

$\langle \text{выражение 1} \rangle \langle \text{операция} \rangle \langle \text{выражение 2} \rangle$

Например:

$y > 0 \ \&\& \ x = 7 \rightarrow$ истина, если 1-е и 2-е выражения истинны;

$e > 0 \ || \ x = 7 \rightarrow$ истина, если хотя бы одно выражение истинно.

Ненулевое значение операнда – *истина*, а нулевое – *ложь*, например:

$!0 \rightarrow 1$

$!5 \rightarrow 0$

$x = 10;$

$!((x=y)>0) \rightarrow 0$

Особенность операций конъюнкции и дизъюнкции – экономное последовательное вычисление выражений-операндов:

$\langle \text{выражение 1} \rangle \langle \text{операция} \rangle \langle \text{выражение 2} \rangle$

– если *выражение 1* операции конъюнкции ложно, то результат операции ноль и *выражение 2* не вычисляется;

– если *выражение 1* операции дизъюнкции истинно, то результат операции единица и *выражение 2* не вычисляется.

Пример правильной записи двойного неравенства:

$0 < x < 100 \leftrightarrow (0 < x) \ \&\& \ (x < 100)$

6.9. Побитовые логические операции, операции над битами

В Си предусмотрен набор операций для работы с отдельными битами. Эти операции нельзя применять к переменным вещественного типа.

Операции над битами и их обозначения:

\sim – дополнение (унарная операция); инвертирование (одноместная операция);

$\&$ – побитовое И – конъюнкция;

$|$ – побитовое включающее ИЛИ – дизъюнкция;

\wedge – побитовое исключающее ИЛИ – сложение по модулю 2;

\gg – сдвиг вправо;

\ll – сдвиг влево.

Общий вид операции инвертирования:

$\sim \langle \text{выражение} \rangle$

Остальные операции над битами имеют вид

$\langle \text{выражение 1} \rangle \langle \text{знак операции} \rangle \langle \text{выражение 2} \rangle$

Операндами операций над битами могут быть только *выражения*, приводимые к целому типу. Операции (\sim , $\&$, $|$, \wedge) выполняются поразрядно над всеми битами операндов (знаковый разряд особо не выделяется):

$\sim 0xF0 \leftrightarrow x0F$

$0xFF \ \& \ 0x0F \leftrightarrow x0F$

$0xF0 \ | \ 0x11 \leftrightarrow xF1$

$0xF4 \ \wedge \ 0xF5 \leftrightarrow x01$

Операция $\&$ часто используется для маскирования некоторого множества бит. Например, оператор $w = n \& 0177$ передает в w семь младших бит n , полагая остальные равными нулю.

Операции сдвига выполняются также для всех разрядов с потерей выходящих за границы бит.

Операция $(!)$ используется для включения бит $w = x ! y$, устанавливает в единицу те биты в x , которые $=1$ в y .

Необходимо отличать побитовые операции $\&$ и $!$ от логических операций $\&\&$ и $||$, если $x=1$, $y=2$, то $x \& y$ равно нулю, а $x \&\& y$ равно 1.

$$0x81 \ll 1 \quad \leftrightarrow \quad 0x02$$

$$0x81 \gg 1 \quad \leftrightarrow \quad 0x40$$

Если *выражение* 1 имеет тип *unsigned*, то при сдвиге вправо освобождающиеся разряды гарантированно заполняются нулями (логический сдвиг). Выражения типа *signed* могут, но необязательно, сдвигаться вправо с копированием знакового разряда (арифметический сдвиг). При сдвиге влево освобождающиеся разряды всегда заполняются нулями. Если *выражение* 2 отрицательно либо больше длины *выражения* 1 в битах, то результат операции сдвига не определен.

Унарная операция (\sim) дает дополнение к целому, т.е. каждый бит со значением 1 получает значение 0 и наоборот. Эта операция оказывается полезной в выражениях типа

$$X \& (\sim)077,$$

где последние 6 бит X маскируются нулем.

Операции сдвига \ll и \gg осуществляют соответственно сдвиг вправо (влево) своего левого операнда на число позиций, задаваемых правым операндом, например, $x \ll 2$ сдвигает x влево на две позиции, заполняя освобождающиеся биты нулями (эквивалентно умножению на 4).

Операции сдвига вправо на k разрядов весьма эффективны для деления, а сдвиг влево – для умножения целых чисел на 2 в степени k :

$$x \ll 1 \quad \leftrightarrow \quad x * 2; \quad x \gg 1 \quad \leftrightarrow \quad x / 2$$

$$x \ll 3 \quad \leftrightarrow \quad x * 8$$

Подобное применение операций сдвига безопасно для беззнаковых и положительных значений *выражения* 1.

В математическом смысле операнды логических операций над битами можно рассматривать как отображение некоторых множеств с размерностью не более разрядности операнда на значения $\{0,1\}$.

Пусть единица означает обладание элемента множества некоторым свойством, тогда очевидна теоретико-множественная интерпретация рассматриваемых операций:

\sim – дополнение; $|$ – объединение; $\&$ – пересечение.

Простейшее применение – проверка нечетности целого числа:

`int i;`

`...
if (i & 1) printf (" Значение i нечетно!");`

Комбинирование операций над битами с арифметическими операциями часто позволяет упростить выражения.

6.10. Операция «,» (запятая)

Данная операция используется при организации строго гарантированной последовательности вычисления выражений (используется там, где по синтаксису допустима только одна операция, а нам необходимо разместить две и более, например, в операторе *for*). Форма записи:

выражение 1, ..., выражение N;

выражения 1,...,N вычисляются последовательно и результатом операции становится значение *выражения N*, например:

$m=(i=1, j=i++, k=6, n=i+j+k);$

получим последовательность вычислений: $i=1, j=i=1, i=2, k=6, n=2+1+6$, и в результате $m=n=9$.

7. Обзор базовых инструкций языка Си

7.1. Стандартная библиотека языка Си

В любой программе кроме операторов и операций используются средства библиотек, входящих в среду программирования, которые облегчают создание программ.

Часть библиотек стандартизована и поставляется с компилятором.

В стандартную библиотеку входят функции, макросы, глобальные константы. Это файлы с расширением **.h*, хранящиеся в папке *include*.

Рассмотрим наиболее часто используемые функции из стандартной библиотеки языка Си.

7.2. Стандартные математические функции

Математические функции языка Си декларированы в файлах *math.h* и *stdlib.h*.

В большинстве приведенных здесь функций аргументы x, y и результат выполнения имеют тип *double*. Аргументы тригонометрических функций должны быть заданы в радианах (2π радиан = 3600), табл. 3.

Таблица 3

Математическая функция	ID функции в языке Си
\sqrt{x}	<code>sqrt(x)</code>
$ x $	<code>fabs(x)</code>
e^x	<code>exp(x)</code>
x^y	<code>pow(x,y)</code>
$\ln(x)$	<code>log(x)</code>

$\lg_{10}(x)$	$\log_{10}(x)$
$\sin(x)$	$\sin(x)$
$\cos(x)$	$\cos(x)$
$\operatorname{tg}(x)$	$\tan(x)$
$\arcsin(x)$	$\operatorname{asin}(x)$
$\arccos(x)$	$\operatorname{acos}(x)$
$\operatorname{arctg}(x)$	$\operatorname{atan}(x)$
$\operatorname{arctg}(x / y)$	$\operatorname{atan2}(x)$
$\operatorname{sh}(x)=0.5 (e^x - e^{-x})$	$\sinh(x)$
$\operatorname{ch}(x)=0.5 (e^x + e^{-x})$	$\cosh(x)$
$\operatorname{tgh}(x)$	$\tanh(x)$
остаток от деления x на y	$\operatorname{fmod}(x,y)$
наименьшее целое $\geq x$	$\operatorname{ceil}(x)$
наибольшее целое $\leq x$	$\operatorname{floor}(x)$

7.3. Функции вывода данных

Для вывода информации на экран монитора в языке Си чаще всего используются функции *printf* и *puts*.

Формат функции форматного вывода на экран:

printf (“управляющая строка” , список объектов вывода);

– в *управляющей строке*, заключенной в кавычки, записывают поясняющий текст; список модификаторов форматов, указывающих компилятору способ вывода объектов (признаком модификатора формата является символ **%**) и специальные (управляющие) символы;

– в *списке объектов вывода* указываются идентификаторы печатаемых объектов, разделенных запятыми: переменные, константы или выражения, вычисляемые перед выводом.

Количество и порядок следования форматов должен совпадать с количеством и порядком следования печатаемых объектов.

Функция *printf* выполняет вывод данных в соответствии с указанными форматами, поэтому формат может использоваться и для преобразования типов выводимых объектов.

Если признака модификации (%) нет, то вся информация выводится как комментарии.

Основные модификаторы формата:

- %d (%i)** – десятичное целое число;
- %c** – один символ;
- %s** – строка символов;
- %f** – число с плавающей точкой, десятичная запись;
- %e** – число с плавающей точкой, экспоненциальная запись;
- %g** – используется вместо *f*, *e* для исключения незначащих нулей;
- %o** – восьмеричное число без знака;
- %x** – шестнадцатеричное число без знака.

Для типов *long* и *double* добавляется символ *l*, например, *%ld* – длинное целое, *%lf* – число вещественное с удвоенной точностью.

Если нужно напечатать сам символ *%*, то его следует указать 2 раза.

```
printf("Только %d%% предприятий не работало. \n",5);
```

Получим: Только 5% предприятий не работало.

Управляют выводом специальные последовательности символов: *\n* – новая строка; *\t* – горизонтальная табуляция; *\b* – шаг назад; *\r* – возврат каретки; *\v* – вертикальная табуляция; ** – обратная косая; *\'* – апостроф; *\"* – кавычки; *\0* – нулевой символ (пусто).

В модификаторах формата функции *printf* после символа *%* можно указывать строку цифр, задающую минимальную ширину поля вывода, например: *%5d* (для целых), *%4.2f* (для вещественных – две цифры после запятой для поля, шириной 4 символа). Если указанной ширины не хватает, происходит автоматическое расширение.

Можно использовать функцию *printf* для нахождения кода ASCII некоторого символа:

```
printf(" %c - %d\n",'a','a');
```

получим десятичный код ASCII символа *a*: *a* - 65

Функция *puts* выводит на экран дисплея строку символов, автоматически добавляя к ней символ перехода на начало новой строки (*\n*).

Функция *putchar* выдает на экран дисплея один символ без добавления символа *'\n'*.

7.4. Функции ввода информации

Функция, предназначенная для форматированного ввода исходной информации с клавиатуры:

```
scanf("управляющая строка", список объектов ввода);
```

в управляющей строке указываются **только модификаторы форматов**, количество, тип и порядок следования которых должны совпадать с количеством, типом и порядком следования вводимых объектов, иначе результат ввода непредсказуем.

Список объектов ввода представляет собой **адреса** переменных, разделенные запятыми, т.е. для ввода значения переменной перед ее идентификатором указывается символ *&*, обозначающий «взять адрес».

Если нужно ввести значение строковой переменной, то использовать символ *&* не нужно, т.к. строка – это массив символов, а ID массива эквивалентно адресу его первого элемента. Например:

```
int course;
float grant;
char name[20];
printf(" Укажите курс, стипендию, имя \n ");
scanf("%d%f%s",&course, &grant, name);
```

Вводить данные с клавиатуры можно как в одной строке через пробелы, так и в разных строках.

Функция *scanf* использует практически тот же набор модификаторов форматов, что и *printf*, отличия – отсутствует формат %g, форматы %e,%f – эквивалентны.

Внимание! Функцией *scanf* (формат %s) строка вводится только до первого пробела.

Для ввода фраз, состоящих из слов, используется функция *gets* (ID строковой переменной);

7.5. Ввод - вывод потоками

Поток – это абстрактное понятие расширенной версии языка Си, которое относится к любому переносу данных от источника к приемнику.

Для ввода-вывода используются две переопределенные операции побитового сдвига << , >>. Формат записи:

cout << ID переменной ;

cin >> ID переменной ;

Стандартный поток вывода *cout* – по умолчанию подключен к монитору, ввода *cin* – к клавиатуре. Для их работы необходимо подключить файл *iostream.h*.

Пример:

```
#include<iostream.h>
#include<conio.h>
void main (void) {
    cout << " Hello! " << endl;    // end line – переход на новую строку
    cout << " Input i, j ";
    int i, j, k;
    cin >> i >> j ;
    k = i + j ;
    cout << " Sum i , j = " << k << endl;
}
```

8. Синтаксис операторов языка Си

Операторы языка Си можно разделить на три группы: операторы-декларации (рассмотрены ранее); операторы преобразования объектов; операторы управления процессом выполнения алгоритма.

Программирование процесса преобразования объектов производится посредством записи выражений.

Простейший вид операторов – выражение, заканчивающееся символом «;» (точка с запятой).

Простые операторы: оператор присваивания (выполнение операций присваивания); оператор вызова функции (выполнение операции вызова функции); пустой оператор «;».

Когда необходимо связать фразу *else* с внешним *if*, то используем операторные скобки:

```
if(n>0)
    { if(a>b) z=a; }
    else z=b;
```

В следующей цепочке операторов *if-else-if* выражения просматриваются последовательно:

```
if(выражение 1) оператор 1;
else if(выражение 2) оператор 2;
else if(выражение 3) оператор 3;
else оператор 4;
```

Если какое-то *выражение* оказывается истинным, то выполняется относящийся к нему *оператор* и этим вся цепочка заканчивается. Последняя часть с *else* – случай, когда ни одно из проверяемых условий не выполняется. Когда при этом не нужно предпринимать никаких явных действий, *else оператор 4*; может быть опущен, или его можно использовать для контроля, чтобы засечь "невозможное" условие.

Пример:

```
if( n < 0 ) printf( "N отрицательное\n" );
else if( n == 0 ) printf( "N равно нулю\n" );
else printf( "N положительное \n" );
```

8.2. Условная операция «? :»

Условная операция – тернарная, в ней участвуют три операнда. Формат написания условной операции:

выражение 1 ? выражение 2 : выражение 3;

если *выражение 1* отлично от нуля (истинно), то результатом операции является *выражение 2*, в противном случае – *выражение 3*; каждый раз вычисляется только одно из выражений 2 или 3.

Для нахождения максимального значения из *a* и *b* (значение *z*) можно использовать оператор *if*:

```
if(a > b) z=a;
else z=b;
```

Используя условную операцию, этот пример можно записать как

```
z = (a>b) ? a : b;
```

Условную операцию можно использовать так же, как и любое другое выражение. Если выражения 2 и 3 имеют разные типы, то тип результата определяется по общим правилам.

8.3. Оператор выбора альтернатив (переключатель)

Общий вид оператора:

```
switch (выражение) {  
    case константа 1: оператор 1; break;  
    case константа 2: оператор 2; break;  
    ...  
    case константа N: оператор N; break;  
    default: оператор N+1; break; – может отсутствовать  
}
```

Значение вычисленного *выражения* должно быть целого типа (символьного). Это значение (константа выбора) сравнивается со значениями *констант*, стоящих после *case*, и при совпадении с одной из них выполняется передача управления соответствующему *оператору*. В случае несовпадения значения *выражения* с одной из *констант* происходит переход на *default* либо при отсутствии *default* – к оператору, следующему за оператором *switch*. Оператор *break* (разрыв) выполняет выход из оператора *switch*; *break* может отсутствовать.

Пример 1 с использованием оператора *break*:

```
void main(void)  
{ int i = 2;  
    switch(i)    {  
        case 1: puts ( "Случай 1. "); break;  
        case 2: puts ( "Случай 2. "); break;  
        case 3: puts ( "Случай 3. "); break;  
        default: puts ( "Случай default. "); break;  
    }  
}
```

Для того чтобы выйти из оператора *switch*, использовали оператор *break*, поэтому результатом будет Случай 2.

Пример 2 (оператор *break* отсутствует):

```
void main(void)  
{ int i=2;  
    switch(i)    {  
        case 1: puts ( "Случай 1. ");  
        case 2: puts ( "Случай 2. ");  
        case 3: puts ( "Случай 3. ");  
        default: puts ( "Случай default. ");  
    }  
}
```

Так как оператор разрыва отсутствует, результатом будет:

Случай 2.

Случай 3.

Случай default.

9. Составление циклических алгоритмов

9.1. Понятие цикла

Практически все алгоритмы решения задач содержат циклически повторяемые участки. Цикл – это одно из фундаментальных понятий программирования. Под циклом понимается организованное повторение некоторой последовательности операторов.

Для организации циклов используются специальные операторы:

- оператор цикла с предусловием;
- оператор цикла с постусловием;
- оператор цикла с предусловием и коррекцией.

Любой цикл состоит из кода цикла, т.е. тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла.

Один проход цикла называется итерацией. Проверка условия выполняется на каждой итерации либо до кода цикла (с предусловием), либо после кода цикла (с постусловием).

9.2. Оператор с предусловием *while*

Общий вид записи:

while (*выражение*) *код цикла*;

Если *выражение* – истинно (не равно 0), то выполняется *код цикла*; это повторяется до тех пор, пока *выражение* не примет значение 0 (ложь) – в этом случае выполняется оператор, следующий за *while*. Если *выражение* ложно (равно 0), то цикл не выполнится ни разу.

Код цикла может включать любое количество управляющих операторов, связанных с конструкцией *while*, взятых в фигурные скобки (блок), если их более одного. Среди этих операторов могут быть *continue* – переход к следующей итерации цикла и *break* – выход из цикла.

Например, необходимо сосчитать количество символов в строке. Предполагается, что входной поток настроен на начало строки. Тогда подсчет символов выполняется следующим образом:

```
char ch;  
int count=0;  
while (( ch = getchar() ) != '\n') count++;
```

Для выхода из цикла *while* при истинности *выражения*, как и для выхода из других циклов, можно пользоваться оператором *break*.

Пример 1:

```
while (1) {                                     – организация бесконечного цикла  
    ...  
    if ( kbhit() && (getch() == 27) ) break;
```

если нажата клавиша (результат работы функции *kbhit()*>0) и код ее равен 27 – код клавиши “Esc”, то выходим из цикла;

```
...  
}  
Пример 2:
```

```
...  
while ( !kbhit() ); – выполнять до тех пор, пока не нажата клавиша;  
...
```

9.3. Оператор цикла с постусловием *do – while*

Общий вид записи:

do код цикла *while* (выражение);

код цикла будет выполняться до тех пор, пока *выражение* истинно. Все, что говорилось выше, справедливо и здесь, за исключением того, что данный цикл всегда выполняется хотя бы один раз.

9.4. Оператор цикла с предусловием и коррекцией *for*

Общий вид оператора:

for (выражение 1; выражение 2; выражение 3) код цикла;

Цикл *for* эквивалентен последовательности инструкций:

```
выражение 1;  
while (выражение 2)  
{  
    код цикла . . .  
    выражение 3;  
}
```

здесь *выражение 1* – инициализация счетчика (начальное значение), *выражение 2* – условие продолжения счета, *выражение 3* – увеличение счетчика. Выражения 1, 2 и 3 могут отсутствовать (пустые выражения), но символы «;» опускать нельзя.

Например, для суммирования первых N натуральных чисел можно записать

```
sum = 0;  
for ( i=1; i<=N; i++) sum+=i;
```

Операция «запятая» чаще всего используется в операторе *for*. Она позволяет включать в его спецификацию несколько инициализирующих выражений. Предыдущий пример можно записать в виде

```
for ( sum=0 , i=1; i<=N; sum+= i , i++) ;
```

Оператор *for* имеет следующие возможности:

– можно вести подсчет с помощью символов, а не только чисел:

```
for (ch = 'a'; ch<='z'; ch++) ... ;
```

– можно проверить выполнение некоторого произвольного условия:

```
for (n = 0; s[i]>='0' && s[i]<'9'; i++) ... ;
```

или

```
for (n = 1; n*n*n <= 216; n++) ... ;
```

Первое выражение необязательно должно инициализировать переменную. Необходимо только помнить, что первое выражение вычисляется только один раз перед тем, как остальные части начнут выполняться.

```
for (printf(" Вводите числа по порядку! \n"); num != 6;)
    scanf("%d", &num);
printf(" Последнее число – это то, что нужно. \n");
```

В этом фрагменте первое сообщение выводится на печать один раз, а затем осуществляется прием вводимых чисел, пока не поступит число 6.

Параметры, входящие в выражения, находящиеся в спецификации цикла, можно изменять при выполнении операций в коде цикла, например:

```
for (n = 1; n < Nk; n += delta) ... ;
```

параметры *Nk*, *delta* можно менять в процессе выполнения цикла.

Использование условных выражений позволяет во многих случаях значительно упростить программу. Например:

```
for (i=0; i<n; i++)
    printf("%6d%c", a[i], (i%10==0) || (i==n-1) ? '\n' : ' ');
```

В этом цикле печатаются *n* элементов массива *a* по 10 в строке, разделяя каждый столбец одним пробелом и заканчивая каждую строку (включая последнюю) одним символом перевода строки. Символ перевода строки записывается после каждого десятого и *n*-го элементов. За всеми остальными – пробел.

10. Операторы передачи управления

Формально к операторам передачи управления относятся:

- оператор безусловного перехода **goto**;
- оператор перехода к следующему шагу (итерации) цикла **continue**;
- выход из цикла или оператора **switch** – **break**;
- оператор возврата из функции **return**.

Рассмотрим их более подробно.

10.1. Оператор безусловного перехода **goto**

В языке Си предусмотрен оператор **goto**, хотя в большинстве случаев можно обойтись без него. Общий вид оператора

```
goto метка;
```

Он предназначен для передачи управления на оператор, помеченный *меткой*. Метка представляет собой идентификатор, оформленный по всем правилам идентификации переменных с символом «двоеточие» после него, например, пустой помеченный оператор:

```
m1 : ;
```

Область действия метки – функция, где эта метка определена.

Программа с *goto* может быть написана без него за счет повторения некоторых проверок и введения дополнительных переменных.

Наиболее характерный случай использования оператора *goto* – выполнение прерывания (выхода) во вложенной структуре при возникновении грубых неисправимых ошибок во входных данных. И в этом случае необходимо выйти из двух (или более) циклов, где нельзя использовать непосредственно оператор *break*, т.к. он прерывает только самый внутренний цикл:

```
for (...)  
  for (...) { ...  
              if ( ошибка ) goto Error;  
            }
```

...

Error : – операторы для устранения ошибки;

Если программа обработки ошибок сложная, а ошибки могут возникать в нескольких местах, то такая организация оказывается удобной.

10.2. Оператор *continue*

Этот оператор может использоваться во всех типах циклов, но не в операторах *switch*. Наличие оператора *continue* вызывает пропуск "оставшейся" части итерации и переход к началу следующей, т.е. досрочное завершение текущего шага и переход к следующему шагу.

В циклах *while* и *do* это означает непосредственный переход к проверочной части. В цикле *for* управление передается на шаг коррекции, т.е. модификации выражения 3.

Фрагмент программы обработки только положительных элементов массива *a*, отрицательные значения пропускаются:

```
for ( i = 0; i<n; i++) {  
    if( a[i]<0) continue;  
    обработка положительных элементов;  
}
```

10.3. Оператор *break*

Оператор ***break*** производит экстренный выход из самого внутреннего цикла или оператора *switch*, к которому он принадлежит, и передает управление первому оператору, следующему за текущим оператором.

10.4. Оператор *return*

Оператор ***return***; производит досрочный выход из текущей функции. Он также возвращает значение результата функции:

return *выражение*;

В функциях, не возвращающих результат, он неявно присутствует после последнего оператора. Значение выражения при необходимости преобразуется к типу возвращаемого функцией значения.

11 . Указатели

Указатель – это переменная, которая может содержать адрес некоторого объекта. Указатель объявляется следующим образом:

*<тип> * < ID переменной-указателя>;*

Например: `int *a; double *f; char *w;`

С указателями связаны две унарные операции `&` и `*`.

Операция **&** означает «взять адрес» операнда. Операция ***** имеет смысл – «значение, расположенное по указанному адресу».

Обращение к объектам любого типа как операндам операций в языке Си может проводиться:

- по имени (идентификатору – ID);
- по указателю (операция косвенной адресации):

указатель = & ID объекта ;

Пример 1:

```
int x,           // переменная типа int
*y;             // указатель на элемент данных типа int
y=&x;            // y – адрес переменной x
*y=1;           // косвенная адресация указателем поля x, т.е.
                // по указанному адресу записать 1 → x=1;
```

Пример 2:

```
int i, j=8, k=5, *y;
y=&i;
*y=2;           // i=2
y=&j;
*y+=i;          // j+=i → j=j+i → j=j+2=10
y=&k;
k+=*y;          // k+=k → k=k+k = 10
(*y)++;         // k++ → k=k+1 = 10+1 = 11
```

При вычислении адресов объектов следует учитывать, что идентификаторы массивов и функций являются константными указателями. Такую константу можно присвоить переменной типа «указатель», но нельзя подвергать преобразованиям, например:

```
int x[100], *y;
y = x;          – присваивание константы переменной;
x = y;          – ошибка, в левой части – указатель-константа.
```

Указателю-переменной можно присвоить значение другого указателя либо выражения типа «указатель» с использованием при необходимости операции приведения типа. Приведение типа необязательно, если один из указателей имеет тип *void*.

Значение указателя можно вывести на экран с помощью спецификации `%p` (*pointer*), результат выводится в шестнадцатеричном виде.

Рассмотрим фрагмент программы:

```
int a=5, *p, *p1, *p2;
p=&a; p2=p1=p;
++p1; p2+=2;
printf("a=%d, p=%d, p=%p, p1=%p, p2=%p.\n", a, p, p, p1, p2);
```

Результат выполнения: a=5, *p=5, p=FFC8, p1=FFCC, p2=FFD0.

Графически это выглядит так (адреса взяты символически):

	4001	4003	4005	4007	4009	
4000	4002	4004	4006	4008	400A	
p	p1	p2				

$p = 4000, \quad p1 = 4002 = (4000 + 1 * sizeof(*p)) \rightarrow 4000+2(int)$

$p2 = 4004 = (4000 + 2* sizeof(*p)) \rightarrow 4000+2*2$

11.1. Операции над указателями (косвенная адресация)

Указатель может использоваться в выражениях вида

$$p \# iv, \quad \#\# p, \quad p \#\#, \quad p \# = iv,$$

где p – указатель, iv – целочисленное выражение, $\#$ – символ операций $+$ или $-$.

Значением таких выражений является увеличенное или уменьшенное значение указателя на величину $iv * sizeof(*p)$. Следует помнить, что операции с указателями выполняются в единицах памяти того типа объекта, на который ссылается этот указатель.

Текущее значение указателя ссылается на позицию некоторого объекта в памяти с учетом правил выравнивания для соответствующего типа данных. Таким образом, значение $p \# iv$ указывает на объект того же типа, расположенный в памяти со смещением на iv позиций.

При сравнении указателей могут использоваться операции отношения, наиболее важными являются отношения равенства или неравенства.

Отношения порядка имеют смысл только для указателей на последовательно размещенные объекты (элементы одного массива).

Разность двух указателей дает число объектов адресуемого ими типа в соответствующем диапазоне адресов. Очевидно, что уменьшаемый и вычитаемый указатели также должны соответствовать одному массиву, иначе результат операции не имеет практической ценности.

Любой указатель можно сравнивать со значением NULL, которое означает недействительный адрес, т.е. отсутствие адреса. Значение NULL можно присваивать указателю как признак пустого указателя, NULL заменяется препроцессором на выражение $(void *)0$.

11.2. Ссылка

Ссылка – это не тип данных, а константный указатель, т.е. это объект, который указывает на положение другой переменной.

Ссылка – это константный указатель, который отличается от переменного указателя тем, что для ссылки не требуется специальной операции разыменова-

ния. Над ссылкой арифметические операции запрещены, т.к. ссылка декларируется следующим образом:

тип &ID = *инициализатор*;

Инициализатор – это идентификатор объекта, на который в дальнейшем будет указывать ссылка. Например:

```
int a = 8;  
int &r = a;
```

Ссылка получила псевдоним объекта, указанного в качестве инициализатора. В данном примере одинаковыми будут следующие действия:

```
a++;      r++;
```

12. Массивы

В математике для удобства записи различных операций часто используют индексированные переменные: векторы, матрицы, тензоры. Так, вектор \vec{c} представляется набором чисел $(c_1 c_2 \dots c_n)$, называемых его компонентами, причем каждая компонента имеет свой номер, который принято обозначать в виде индекса. Матрица A – это таблица чисел $(a_{ij}, i=1, \dots, m; j=1, \dots, n)$, i – номер строки, j – номер столбца. Операции над матрицами и векторами обычно имеют короткую запись, которая обозначает определенные, порой сложные действия над их индексными компонентами. Например, произведение матрицы на вектор $\vec{b} = A \cdot \vec{c}, \quad b_i = \sum_{j=1}^n a_{ij} \cdot c_j$;

произведение двух матриц $D = A \cdot G, \quad d_{ij} = \sum_{k=1}^n a_{ik} \cdot g_{kj}$.

Введение индексированных переменных в языках программирования также позволяет значительно облегчить реализацию многих сложных алгоритмов, связанных с обработкой массивов однотипных данных.

В языке Си для этой цели имеется сложный тип переменных – **массив**, представляющий собой упорядоченную конечную совокупность элементов одного типа. Число элементов массива называют его размером. Каждый элемент массива определяется идентификатором массива и своим порядковым номером – индексом. Индекс – целое число, по которому производится доступ к элементу массива. Индексов может быть несколько. В этом случае массив называют многомерным, а количество индексов одного элемента массива является его размерностью.

12.1. Одномерные массивы

Индексы у одномерных массивов в языке Си начинаются с 0, а в программе одномерный массив объявляется следующим образом:

<тип> *<ID массива>*[*размер*] = {*список начальных значений*};

где *тип* – базовый тип элементов (целый, вещественный и т.д.);

размер – количество элементов в массиве.

Список начальных значений используется при необходимости инициализировать данные при объявлении, он может отсутствовать.

Размер массива может задаваться константой или константным выражением. Нельзя задавать массив переменного размера. Для этого существует отдельный механизм – динамическое выделение памяти.

Пример объявления массива целого типа: `int a[5];`

В данном массиве первый элемент – `a[0]`, второй – `a[1]`, ... пятый – `a[4]`.

Обращение к элементу массива в программе на языке Си осуществляется в традиционном для многих других языков стиле – записи операции обращения по индексу, например:

`a[0]=1; a[i]++; a[3] = a[i] + a[i+1];`

Декларация массива целого типа с инициализацией значений:

`int a[5]={2, 4, 6, 8, 10};`

Если в группе `{...}` список значений короче, то оставшимся элементам присваивается 0.

В языке Си с целью повышения быстродействия программы отсутствует механизм контроля границ изменения индексов массивов. При необходимости такой механизм должен быть запрограммирован явно.

12.2. Многомерные массивы

Декларация многомерного массива в общем виде:

`<тип> <ID>[размер 1][размер 2]...[размер N];`

Наиболее быстро изменяется последний индекс элементов массива, поскольку многомерные массивы в языке Си размещаются в памяти компьютера в последовательности столбцов.

Например, элементы двухмерного массива `b[3][2]` размещаются в памяти компьютера в таком порядке:

`b[0][0], b[0][1], b[1][0], b[1][1], b[2][0], b[2][1].`

Следующий пример иллюстрирует определение массива целого типа, состоящего из трех строк и четырех столбцов, с одновременной инициализацией:

`int a[3][4] = {{1,2 }, {9,-2,4,1},{-7 } };`

Если в какой-то группе `{ }` список значений короче, то оставшимся элементам присваивается 0.

12.3. Операция *sizeof*

Данная операция позволяет определить размер объекта по его идентификатору или типу, результатом является размер памяти в байтах (тип результата `int`). Формат записи:

`sizeof(параметр);`

где *параметр* – тип или ID объекта (не ID функции).

Если указан идентификатор сложного объекта (массив, структура, объединение), то получаем размер всего сложного объекта. Например:

`sizeof(int)` → размер памяти 2 байта,
`int b[5];`
`sizeof(b)` → размер памяти 10 байт.

Наиболее часто операция *sizeof* применяется при динамическом распределении памяти.

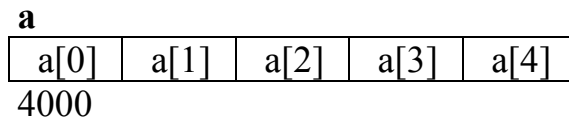
12.4. Применение указателей

Идентификатор массива – это адрес памяти, начиная с которого он расположен, т.е. адрес его первого элемента. Работа с массивами тесно связана с применением указателей.

Пусть объявлены массив *a* и указатель *q*:

`int a[5], *q;`

ID массива *a* является константным указателем на его начало.



Здесь приведено символическое изображение оперативной памяти, выделенной компилятором для объявленного целочисленного массива *a[5]*. Указатель *a* содержит адрес его начала в памяти, т.е. «символический адрес» = 4000 (*a*=4000).

Если выполнена операция *q=a;* – присваивание константы переменной, т.е. *q*=4000 (аналог: *q=&a[0]*), то с учетом адресной арифметики выражения *a[i]* и **(q+i)* приводят к одинаковым результатам – обращению к *i*-му элементу массива.

Идентификаторы *a* и *q* – указатели, очевидно, что выражения *a[i]* и **(a+i)* эквивалентны. Отсюда следует, что операция обращения к элементу массива по индексу применима и при его именовании переменной-указателем. Таким образом, для любых указателей можно использовать две эквивалентные формы выражений для доступа к элементам массива: *q[i]* и **(q+i)*. Первая форма удобнее для читаемости текста, вторая – эффективнее по быstroдействию программы.

Например, для получения значения 4-го элемента массива можно написать *a[3]* или **(a+3)*, результат будет один и тот же.

Очевидна эквивалентность выражений:

1) получение адреса начала массива в памяти:

$\&a[0] \leftrightarrow \&(*a) \leftrightarrow a$

2) обращение к первому элементу массива:

$*a \leftrightarrow a[0]$

Последнее объясняет правильность выражения для определения реального количества элементов массива с объявленной размерностью:

`char s[256];` – размерность должна быть константой;

...
`int kol = sizeof(s)/sizeof(*s);` – количество элементов массива *s*.

Указатели, как и переменные любого другого типа, могут объединяться в массивы.

Объявление массива указателей на целые числа имеет вид

```
int *a[10], y;
```

Теперь каждому элементу массива *a* можно присвоить адрес целочисленной переменной *y*, например *a[1]=&y*; чтобы найти значение переменной *y* через данный элемент массива *a*, необходимо записать **a[1]*.

12.5. Указатели на указатели

В языке Си можно описать переменную типа «указатель на указатель». Это ячейка оперативной памяти, в которой будет храниться адрес указателя на какую-либо переменную. Признак такого типа данных – повторение символа «*» перед идентификатором переменной. Количество символов «*» определяет уровень вложенности указателей друг в друга. При объявлении указателей на указатели возможна их одновременная инициализация. Например:

```
int a=5;
int *p1=&a;
int **pp1=&p1;
int ***ppp1=&pp1;
```

Теперь присвоим целочисленной переменной *a* новое значение (например, 10), одинаковое присваивание произведут следующие операции:

```
a=10; *p1=10; **pp1=10; ***ppp1=10;
```

Для доступа к области памяти, отведенной под переменную *a*, можно использовать и индексы. Справедливы следующие аналоги, если мы работаем с многомерными массивами:

```
*p1 ↔ p1[0] **pp1 ↔ pp1[0][0] ***ppp1 ↔ ppp1[0][0][0]
```

Отметим, что идентификатор двухмерного массива – это указатель на массив указателей (переменная типа «указатель на указатель»: `int **m;`), поэтому выражение *a[i][j]* эквивалентно выражению **(*(m+i)+j)*.

Например, двухмерный массив *m*(3×4) компилятор рассматривает как массив четырех указателей, каждый из которых указывает на начало массива со значениями размером по три элемента каждый.

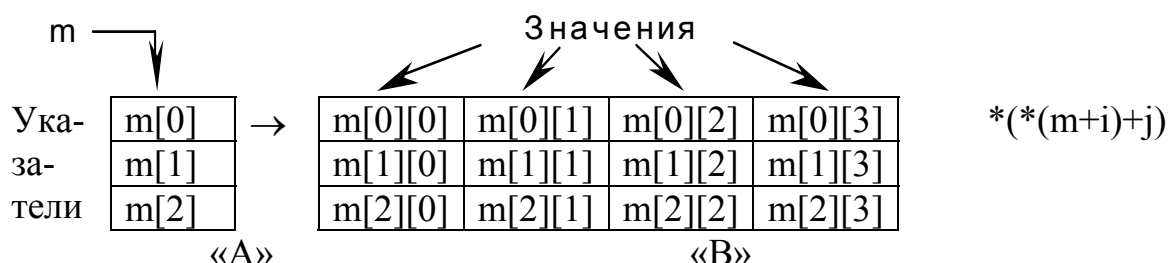


Рис. 4

Очевидна и схема размещения такого массива в памяти – последовательное (друг за другом) размещение *строк* – одномерных массивов со значениями. Аналогичным образом можно установить соответствие между указателями и массивами с произвольным числом измерений: `float name[][][];` ↔ `float ****name;`

Пример программы конструирования массива массивов:

```
#include <stdio.h>
int x0[4]={ 1, 2, 3, 4};
int x1[4]={11,12,13, 14};           // Декларация и инициализация
int x2[4]={21,22,23, 24};           // массивов целых чисел
int *y[3]={x0, x1, x2};             // Создание массива указателей
void main(void)
{   int i,j;
    for (i=0; i<3; i++) {
        printf("\n %2d",i);
        for (j=0; j<4; j++) printf(" %2d",y[i][j]);
    }
}
```

Результаты работы программы:

```
0) 1 2 3 4
1) 11 12 13 14
2) 21 22 23 24
```

Такие же результаты получим и при следующем объявлении массива:

```
int y[3][4]={
    { 1, 2, 3, 4},
    {11,12,13,14},           // Декларация и инициализация
    {21,22,23,24},           // массива массивов целых чисел
};
```

В последнем случае массив указателей на массивы создается компилятором. Здесь собственно данные массива располагаются в памяти последовательно по строкам, что является основанием для объявления массива у в виде

```
int z[3][4]={ 1, 2, 3, 4,
              11,12,13,14,           // Декларация и инициализация
              21,22,23,24};          // массива массивов целых чисел
```

Замена скобочного выражения `z[3][4]` на `z[12]` здесь не допускается, так как массив указателей в данном случае создан не будет.

13. Работа с динамической памятью

В языке Си размерность массива при объявлении должна задаваться константным выражением. При необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти (захватить память). После обработки массива занятую память надо освободить. Библиотечные функции работы с памятью описаны в файле *alloc.h*.

Рассмотрим некоторые функции для манипулирования динамической памятью в стандарте Си:

*void *calloc(unsigned n, unsigned size);* – выделение памяти для размещения *n* объектов размером *size* байт и заполнение полученной области нулями; возвращает указатель на распределенную область памяти или *NULL* при нехватке памяти;

coreleft (void) – получение размера свободной памяти в байтах (тип результата: *unsigned* – для моделей памяти *tiny*, *small* и *medium*; *unsigned long* – для моделей памяти *compact*, *large* и *huge*;

*void free (void *b)* – освобождение блока памяти, адресуемого указателем *b*;

*void *malloc (unsigned n)* – выделение области памяти для размещения блока размером *n* байт; возвращает указатель на распределенную область или *NULL* при нехватке памяти;

Пример создания одномерного динамического массива:

```
float *x;
int  n;
printf("\n Размерность - ");    scanf(" %d",&n);
if ((x = calloc(n, sizeof(*x)))==NULL) {           // Захват памяти
    printf("\n Предел размерности ");
    exit(1); }
else {
    printf("\n Массив создан !");

    ...
    for (i=0; i<n; i++) printf("\n%f",x[i]);

    ...
    free(x);                                     // Освобождение памяти
}
```

В C++ введены две операции: захват памяти – **new** и освобождение захваченной ранее памяти – **delete**.

Общий формат записи:

указатель = **new** тип (значение);

...

delete указатель;

Например:

int *a;

a = new int (8);

В данном случае создана целочисленная динамическая переменная, на которую установлен указатель *a* и которой присвоено начальное значение 8. После работы с ней освобождаем память:

delete a;

Операции **new** и **delete** для массивов:

указатель = new тип [количество];

Результат операции – адрес начала области памяти для размещения данных, указанного количества и типа; при нехватке памяти – *NULL*.

Операция **delete**:

delete [] указатель;

13.1. Пример создания одномерного динамического массива

Массив объявляем указателем.

```
...
double *x;
int i, n;

...
puts(" Введите размер массива: ");
scanf("%d", &n);
x = new double [n] ;
if (x == NULL) {
    puts(" Предел размерности ! ");
    return; }
for (i=0; i<n; i++)           // Ввод элементов массива
    scanf("%lf", &x[i]);

...
delete [ ]x;                  // Освобождение памяти
...
```

13.2. Пример создания двухмерного динамического массива

Напомним, что ID двухмерного массива – указатель на указатель (рис. 4):

```
...
int **m, n1, n2, i, j;
puts(" Введите размеры массива (количество строк и столбцов: ");
scanf("%d%d", &n1, &n2);
m = new int * [n1];           // Захват памяти для указателей «А» (n1=3)
for ( i=0; i<n1; i++)         // Захват памяти для элементов «В» (n2=4)
    m[i] = new int [n2];

...
for ( i=0; i<n1; i++)
    for ( j=0; j<n2; j++)
        m[i] [j] = i+j;      // *(*(m+i)+j) = i+j;

...
for ( i=0; i<n1; i++)         // Освобождение памяти
    delete [ ] m[i];
delete [ ] m;

...
```

14. Строки в языке Си

В языке Си отдельного типа данных «*строки символов*» нет. Работа со строками реализована путем использования одномерных массивов типа *char*, т.е.

строка символов – это одномерный массив типа *char*, заканчивающийся нулевым байтом.

Нулевой байт – это байт, каждый бит которого равен нулю, при этом для нулевого байта определена символьная константа `'\0'` (признак окончания строки, или нуль-терминатор). Поэтому если строка должна содержать *k* символов, то в описании массива необходимо указать *k+1* элемент.

Например, *char a[7];* – означает, что строка может содержать шесть символов, а последний байт отведен под нулевой.

Строковая константа – это набор символов, заключенных в двойные кавычки. Например:

```
char S[]="Работа со строками";
```

В конце строковой константы явно указывать символ `'\0'` не нужно.

При работе со строками удобно пользоваться указателями, например:

```
char *x;
```

```
x = "БГУИР";
```

```
x = (i>0)? "положительное":(i<0)? "отрицательное":"нулевое";
```

Напомним, что для ввода строк обычно используются две стандартные функции:

scanf – вводит значения для строковых переменных спецификатором ввода **%s** до появления первого символа “пробел” (символ «&» перед ID строковых данных указывать не надо);

gets – ввод строки с пробелами внутри этой строки завершается нажатием клавиши ENTER.

Обе функции автоматически ставят в конец строки нулевой байт.

Вывод строк производится функциями *printf()* или *puts()* до первого нулевого байта (`'\0'`):

printf – не переводит курсор после вывода на начало новой строки;

puts – автоматически переводит курсор после вывода строковой информации в начало новой строки.

Например:

```
char Str[30];
```

```
printf(" Введите строку без пробелов : \n");
```

```
scanf("%s",Str);
```

или

```
puts(" Введите строку ");
```

```
gets(Str);
```

Остальные операции над строками выполняются с использованием стандартных библиотечных функций, описание прототипов которых находятся в файле ***string.h***. Рассмотрим наиболее часто используемые функции.

Функция ***int strlen(char *S)*** возвращает длину строки (количество символов в строке), при этом завершающий нулевой байт не учитывается.

Пример:

```
char *S1="Минск!\0", S2[]="БГУИР";
```

```
printf(" %d, %d .", strlen(S1), strlen(S2));
```

Результат выполнения данного участка программы: 6, 5.

Функция *int strcpy(char *S1, char *S2)* – копирует содержимое строки S2 в строку S1.

Функция *strcat(char *S1, char *S2)* – присоединяет строку S2 к строке S1 и помещает ее в массив, где находилась строка S1, при этом строка S2 не изменяется. Нулевой байт, который завершал строку S1, заменяется первым символом строки S2.

Функция *int strcmp(char *S1, char *S2)* сравнивает строки S1 и S2 и возвращает значение <0, если S1<S2; >0, если S1>S2; =0, если строки равны, т.е. содержат одно и то же число одинаковых символов.

Функции преобразования строки S в число:

- целое: *int atoi(char *S)*;
- длинное целое: *long atol(char *S)*;
- действительное: *double atof(char *S)*;

при ошибке данные функции возвращают значение 0.

Функции преобразования числа V в строку S:

- целое: *itoa(int V, char *S, int kod)*;
- длинное целое: *ltoa(long V, char *S, int kod)*; $2 \leq kod \leq 36$, для отрицательных чисел kod=10.

Пример функции *del_c()*, в которой удаляется символ "с" из строки s каждый раз, когда он встречается.

```
void del_c( char s[ ], int c) {  
    int i,j;  
    for( i=j=0; s[i] != '\0'; i++)  
        if( s[i]!=c) s[j++]=s[i];  
    s[j]='\0';  
}
```

14.1. Русификация под Visual

При работе в консольном приложении Visual ввод-вывод выполняется в кодировке ASCII, которая является международной только в первой половине кодов (от 0 до 127, см. прил. 1). Символы национального (русского) алфавита – вторая половина кодов. Для преобразования символов из кодировки ANSI в кодировку ASCII можно использовать функцию *CharToOem* и функцию *OemToChar* – для обратного преобразования (находятся в файле *windows.h*). Приведем пример «русификации» текстов.

```
...  
#include<windows.h>  
    char bufRus[256];  
    char* Rus(const char*);           // Описание прототипа  
void main(void)  
{  
    int a=2;
```

```

float r=5.5;
char s[]="Минск !", s1[256];
printf("\n %s ",Rus(s));
printf("\n Vvedi string ");
gets(s1);
printf("\n %s ",s1);
printf(Rus("\n Значение a = %d r = %f\n"), a, r);
}
char* Rus(const char *text) {      // Функция преобразования символов
CharToOem(text, bufRus);
return bufRus;      }

```

15. Функции пользователя

В языке Си любая подпрограмма – функция, представляющая собой отдельный программный модуль, к которому можно обратиться в любой момент и (в случае необходимости) передавать через параметры некоторые исходные данные и который (в случае необходимости) способен возвращать один или несколько результатов своей работы.

15.1. Декларация функции

Как объект языка Си, функцию необходимо объявить. Объявление функции пользователя, т.е. ее декларация, выполняется в двух формах – в форме описания и форме определения.

Описание функции заключается в приведении в начале программного файла ее прототипа. Прототип функции сообщает компилятору о том, что далее в тексте программы будет приведено ее полное определение (полный ее текст): в текущем или другом файле исходного текста либо находится в библиотеке.

В стандарте языка используется следующий способ декларации функций:

тип_результата ID_функции(тип переменной 1, ..., тип переменной N);

Заметим, что идентификаторы переменных в круглых скобках прототипа указывать необязательно, так как компилятор языка их не обрабатывает.

Описание прототипа дает возможность компилятору проверить соответствие типов и количества параметров при вызове этой функции.

Пример описания функции *fun* со списком параметров:

float fun(int, float, int, int);

Полное определение функции имеет следующий вид:

```

тип_результата ID_функции (список параметров) {
    код функции
    return выражение;
}

```


Тип результата определяет тип *выражения*, значение которого возвращается в точку ее вызова при помощи оператора ***return***.

Если тип функции не указан, то по умолчанию предполагается тип ***int***.

Список параметров состоит из перечня типов и идентификаторов параметров, разделенных запятыми.

Функция может не иметь параметров, но круглые скобки необходимы в любом случае.

Если функция не возвращает значения, она описывается как функция типа ***void*** и в данном случае оператор ***return*** не ставится.

В функции может быть несколько операторов ***return***, но может и не быть ни одного. В таких случаях возврат в вызывающую программу происходит после выполнения последнего в функции оператора.

Пример функции, определяющей наименьшее значение из двух целочисленных переменных:

```
int min (int x, int y) {  
    return (x<y)? x : y;  
}
```

Все функции, возвращающие значение, должны использоваться в правой части выражений языка Си, иначе возвращаемый результат будет утерян.

Если у функции отсутствует список параметров, то при декларации такой функции желательно в круглых скобках также указать ключевое слово ***void***. Например, ***void main(void)***.

В языке Си каждая функция – это отдельный блок программы, вход в который возможен только через вызов данной функции.

Наличие определения функции делает излишним запись ее описания в остатке файла исходного текста.

15.2. Вызов функции

Вызов функции имеет следующий формат:

ID_функции (список аргументов);

где в качестве аргументов можно использовать константы, переменные, выражения (их значения перед вызовом функции будут определены компилятором).

Аргументы списка вызова должны полностью совпадать со списком параметров вызываемой функции по количеству, порядку следования и типам соответствующих им параметров.

Связь между функциями осуществляется через аргументы и возвращаемые функциями значения. Ее можно осуществить также через внешние, глобальные переменные.

Функции могут располагаться в исходном файле в любом порядке. А сама исходная программа может размещаться в нескольких файлах.

В языке Си аргументы при стандартном вызове функции передаются по значению, т.е. в стеке выделяется место для формальных параметров функции и в это выделенное место при ее вызове заносятся значения фактических аргументов. Затем функция использует и может изменять эти значения в стеке.

При выходе из функции измененные значения теряются. Вызванная функция не может изменить значения переменных, указанных как фактические аргументы при обращении к данной функции.

В случае необходимости функцию можно использовать для изменения передаваемых ей аргументов. В этом случае в качестве аргумента необходимо в вызываемую функцию передавать не значение переменной, а ее адрес. А для обращения к значению аргумента-оригинала использовать операцию «*».

Пример функции, в которой меняются местами значения аргументов x и y:

```
void zam (int *x, int *y) {  
    int t;  
    t = *x;  
    *x = *y;  
    *y = t;  
}
```

Участок программы с обращением к данной функции:

```
void zam (int*, int*);  
void main (void) {  
    int a=2, b=3;  
    ...  
    printf(" a = %d , b = %d\n", a, b);  
    zam (&a, &b);  
    printf(" a = %d , b = %d\n", a, b);  
    ...  
}
```

При таком способе передачи аргументов в вызываемую функцию их значения будут изменены, т.е. на экран монитора будет выведено

```
a = 2 , b=3  
a = 3 , b=2
```

15.3. Операция *typedef*

Любому типу данных, как стандартному, так и определенному пользователем, можно задать новое имя с помощью операции

typedef <тип> <новое_имя>;

Введенный таким образом новый тип используется аналогично стандартным типам, например, введя пользовательские типы:

```
typedef unsigned int UINT;  
typedef char M_s[100];
```

декларации идентификаторов введенных типов имеют вид

```
UINT i, j;           →   две переменные типа unsigned int;  
M_s str[10];         →   массив из 10 строк по 100 символов.
```

15.4. Указатели на функции

В языке Си идентификатор функции является константным указателем на начало функции в оперативной памяти и не может быть значением переменной. Но имеется возможность декларировать указатели на функции, с которыми можно обращаться как с переменными (например, можно создать массив, элементами которого будут указатели на функции).

Рассмотрим методику работы с указателями на функции.

1. Как и любой объект языка Си, указатель на функции необходимо декларировать. Формат объявления указателя на функции следующий:

*тип (*переменная-указатель) (список параметров);*

т.е. декларируется указатель, который можно устанавливать на функции, возвращающие результат указанного типа и имеют указанный список параметров. Наличие первых круглых скобок обязательно, так как без них – это декларация функции, которая возвращает указатель на результат своей работы указанного типа и имеет указанный список параметров.

Например, объявление вида *float (* p_f) (char, float);* говорит о том, что декларируется указатель *p_f*, который можно устанавливать на функции, возвращающие вещественный результат и имеющие два параметра: первый – символьного типа, а второй – вещественного типа.

2. Идентификатор функции является константным указателем, поэтому для того чтобы установить переменную-указатель на конкретную функцию, достаточно ей присвоить идентификатор этой функции:

переменная-указатель = ID_функции;

Например, имеется функция с прототипом *float f1(char, float);* тогда операция *p_f = f1;* установит указатель *p_f* на данную функцию.

3. Вызов функции после установки на нее указателя выглядит так:

*(*переменная-указатель)(список аргументов);*

или

переменная-указатель (список аргументов);

После таких действий кроме стандартного обращения к функции

ID_функции(список аргументов);

появляется еще два способа вызова функции:

*(*переменная-указатель)(список аргументов);*

или

переменная-указатель (список аргументов);

Последнее справедливо, так как *p_f* также является адресом начала функции в оперативной памяти.

Для нашего примера к функции *f1* можно обратиться следующими способами:

```
f1('z', 1.5);           // Обращение к функции по ID
(* p_f)('z', 1.5);      // Обращение к функции по указателю
```

`p_f('z', 1.5);` // Обращение к функции по ID указателя

4. Пусть имеется вторая функция с прототипом: *float f2(char, float);* тогда, переустановив указатель `p_f` на эту функцию: `p_f = f2;` имеем опять три способа ее вызова:

`f2('z', 1.5);` // по ID функции
`(* p_f)('z', 1.5);` // по указателю на функцию
`p_f('z', 1.5);` // по ID указателя на функцию

Основное назначение указателей на функции – это обеспечение возможности передачи идентификаторов функций в качестве параметров в функцию, которая реализует некоторый вычислительный процесс, используя формальное имя вызываемой функции.

Пример. Написать функцию вычисления суммы `sum`, обозначив слагаемое формальной функцией `fun(x)`, а при вызове функции суммирования передавать через параметр реальное имя функции, в которой запрограммирован явный вид этого слагаемого. Например, пусть требуется вычислить две суммы:

$$S_1 = \sum_{i=1}^{2n} \frac{x}{5} \quad \text{и} \quad S_2 = \sum_{i=1}^n \frac{x}{2}.$$

Поместим слагаемые этих сумм в пользовательские функции `f1` и `f2`.

При этом для удобства работы воспользуемся операцией *typedef*, введем пользовательский тип данных: указатель на функции, который можно устанавливать на функции, возвращающие результат, указанного типа, и имеющие указанный список параметров:

typedef тип_результата (переменная-указатель)(параметры);*

Тогда в списке параметров функции суммирования достаточно указывать фактические ID функций данного типа.

Программа для решения данной задачи может быть следующей:

```
...
typedef float (*p_f)(float);
float sum(p_f fun, int, float);           // Декларации прототипов функций
float f1(float);
float f2(float);
void main(void) {
    float x, s1, s2;
    int n;
    puts(" Введите кол-во слагаемых n и значение x: ");
    scanf("%d%f", &n, %x);
    s1=sum(f1, 2*n, x);
    s2=sum(f2, n, x);
    printf("\n\t N = %d , X = %f", n, x);
    printf("\n\t Сумма 1 = %f\n\t Сумма 2 = %f", s1, s2);
}
/* Функция вычисления суммы, первый параметр которой – формальное имя
функции, имеющей тип, введенный с помощью операции typedef */
```

```

float sum(p_f fun, int n, float x) {
    float s=0;
    for(int i=1; i<=n; i++) s+=fun(x);
    return s;
}
float f1(float r) {           // Первое слагаемое
    return (r/5.);
}
float f2(float r) {           // Второе слагаемое
    return (r/2.);
}

```

В заключение рассмотрим оптимальную передачу в функции одномерных и двумерных массивов.

Передача в функцию одномерного массива:

```

void main (void) {
    int vect [20];
        ...    fun(vect);        ...
}
void fun( int v [ ])
{    ...    }

```

Передача в функцию двумерного массива:

```

void main(void) {
    int mass [ 2 ][ 3 ]={ {1,2,3}, {4,5,6} };
        ...    fun (mas);        ...
}
void fun( int m [ ][3]) {    ...    }

```

16. Классы памяти и области действия объектов

Напомним, что все объекты перед их использованием должны быть декларированы. Одним из атрибутов в декларации объекта является *класс памяти*, который определяет время существования (время жизни) переменной и область ее видимости (действия).

Имеется три основных места, где объявляются переменные: внутри функции, при определении параметров функции и вне функции. Эти переменные называются соответственно локальными (внутренними) переменными, формальными параметрами и глобальными (внешними) переменными.

Классы памяти объектов в языке Си:

– динамическая память, которая выделяется при вызове функции и освобождается при выходе из нее (атрибуты: *auto* – автоматический; *register* – регистровый);

– статическая память, которая распределяется на этапе трансляции и заполняется по умолчанию нулями (атрибуты: внешний – *extern*, статический – *static*).

16.1. Автоматические переменные

Переменные, декларированные внутри функций, являются внутренними и называются *локальными* переменными. Никакая другая функция не имеет прямого доступа к ним. Такие объекты существуют временно на этапе активности функции.

Каждая локальная переменная существует только в блоке кода, в котором она объявлена, и уничтожается при выходе из него. Эти переменные называют автоматическими и располагаются в стековой области памяти.

По умолчанию локальные объекты, объявленные в коде функции, имеют атрибут класса памяти *auto*.

Принадлежность к этому классу можно подчеркнуть явно, например:

```
void main(void)    {  
    auto int max, lin;  
    ...            }  
}
```

так поступают, если хотят показать, что определение переменной не нужно искать вне функции.

Регистровая память (атрибут *register*) – объекты целого типа и символы рекомендуется разместить не в оперативной памяти, а в регистрах общего назначения (процессора), а при нехватке регистров – в стековой памяти (размер объекта не должен превышать разрядности регистра), для других типов компилятор может использовать другие способы размещения, а может просто проигнорировать данную рекомендацию.

Регистровая память позволяет увеличить быстродействие программы, но к размещаемым в ней объектам в языке Си (но не C++) неприменима операция адресации *&*, а также это неприменимо для массивов, структур, объединений и переменных с плавающей точкой.

16.2. Внешние переменные

Объекты, размещаемые в статической памяти, декларируются с атрибутом *static* и могут иметь любой атрибут области действия. Глобальные объекты всегда являются статическими. Атрибут *static*, использованный при описании глобального объекта, предписывает ограничение его области применимости только в пределах остатка текущего файла. Значения локальных статических объектов сохраняются при повторном вызове функции. Таким образом, в языке Си ключевое слово *static* имеет разный смысл для локальных и глобальных объектов.

Переменные, описанные вне функции, являются внешними переменными. Их еще называют *глобальными* переменными.

Так как внешние переменные доступны всюду, их можно использовать вместо списка аргументов для передачи значений между функциями.

Внешние переменные существуют постоянно. Они сохраняют свои значения и после того, как функции, присвоившие им эти значения, завершат свою работу.

При отсутствии явной инициализации для внешних и статических переменных гарантируется их обнуление, автоматические и регистровые переменные имеют неопределенные начальные значения («мусор»).

Внешняя переменная должна быть определена вне всех функций. При этом ей выделяется фактическое место в памяти. Такая переменная должна быть описана в каждой функции, которая собирается ее использовать. Это можно сделать либо явным описанием *extern*, либо по контексту.

Чтобы функция могла использовать внешнюю переменную, ей нужно сообщить идентификатор этой переменной. Один из способов – включить в функцию описание *extern*.

Описание *extern* может быть опущено, если внешнее определение переменной находится в том же файле, но до ее использования в некоторой конкретной функции.

Включение ключевого слова *extern* позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле.

Во всех файлах, составляющих исходную программу, должно содержаться только одно определение внешней переменной. Другие файлы могут содержать описание *extern* для доступа к ней.

Любая инициализация внешней переменной проводится только в декларации. В декларации должны указываться размеры массивов, а в описании *extern* этого можно не делать.

Например, в файле 1:

```
int sp = 0;
double val [20];
...
```

в файле 2:

```
extern int sp;
extern double val [];
```

...

В Си есть возможность с помощью *#include* иметь во всей программе только одну копию описаний *extern* и вставлять ее в каждый файл во время его компиляции.

Если переменная с таким же идентификатором, как внешняя, декларирована в функции без указания *extern*, то тем самым она становится внутренней в данной функции.

Не стоит злоупотреблять внешними переменными. Такой стиль программирования приводит к программам, связи данных внутри которых не вполне очевидны. Переменные при этом могут изменяться неожиданным образом. Программу становится трудно модифицировать.

Файл 1:

```

int x, y;
char str[ ] = "Rezult = ";

void main(void) {
    ...
}

void fun1(void) {
    y = 15;
    cout << str << y;
}

```

Файл 2:

```

extern int x, y;
extern char str[ ];
    int r, q;
void fun2(void) {
    x = y / 5;
    cout << str << x;
}
void fun3(void) {
    int z= x + y;
    cout << str << z;
}

```

Описания функций подразумевают атрибут *extern* по умолчанию.

16.3. Область действия переменных

В языке Си нет ключевого слова, указывающего область действия объекта. Область действия определяется местоположением декларации объекта в файле исходного текста программы (любой объект полностью описывается в одном операторе).

Напомним общую структуру исходного текста программ на языке Си:

```

<директивы препроцессора>
<описание глобальных объектов>
<заголовок функции>
{
    <описание локальных объектов>
    <список инструкций>
}

```

Файл исходного текста может включать любое количество определений функций и/или глобальных данных.

Параметры функции являются локальными объектами и должны отличаться по идентификаторам от используемых в коде функции глобальных объектов. Локальные объекты, описанные в коде функции, имеют приоритет перед объектами, описанными вне функции, например

```

#include<stdio.h>
    int f1(int);
    int f2(int);
    int f3(int);
        int n;                                // глобальная n

void main (void) {
    int i=2;                                // локальная i=2
    n=3;                                    // глобальная n=3
    i = f1(i);                             // обе переменные i – локальные
}

```



```

    printf(" 1: i=%d , n=%d\n",i,n);    // i=7, n=3
    n = f1(i);                          // n – глобальная, i – локальная
    printf(" 2: i=%d , n=%d\n",i,n);    // i=7, n=12
    i = f2(n);                          // i и n – локальные
    printf(" 3: i=%d , n=%d\n",i,n);    // i=15, n=12
    i = f3(i);                          // обе переменные i – локальные
    printf(" 4: i=%d , n=%d\n",i,n);    // i=29, n=14
}

//
int f1(int i)                          // параметр функции i – локальная
{
    int n = 5;                          // n – локальная
    n+=i;
    return n;
}

//
int f2(int n)                          // параметр функции n – локальная
{
    n+=3;
    return n;
}

//
int f3(int i)
{
    n+=2;                              // n – глобальная
    return i+n;
}

```

Следует учитывать, что любая декларация объекта действует только на остаток файла исходного текста.

В C++ допускается в разных блоках программы использовать один и тот же идентификатор объекта, тогда внутреннее объявление объекта скрывает доступ к объекту, объявленному на более высоком уровне, например:

```

...
void main(void) {
    int i = 3;
    cout << "\n Block 1 - " << i ;    {
        float i = 2.5;
        cout << "\n Block 2 - " << i ;    {
            char i = 'a';
            cout << "\n Block 3 - " << i ;
        }
    }
    cout << "\n New Block 1 - " << ++i ;
}

```

}

В результате выполнения этой программы на экране получим:

Block 1 - 3

Block 2 - 2.5

Block 3 - a

New Block 1 - 4

17. Структуры, объединения, перечисления

17.1. Структуры

Структура – это составной объект языка Си, представляющий собой совокупность логически связанных данных различного типа, объединенных в группу под одним идентификатором. Данные, входящие в эту группу, называют полями.

Термин «структура» в языке Си соответствует двум разным по смыслу понятиям:

- структура – обозначение участка оперативной памяти, где располагаются конкретные значения данных; в дальнейшем – это структурная переменная, поля которой располагаются в смежных областях памяти;

- структура – правила формирования структурной переменной, которыми руководствуется компилятор при выделении ей места в памяти и организации доступа к ее полям.

Определение объектов типа структуры производится за два шага:

- декларация структурного типа данных, не приводящая к выделению участка памяти;
- определение структурных переменных с выделением памяти.

17.2. Декларация структурного типа данных

Структурный тип данных задается в виде шаблона, общий формат описания которого следующий:

```
struct ID структурного типа {  
    описание полей  
};
```

Атрибут *ID структурного типа*, т.е. ее идентификатор, является необязательным и может отсутствовать.

Описание полей производится обычным способом – указываются типы и идентификаторы.

Пример определения структурного типа – необходимо создать шаблон, описывающий информацию о студенте: номер группы, Ф.И.О. и средний балл. Один из возможных вариантов:

```
struct Stud_type {  
    char Number[10];  
    char Fio[40];  
    double S_b;
```

};

Интерпретация объекта типа Stud_type:

Number	Fio	S_b
10	40	8

длина в байтах

Структурный тип данных удобно применять для групповой обработки логически связанных объектов. Параметрами таких операций являются адрес и размер структуры.

Примеры групповых операций:

- захват и освобождение памяти для объекта;
- запись и чтение данных, хранящихся на внешних носителях как физические и/или логические записи с известной структурой (при работе с файлами).

Так как одним из параметров групповой обработки структурных объектов является размер, не рекомендуется декларировать поле структуры указателем на объект переменной размерности, поскольку в данном случае многие операции со структурными данными будут некорректны.

17.3. Создание структурных переменных

Как уже отмечалось, само описание структуры не приводит к выделению под нее места в памяти. Теперь необходимо создать нужное количество переменных с приведенной структурой и сделать это можно двумя способами.

Способ 1. В любом месте программы для декларации структурных переменных, массивов, функций и т.д. используется объявленный в шаблоне структурный тип, например:

```
struct Stud_type student;      – структурная переменная;
Stud_type Stud[100];          – массив структур
Stud_type *p_stud;            – указатель на структуру
Stud_type* Fun(Stud_type);    – прототип функции с параметром структурного типа, возвращающей указатель на объект структурного типа.
```

Способ 2. В шаблоне структуры между закрывающейся фигурной скобкой и символом «;» указывают через запятые идентификаторы структурных данных.

Для нашего примера можно записать:

```
struct Stud_type {
    char Number[10], Fio[40];
    double S_b;
} student, Stud[100], *p_stud;
```

Если дальше в программе не понадобится вводить новые данные объявленного структурного типа, *Stud_type* можно не указывать.

При декларации структурных переменных возможна их одновременная инициализация, например:

```
struct Stud_type {
    char Number[10], Fio[40];
    double S_b;
```

```
} student = {"123456", "Иванов И.И.", 6.53 };
```

или

```
Stud_Type stud1 = {"123456", "Иванов И.И." };
```

Если список инициализации будет короче, то оставшиеся поля структурной переменной будут заполнены нулями.

Некоторые особенности:

1) поля структуры, как правило, имеют разный тип, кроме функций, файлов и самой этой структуры;

2) поля не могут иметь атрибут *класс памяти*, данный атрибут можно определить только для всей структуры;

3) идентификаторы как самой структуры, так и ее полей могут совпадать с ID других объектов программы, т.к. шаблон структуры обладает собственным пространством имен;

4) при наличии в программе функций пользователя шаблон структуры рекомендуется поместить глобально перед определениями всех функций, в этом случае он будет доступен всем функциям.

Элементы структур в общем случае размещаются в памяти последовательно с учетом выравнивания начальных адресов.

Выравнивание – установка значения адреса, кратного некоторой величине, определяемой особенностями адресации данных на аппаратном уровне.

Обращение к полям структур производится при помощи составных имен, которые образуются двумя способами:

а) использованием операции принадлежности (.) в виде

```
ID_структуры . ID_поля
```

или

```
(*указатель_структуры) . ID_поля
```

б) при помощи операции косвенной адресации (->) в виде

```
указатель_структуры -> ID_поля
```

или

```
(&ID_структуры) -> ID_поля
```

Примеры обращения к полям объявленного ранее шаблона:

```
Stud_Type s1, *s2;
```

```
s1. Number,
```

```
s1. Fio,
```

```
s1. S_b;
```

```
s2 -> Number,
```

```
s2 -> Fio,
```

```
s2 -> S_b.
```

17.4. Вложенные структуры

Структуры могут быть вложенными, т.е. поле структуры может быть связующим полем с внутренней структурой, описание которой должно предшествовать по отношению к основному шаблону.

Например, в структуре *person*, содержащей Ф.И.О. и дату рождения, сделать дату рождения внутренней структурой *date* по отношению к структуре *person*. Шаблон такой конструкции будет выглядеть следующим образом:

```
struct date {  
    int day, month, year;  
};  
struct person {
```

```
char fio[40];
struct date fl;
};
```

Объявляем переменную и указатель на переменные такой структуры:

```
struct person a, *p;
```

Инициализируем указатель p адресом переменной a :

```
p = &a;
```

Тогда обращение к полям структурной переменной a будет следующим:

```
а .fio,      а.fl.day,      а.fl.month,      а.fl.year;
или          p->fio,      p->fl.day,      p->fl.month,      p->fl.year.
```

Можно в качестве связи с вложенной структурой использовать указатель на нее:

```
struct date {
    int day, month, year;
};
struct person {
    char fio[40];
    struct date *fl;
};
```

Тогда обращение к полям будет следующим:

```
а .fio,      а.fl->day,      а.fl->month,      а.fl->year;
или          p->fio,      p->fl->day,      p->fl->month,      p->fl->year.
```

Использование *typedef* упрощает определение структурных переменных, например:

```
typedef struct person {
    char fio[40];
    int day, month, year;
} W;
```

здесь W – созданный пользователем тип данных – *структура с указанными полями*, и для нашего примера:

$W\ t1, t2$; – декларация двух переменных типа W .

17.5. Массивы структур

Структурный тип может быть использован для декларации массивов, элементами которых являются структурные переменные, например:

```
struct person spisok[100];      – spisok – массив структур;
или
struct person {
    char fio[40];
    int day, month, year;
} spisok[100];
```

В данном случае обращение к полю, например, day i -й записи может быть выполнено одним из следующих способов:

```
spisok[i].day,          *(spisok+i).day,          (spisok+i)->day.
```

Участок программы, иллюстрирующий передачу структурных данных в функцию:

```
    struct Spisok {
        char Fio[20];
        float S_Bal;
    };
void Out(Spisok );           // Описание прототипов
void Vvod(int, Spisok *);
void main(void) {
    Spisok Stud[50], *sved;
    ...
        for(i=0;i<N;i++)  Vvod(i, &Stud[i]);
        puts("\n Spisok Students");
        for(i=0;i<N;i++)  Out(Stud[i]);
    ...
}

void Out(Spisok dan) {
    printf("\n %20s  %4.2f",dan.Fio, dan.S_Bal);
}

void Vvod(int nom, Spisok *sved) {
    printf("\n Введите сведения %d : ",nom+1);
    fflush(stdin);
    puts("\n ФИО      - ");
    gets(sved->Fio);
    puts("\n Средний балл - ");
    scanf("%f", &sved->S_Bal);
}
```

17.6. Размещение структурных переменных в памяти

При анализе размеров структурных переменных иногда число байт, выделенных компилятором под структурную переменную, оказывается больше, чем сумма байт ее полей. Это связано с тем, что компилятор выделяет участок памяти для структурных переменных с учетом выравнивания, добавляя между полями пустые байты по следующим правилам:

- структурные переменные, являющиеся элементами массива, начинаются на границе слова, т.е. с четного адреса;
- любое поле структурной переменной начинается на границе слова, т.е. с четного адреса и имеет четное смещение по отношению к началу переменной;
- при необходимости в конец переменной добавляется пустой байт, чтобы общее число байт было четное.

17.7. Объединения

Объединение – это поименованная совокупность данных разных типов, размещаемых с учетом выравнивания в одной и той же области памяти, размер которой достаточен для хранения наибольшего элемента.

Объединенный тип данных декларируется подобно структурному:

```
union ID_объединения {  
    описание полей  
};
```

Пример описания объединенного типа:

```
union word {  
    int nom;  
    char str[20];  
};
```

Пример объявления объектов объединенного типа:

```
union word *p_w, mas_w[100];
```

Объединения применяют для экономии памяти в случае, когда объединяемые элементы логически существуют в разные моменты времени либо требуется разнотипная интерпретация поля данных.

Например, пусть поток сообщений по каналу связи содержит сообщения трех видов:

```
struct m1 {  
    char code;  
    float data[100];    };  
struct m2 {  
    char code;  
    int mode;    };  
struct m3 {  
    char code, note[80];    };
```

Элемент *code* – признак вида сообщения. Удобно описать буфер для хранения сообщений в виде

```
struct m123 {  
    char code;  
    union {  
        float data[100];  
        int mode;  
        char note[80];    };  
};
```

Декларация данных типа *union*, создание переменных этого типа и обращение к полям объединений производится аналогично структурам.

Пример использования переменных типа *union*:

```
...  
typedef union q {  
    int a;  
    float b;
```

```

        char s[5];
        } W;

void main(void) {
    W s, *p = &s;
    s.a = 4;
    printf("\n Integer a = %d, Sizeof(s.a) = %d", s.a, sizeof(s.a));
    p -> b = 1.5;
    printf("\n Float b = %f, Sizeof(s.b) = %d", s.b, sizeof(s.b));
    strcpy(p->s, "Minsk");
    printf("\n String a = %s, Sizeof(s.s) = %d", s.s, sizeof(s.s));
    printf("\n Sizeof(s) = %d", sizeof(s));
}

```

Результат работы программы:

Integer a = 4, Sizeof(s.a) = 2

Float b = 1.500000, Sizeof(s.b) = 4

String a = Minsk, Sizeof(s.s) = 5

Sizeof(s) = 5

17.8. Перечисления

Перечисления – это средство создания типа данных посредством задания ограниченного множества значений.

Определение перечислимого типа данных имеет вид

```

enum ID_перечислимого типа {
    список значений };

```

Значения данных перечислимого типа указываются идентификаторами, например:

```

enum marks {
    zero, two, three, four, five
};

```

Транслятор последовательно присваивает идентификаторам списка значений целочисленные величины 0,1,..., . При необходимости можно явно задать значение идентификатора, тогда очередные элементы списка будут получать последующие возрастающие значения. Например:

```

enum level {
    low=100, medium=500, high=1000, limit
};

```

Примеры объявления переменных перечислимого типа:

```

enum marks Est;
enum level state;

```

Переменная типа marks может принимать только значения из множества {zero, two, three, four, five}.

Основные операции с данными перечислимого типа:

– присваивание переменных и констант одного типа;

– сравнение для выявления равенства либо неравенства.

Практическое назначение перечисления – определение множества различающихся символических констант целого типа.

Пример использования переменных перечислимого типа:

```
...
typedef enum {
    mo=1, tu, we, th, fr, sa, su
} days;
void main(void) {
    days w_day;          // Переменная перечислимого типа
    int t_day, end, start; // Текущий день, начало и конец недели
    puts(" Введите день недели (от 1 до 7) : ");
    scanf("%d", &t_day);
    w_day = su;
    start = mo;
    end = w_day - t_day;
    printf("\n Понедельник - %d-й день недели, \
сейчас %d-й день. \n\
До конца недели %d дней (дня). ", start, t_day, end );
}
```

Результат работы программы:

Введите день недели (от 1 до 7) : **2**

Понедельник - 1-й день недели, сейчас 2-й день.

До конца недели 5 дней (дня).

18. Файлы в языке Си

Файл – это набор данных, размещенный на внешнем носителе и рассматриваемый в процессе обработки как единое целое. В файлах размещаются данные, предназначенные для длительного хранения.

Различают два вида файлов: текстовые и бинарные. Текстовые файлы представляют собой последовательность ASCII символов и могут быть просмотрены и отредактированы с помощью любого текстового редактора.

Бинарные (двоичные) файлы представляют собой последовательность данных, структура которых определяется программно.

В языке Си имеется большой набор функций для работы с файлами, большинство которых находятся в библиотеках *stdio.h* и *io.h*.

18.1. Открытие файла

Каждому файлу присваивается внутреннее логическое имя, используемое в дальнейшем при обращении к нему. Логическое имя (идентификатор файла) – это

указатель на файл, т.е. на область памяти, где содержится вся необходимая информация о файле. Формат объявления указателя на файл следующий:

FILE *указатель на файл;

FILE – идентификатор структурного типа, описанный в стандартной библиотеке *stdio.h* и содержащий следующую информацию:

```
type struct {  
    short level;           – число оставшихся в буфере непрочитанных байт;  
                           – обычный размер буфера – 512 байт; как только level=0,  
                           – в буфер из файла читается следующий блок данных;  
    unsigned flags;       – флаг статуса файла – чтение, запись, дополнение;  
    char fd;              – дескриптор файла, т.е. число, определяющее его номер;  
    unsigned char hold;   – переданный символ, т.е. ungetc-символ;  
    short bsize;          – размер внутреннего промежуточного буфера;  
    unsigned char buffer; – значение указателя для доступа внутри буфера, т.е.  
                           – задает начало буфера, начало строки или текущее значение  
                           – указателя внутри буфера в зависимости от режима  
                           – буферизации;  
    unsigned char *curp;  – текущее значение указателя для доступа внутри буфера,  
                           – т.е. задает текущую позицию в буфере для обмена с программой;  
    unsigned istemp;      – флаг временного файла;  
    short token;          – флаг при работе с файлом;  
} FILE;
```

Прежде чем начать работать с файлом, т.е. получить возможность чтения или записи информации в файл, его нужно открыть для доступа. Для этого обычно используется функция

FILE* *fopen*(char * имя_файла, char *режим);

она берет внешнее представление – физическое имя файла на носителе (дискета, винчестер) и ставит ему в соответствие логическое имя.

Физическое имя, т.е. имя файла и путь к нему задается первым параметром – строкой, например, “a:Mas_dat.dat” – файл с именем Mas_dat.dat, находящийся на дискете, “d:\\work\\Sved.txt” – файл с именем Sved.txt, находящийся на винчестере в каталоге work.

Внимание! Обратный слеш (\), как специальный символ, в строке записывается дважды.

При успешном открытии функция *fopen* возвращает указатель на файл (в дальнейшем – указатель файла). При ошибке возвращается **NULL**. Данная ситуация обычно возникает, когда неверно указывается путь к открываемому файлу. Например, если в дисплейном классе нашего университета указать путь, запрещенный для записи (обычно разрешенным является d:\\work\\).

Второй параметр – строка, в которой задается режим доступа к файлу:

w – файл открывается для записи; если файла с заданным именем нет, то он будет создан; если такой файл существует, то перед открытием прежняя информация уничтожается;

r – файл открывается только для чтения; если такого файла нет, то возникает ошибка;

a – файл открывается для добавления в конец новой информации;

r+ – файл открывается для редактирования данных – возможны и запись, и чтение информации;

w+ – то же, что и для **r+**;

a+ – то же, что и для **a**, только запись можно выполнять в любое место файла; доступно и чтение файла;

t – файл открывается в текстовом режиме;

b – файл открывается в двоичном режиме.

Текстовый режим отличается от двоичного тем, что при открытии файла как текстового пара символов «перевод строки», «возврат каретки» заменяется на один символ: «перевод строки» для всех функций записи данных в файл, а для всех функций вывода символ «перевод строки» теперь заменяется на два символа: «перевод строки», «возврат каретки».

По умолчанию файл открывается в текстовом режиме.

Пример: `FILE *f;` – объявляется указатель на файл `f`;

`f = fopen ("d:\\work\\Dat_sp.cpp", "w");` – открывается для записи файл с логическим именем `f`, имеющим физическое имя `Dat_sp.cpp`, находящийся на диске `d`, в каталоге `work`; или более кратко

`FILE *f = fopen ("d:\\work\\Dat_sp.cpp", "w");`

18.2. Закрывание файла

После работы с файлом доступ к нему необходимо закрыть. Это выполняет функция `int fclose(указатель_файла)`. Например, из предыдущего примера файл закрывается так: `fclose (f)`;

Для закрытия нескольких файлов введена функция, объявленная следующим образом: `void fcloseall(void)`;

Если требуется изменить режим доступа к файлу, то для этого сначала необходимо закрыть данный файл, а затем вновь его открыть, но с другими правами доступа. Для этого используют стандартную функцию:

`FILE* freopen (char* имя_файла, char *режим, FILE *указатель_файла);`

Эта функция сначала закрывает файл, объявленный `указателем_файла` (как это делает функция `fopen`), а затем открывает файл с `именем_файла` и правами доступа `«режим»`.

В языке Си имеется возможность работы с временными файлами, которые нужны только в процессе работы программы. В этом случае используется функция

`FILE* tmpfile (void);`

которая создает на диске временный файл с правами доступа «w+b», после завершения работы программы или после закрытия временного файла он автоматически удаляется.

18.3. Запись – чтение информации

Все действия по чтению-записи данных в файл можно разделить на три группы: операции посимвольного ввода-вывода; операции построчного ввода-вывода; операции ввода-вывода по блокам.

Рассмотрим основные функции, применяемые в каждой из указанных трех групп операций.

Посимвольный ввод-вывод

В функциях посимвольного ввода-вывода происходит прием одного символа из файла или передача одного символа в файл:

int fgetc(FILE *f) – считывает и возвращает символ из файла f;
int fputc(int ch, FILE *f) – записывает в файл f код ch символа.

Построчный ввод-вывод

В функциях построчного ввода-вывода происходит перенос из файла или в файл строк символов:

int fgets (char *S, int m, FILE *f) – чтение из файла f в строку S m байт;
int fputs (char *S, FILE *f) – запись в файл f строки S до тех пор, пока не встретится '\0', который в файл не переносится и на символ '\n' не заменяется.

Блоковый ввод-вывод

В функциях блочного ввода-вывода работа происходит с целыми блоками информации:

int fread (void *p, int size, int n, FILE *f) – считывает n блоков по size байт каждый из файла f в область памяти с указателем p (необходимо заранее отвести память под считываемый блок);
int fwrite (void *p, int size, int n, FILE *f) – записывает n блоков по size байт каждый из области памяти с указателем p в файл f.

Форматированный ввод-вывод производится функциями:

int fscanf (FILE *f, char *формат, список адресов объектов) – считывает из файла f информацию для объектов в соответствии с указанными форматами;
int fprintf (FILE *f, char *формат, список объектов) – записывает в файл f объекты, указанные в списке в соответствии с форматами.

Данные функции аналогичны функциям *scanf* и *printf*, рассмотренным раньше, только добавлен параметр – *указатель на файл*.

18.4. Текстовые файлы

Для работы с текстовыми файлами удобнее всего пользоваться функциями *fprintf*, *fscanf*, *fgets* и *fputs*.

Создание текстовых результирующих файлов обычно необходимо для оформления отчетов по лабораторным и курсовым работам.

Пример создания текстового файла:

```
#include<stdio.h>

void main(void) {
    FILE *f1;
    int a=2, b=3;
    If(!(f1=fopen("d:\\work\\f_rez.txt","w+t"))) {
        puts("Файл не создан!");
        return;    }
    fprintf(f1," Файл результатов \n");
    fprintf(f1," %d плюс %d = %d\n",a,b,a+b);
    fclose(f1);
}
```

Просмотрев содержимое файла, можно убедиться, что данные в нем располагаются точно так, как на экране при использовании функции *printf*.

18.5. Бинарные файлы

Бинарные (двоичные) файлы обычно используются для организации баз данных, состоящих, как правило, из объектов структурного типа. При чтении-записи бинарных файлов удобнее всего пользоваться функциями, выполняемыми блоковый ввод-вывод *fread* и *fwrite*.

Рассмотрим наиболее распространенные функции, с помощью которых можно организовать работу с файлами:

1) *int fileno*(FILE *f) – возвращает значение дескриптора файла *f* – *fd* (число, определяющее номер файла);

2) *long filelength*(int fd) – возвращает длину файла, имеющего номер (дескриптор) *fd* в байтах;

3) *int chsize*(int fd, long pos) – выполняет изменение размера файла, имеющего номер *fd*, признак конца файла устанавливается после байта с номером *pos*;

4) *int fseek*(FILE *f, long size, int kod) – выполняет смещение указателя файла *f* на *size* байт в направлении признака *kod*: 0 – от начала файла; 1 – от текущей позиции указателя; 2 – от конца файла;

5) *long ftell*(FILE *f) – возвращает значение указателя на текущую позицию файла (-1 – ошибка);

6) *int feof*(FILE *f) – возвращает ненулевое значение при правильной записи признака конца файла;

7) *int fgetpos*(FILE *f, long *pos) – определяет значение текущей позиции *pos* файла *f*, возвращает 0 при успешном завершении.

Пример программы работы с файлом структур:

```
...
struct Sved {
    char Fam[30];
    float S_Bal;
} zap,zapt;
char Spis[]="c:\\bc31\\work\\Sp.dat";
FILE *F_zap;
FILE* Open_file(char *, char *);
void main (void) {
    int i, j, size = sizeof(Sved);
    char kodR;
    while(1) {
        puts("Создание - 1\nПросмотр - 2\nДобавление - 3\nВыход - 0");
        switch(kodR = getch()) {
            case '1': case '3':
                if(kodR=='1') F_zap = Open_file (Spis,"w+");
                else F_zap = Open_file (Spis,"a+");
                while(2) {
                    cout << "\n Fam "; cin >> zap.Fam;
                    if((zap.Fam[0])=='0') break;
                    cout << "\n Средний балл: ";
                    cin >> zap.S_Bal;
                    fwrite(&zap,1,size,F_zap);
                }
                fclose(F_zap);
                break;
            case '2': F_zap = Open_file (Spis,"r+"); int nom=1;
                while(2) {
                    if(!fread(&zap,size, 1, F_zap)) break;
                    printf(" %2d: %20s %5.2f\n", nom++, zap.Fam, zap.S_Bal);
                }
                fclose(F_zap);
                break;
            case '0': return; // exit(0);
        } // Конец Switch
    } // Конец While(1)
} // Конец программы

FILE* Open_file(char *file, char *kod) {
    FILE *f;
```

```
if(!(f = fopen(file, kod))) {  
    puts("Файл не создан!");  
    getch();  
    exit(1);  
}  
else return f;  
}
```

Литература

1. Бусько В.Л., Корбит А.Г. и др. Программирование: Лаб. практикум для студ. 1-2-го курсов всех спец. БГУИР всех форм обучения. Ч.2. – Мн.: БГУИР, 2003.
2. Березин Б.И., Березин С.Б. Начальный курс С и С++. – М.: Диалог-МРТИ, 1999.
3. Демидович Е.М. Основы алгоритмизации и программирования. Язык Си. – Мн.: Бестпринт, 2001.
4. Касаткин А.И., Вольвачев А.Н. Профессиональное программирование на языке Си: От Turbo-C к Borland C++: Справ.пособие. – Мн.: Выш. шк., 1992.
5. Касаткин А.Н. Профессиональное программирование на языке Си. Управление ресурсами: Справ.пособие. – Мн.: Выш. шк. 1992
6. Керниган Б., Ритчи Д. Язык программирования Си. – М.: Финансы и статистика, 1992. – 271 с.
7. Климова Л.И. С++. Практическое программирование. – М.: Кудиц-Образ, 2001. – 587 с.
8. Павловская Т.А. С/С++. Программирование на языке высокого уровня. – СПб.: Питер, 2004. – 641 с.
9. Петзольд Ч. Программирование для Windows 95. – BHV.: Санкт-Петербург, 1997.
10. Подбельский В.В., Фомин С.С. Программирование на языке Си. – М.: Финансы и статистика, 2001.
11. Страуструп Б. Язык программирования С++. 2-е изд.: В 2 т. Киев: ДИА-Софт, 1993.
12. Тимофеев В.В. Программирование в среде С++ Builder 5. – М.: БИНОМ, 2000.
13. Шилд Г. Программирование на Borland C++. – Мн.: ПОПУРРИ, 1999.
14. Юлин В.А., Булатова И.Р. Приглашение к Си. – Мн.: Выш.шк., 1990.
15. Сеницын А.К. Конспект лекций по курсу «Программирование» для студентов 1-2-го курсов радиотехнических специальностей. – Мн.: БГУИР, 2001.

Приложение 1. Таблицы символов ASCII

Стандартная часть таблицы символов ASCII

КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С
0		16	►	32		48	0	64	@	80	P	96	`	112	p
1	☺	17	◄	33	!	49	1	65	A	81	Q	97	a	113	q
2	☹	18	↕	34	"	50	2	66	B	82	R	98	b	114	r
3	♥	19	!!	35	#	51	3	67	C	83	S	99	c	115	s
4	♦	20	¶	36	\$	52	4	68	D	84	T	100	d	116	t
5	♣	21	§	37	%	53	5	69	E	85	U	101	e	117	u
6	♠	22	—	38	&	54	6	70	F	86	V	102	f	118	v
7	•	23	↕	39	'	55	7	71	G	87	W	103	g	119	w
8	■	24	↑	40	(56	8	72	H	88	X	104	h	120	x
9	○	25	↓	41)	57	9	73	I	89	Y	105	i	121	y
10	◼	26	→	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	27	←	43	+	59	;	75	K	91	[107	k	123	{
12	♀	28	└	44	,	60	<	76	L	92	\	108	l	124	
13	♪	29	↔	45	-	61	=	77	M	93]	109	m	125	}
14	🎵	30	▲	46	.	62	>	78	N	94	^	110	n	126	~
15	☼	31	▼	47	/	63	?	79	O	95	_	111	o	127	△

Некоторые из вышеперечисленных символов имеют особый смысл. Так, например, символ с кодом 9 обозначает символ горизонтальной табуляции, символ с кодом 10 – символ перевода строки, символ с кодом 13 – символ возврата каретки.

Дополнительная часть таблицы символов

КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С	КС	С
128	A	144	P	160	a	176	░	192	L	208	Ш	224	p	240	Ё
129	Б	145	С	161	б	177	▒	193	┐	209	▄	225	с	241	ё
130	В	146	Т	162	в	178	▓	194	└	210	▌	226	т	242	Є
131	Г	147	У	163	г	179	█	195	┌	211	▐	227	у	243	є
132	Д	148	Ф	164	д	180	░	196	┐	212	▄	228	ф	244	Ї
133	Е	149	Х	165	е	181	▒	197	└	213	▌	229	х	245	ï
134	Ж	150	Ц	166	ж	182	▓	198	┌	214	▐	230	ц	246	ÿ
135	З	151	Ч	167	з	183	█	199	┐	215	▄	231	ч	247	ÿ
136	И	152	Ш	168	и	184	░	200	└	216	▌	232	ш	248	◦
137	Й	153	Щ	169	й	185	▒	201	┌	217	▐	233	щ	249	·
138	К	154	Ъ	170	к	186	▓	202	┐	218	▄	234	ъ	250	·
139	Л	155	Ы	171	л	187	█	203	└	219	▌	235	ы	251	√
140	М	156	Ь	172	м	188	░	204	┐	220	▄	236	ь	252	№
141	Н	157	Э	173	н	189	▒	205	└	221	▌	237	э	253	α
142	О	158	Ю	174	о	190	▓	206	┌	222	▐	238	ю	254	■
143	П	159	Я	175	п	191	█	207	┐	223	▄	239	я	255	

В таблицах обозначение КС означает "код символа", а С – "символ".

Приложение 2. Операции языка Си

Операции приведены в порядке убывания приоритета, операции с разными приоритетами разделены чертой.

Опера- ция	Краткое описание	Использование	Порядок выполне- ния
Унарные операции			
. -> [] ()	Доступ к члену Доступ по указателю Индексирование Вызов функции	объект . член указатель -> член переменная [выражение] ID(список)	Слева направо
++ -- sizeof ++ -- ~ ! - (+) * & ()	Постфиксный инкремент Постфиксный декремент Размер объекта (типа) Префиксный инкремент Префиксный декремент Побитовое НЕ Логическое НЕ Унарный минус (плюс) Раскрытие указателя Адрес Приведение типа	lvalue++ lvalue-- sizeof(ID или тип) ++lvalue --lvalue ~выражение !выражение - (+)выражение *выражение &выражение (тип)выражение	Справа налево
Бинарные и тернарная операции			
* / % + - << >> < <= > >= == != & ^ &&	Умножение Деление Получение остатка Сложение Вычитание Сдвиг влево Сдвиг вправо Меньше Меньше или равно Больше Больше или равно Равно Не равно Побитовое И Побитовое искл. ИЛИ Побитовое ИЛИ Логическое И	выражение * выражение выражение / выражение выражение % выражение выражение + выражение выражение – выражение выражение << выражение выражение >> выражение выражение < выражение выражение <= выражение выражение > выражение выражение >= выражение выражение == выражение выражение != выражение выражение & выражение выражение ^ выражение выражение выражение выражение && выражение	Слева направо

Опера- ция	Краткое описание	Использование	Порядок выполне- ния
	Логическое ИЛИ	<i>выражение выражение</i>	Слева на- право
?:	Условная операция (тернар- ная)	<i>выражение ? выражение : выражение</i>	
=	Присваивание	<i>lvalue = выражение</i>	Справа налево
*=	Умножение с присваива- нием	<i>lvalue *= выражение</i>	
/=	Деление с присваиванием	<i>lvalue /= выражение</i>	
%=	Остаток от деления с при- сваиванием	<i>lvalue %= выражение</i>	
+=	Сложение с присваивани- ем	<i>lvalue += выражение</i>	
- =	Вычитание с присваивани- ем	<i>lvalue -= выражение</i>	
<<=	Сдвиг влево с присваива- нием	<i>lvalue <<= выражение</i>	
>>=	Сдвиг вправо с присваива- нием	<i>lvalue >>= выражение</i>	
&=	Поразрядное И с присваи- ванием	<i>lvalue &= выражение</i>	
=	Поразрядное ИЛИ с при- сваиванием	<i>lvalue = выражение</i>	
^=	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием	<i>lvalue ^= выражение</i>	
,	Последовательное вычис- ление	<i>выражение, выражение</i>	Слева направо

Приложение 3. Возможности препроцессора

Препроцессор, как мы уже знаем, это программа предварительной обработки исходного текста программы перед этапом компиляции. Чаще всего препроцессор автоматически вызывается на этапе компиляции, если в исходном тексте обнаружена хотя бы одна его директива.

Признаком директивы препроцессора является символ `#`. При необходимости продолжения директивы в следующей строке текущую строку должен завершать символ `\`.

Возможности препроцессора языка Си:

- лексемное замещение идентификаторов;
- макрозамещение;
- включение файлов исходного текста;
- условная компиляция;
- изменение нумерации строк и текущего имени файла.

Директивы лексемного замещения идентификаторов

Директива определения значения идентификатора (ID):

```
#define ID строка
```

В результате каждое вхождение в исходный текст элемента ID заменяется на значение элемента *строка*:

```
#define L_bufs 2048
#define binary int
#define WAIT fflush(stdin); getch()
#define BEEP sound(800);\
                delay(100);\
                nosound()
```

Лексемное замещение весьма удобно для сокращения записи повторяющихся фрагментов теста и определения символических констант:

```
#define YES 1
#define NO 2
#define ESC 27
#define Enter 30
```

которые могут быть в дальнейшем использованы:

```
if (x==ESC) break;
BEEP;
return(YES);
```

Директива отмены

```
#undef ID
```

Далее по исходному тексту можно назначить новое значение такого идентификатора.

Макрозамещение

Макрозамещение – обобщение лексемного замещения посредством параметризации строки директивы `define` в виде:

```
#define ID(параметр1,... ) строка
```

между элементом ID и открывающей скобкой пробелы не допускаются.

Такой вариант директивы `define` иногда называют макроопределением. Элемент *строка* обычно содержит параметры, которые будут заменены препроцессором фактическими аргументами так называемой макрокоманды, записываемой в формате

```
ID(аргумент1,... )
```

Пример макроопределения и макрокоманд:

```
#define P(X) printf("\n%s",X)
```

```
...
```

```
char *x;
```

```
P(x); // Использование макроопределения P(X)
```

```
P(" НАЧАЛО ОПТИМИЗАЦИИ");
```

```
printf("\n%s",x); // Эквивалентные операторы
```

```
printf("\n%s"," НАЧАЛО ОПТИМИЗАЦИИ");
```

В строке макроопределений идентификаторы параметров сложных выражений рекомендуется заключать в круглые скобки:

```
#define MAX(A,B) ((A)>(B)? (A):(B))
```

```
#define ABS(X) ((X)<0? -(X):(X))
```

Потребность в круглых скобках возникает при опасности искажения смысла вложенных выражений из-за действия правил приоритета операций. Пример искажения смысла операций:

```
#define BP(X) X*X
```

```
...
```

```
int x,y,z;
```

```
x=BP(y+z); ↔ x=y+z*y+z; ↔ x=y+(z*y)+z;
```

Очевидно, что ошибки будут и при следующих вариантах:

```
#define BP(X) (X*X)
```

```
#define BP(X) (X)*(X)
```

Безопасный вариант:

```
#define BP(X) ((X)*(X))
```

Иногда источником ошибок может быть символ «точка с запятой» в конце строки макроопределения:

```
#define BP(X) ((X)*(X));
```

```
...
```

```
int x,y,z;
```

```
x=BP(z)-BP(y); ↔ y=((z)*(z)); -((y)*(y));
```

Макроопределение отменяется директивой ***undef***.

Идентификаторы макроопределений обычно составляют из прописных букв латинского алфавита. Это позволяет отличать макрокоманды от вызова функций.

Макрокоманда внешне синтаксически эквивалентна операции вызова функции, но смысл их различен. Функция в программе имеется в одном экземпляре, но на ее вызов тратится время для подготовки параметров и передачи управления. Каждая макрокоманда замещается соответствующей частью макроопределения, но потерь на передачу управления нет.

Подключение файлов исходного текста

Напомним, что имеются два варианта запроса включения в текущий файл содержимого другого файла. Директива

```
#include < ID_файла >
```

вводит содержимое файла из стандартного каталога (обычно – \include\), а директива

```
#include " ID_файла "
```

организует последовательный поиск в текущем, системном и стандартном каталогах. Например:

```
#include <alloc.h>           // Средства распределения памяти
```

```
#include <dos.h>             // Обращения к функциям ОС
```

```
#include "a:\prg\head.h"    // Включение файла пользователя
```

Рекомендуется описания системных объектов включать из стандартных каталогов и размещать их в начале файла исходного текста программы. Системные объекты в результате получают атрибут области действия «глобальный», что устраняет неоднозначность их описания.

Условная компиляция

Директивы условной компиляции и реализуемые правила включения исходного текста:

а) условное включение (аналог работы оператора if):

```
#if<предикат_условия>
```

```
    ТЕКСТ_1
```

```
#endif
```

б) альтернативное включение (аналог if-else):

```
#if<предикат_условия>
```

```
    ТЕКСТ_1
```

```
#else
```

```
    ТЕКСТ_2
```

```
#endif
```

Виды предикатов условий:

константное_выражение → истина, если его значение ≠ 0;

def ID → истина, если ID был определен ранее оператором #define;

ndef ID → истина, если ID не был определен оператором #define.

Константное_выражение отделяется от ключевого слова *if* разделителем, а *def* и *ndef* – нет.

Пример:

```
#ifdef DEBUG
    print_state();
#endif
```

Элементы исходного текста "ТЕКСТ_1" или "ТЕКСТ_2" могут содержать любые директивы препроцессора.

Примеры:

```
#ifndef EOF
#define EOF -1
#endif
#if UNIT==CON
#include "conproc.c"
#else
#include "outproc.c"
#endif
```

Изменение нумерации строк и идентификатора файла

По умолчанию диагностические сообщения компилятора привязываются к номеру строки и ID файла исходного текста.

Директива

#line номер_строки ID_файла

позволяет с целью более приметной привязки к фрагментам текста изменить номер текущей строки и ID файла на новые значения («ID_файла» можно опустить).

Приложение 4. Интегрированная среда программирования Visual C++

Интегрированная среда разработки (Integrated Development Environment, или, сокращенно, IDE) – это программный продукт, объединяющий текстовый редактор, компилятор, отладчик и справочную систему.

Любая программа в среде Visual C++ всегда создается в виде отдельного проекта. Проект (project) – это набор взаимосвязанных исходных файлов и, возможно, включаемых заголовочных файлов, компиляция и компоновка которых позволяет создать исполняемую программу. Основу Visual C++ составляет рабочая область (project workspace). Она может содержать любое количество различных проектов, сгруппированных вместе для согласованной разработки: от отдельного приложения до библиотеки функций или целого программного пакета. Решение же простых (учебных) задач сводится к оформлению каждой программы в виде одного проекта, т.е. рабочая область проекта будет содержать ровно один проект.

1. Вид рабочего стола консольного приложения Visual C++

После запуска Visual C++ появляется главное окно программы, вид которого приведен на рис. 4.1. (В зависимости от настроек Visual C++ его вид может несколько иным.)

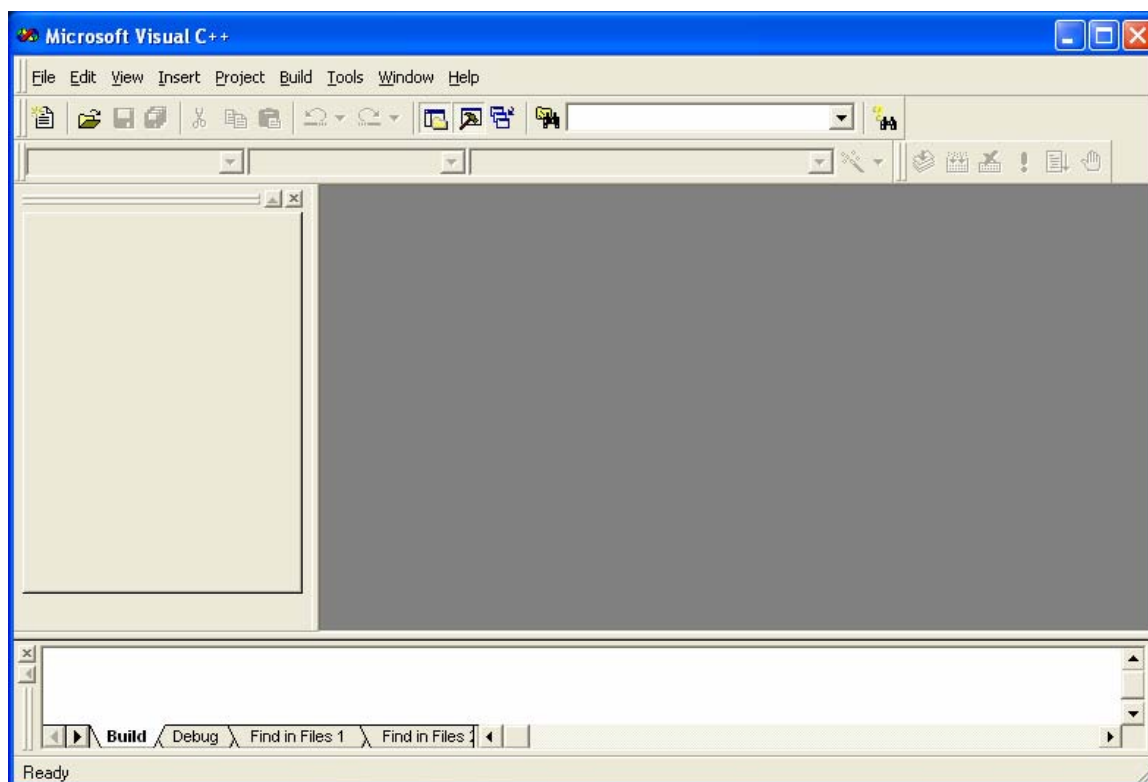


Рис. 4.1

Экран Visual C++ разделен на четыре основные зоны.

Сверху расположены меню и панели инструментов. Кроме них рабочий стол Visual C++ включает в себя три окна:

Окно **Project Workspace** (окно рабочей области) – расположено в левой части. Первоначально окно закрыто, но после создания нового проекта или загрузки существующего проекта это окна будет содержать несколько вкладок.

Справа расположено окно **Editor** (окно редактирования). Его используют для ввода и проверки и редактирования исходного кода программы.

Окно **Output** (окно вывода) служит для вывода сообщений о ходе компиляции, сборки и выполнения программы и сообщений о возникающих ошибках.

Для кнопок панелей инструментов предусмотрена удобная контекстная помощь: если навести курсор мыши на кнопку и задержать на пару секунд, то всплывет подсказка с назначением данной кнопки.

Developer Studio позволяет создавать проекты различных типов, которые ориентированы на различные сферы применения. Большинство типов проектов являются оконными Windows-приложениями с соответствующим графическим интерфейсом. Но также предусмотрена работа и консольными приложениями. При запуске консольного приложения операционная система создает консольное окно, через которое идет весь ввод-вывод данных программы. Такая работа и представляет имитацию работы в операционной системе MS DOS или других операционных системах в режиме командной строки. Этот тип приложений больше всего подходит для целей изучения языка C/C++ так как не требует создания Windows-кода для пользовательского интерфейса. Рассмотрим приемы работы с консольными приложениями более подробно. (Рекомендуется для размещения проектов создать специальную рабочую папку.)

2. Создание нового проекта

Для создания нового проекта типа «консольное приложение» выполните следующие действия:

1. Выберите в строке меню главного окна команду **File/New...**
2. В открывшемся диалоговом окне New выберите вкладку **Projects**:
 - выберите тип проекта: **Win32 Console Application**;
 - введите имя проекта в текстовом поле **Project Name**, например **lr1**;
 - в текстовом поле **Location** введите имя каталога (полный путь к нему) для размещения будущих файлов проекта (если указанный вами каталог отсутствует, то он будет создан автоматически); Путь к будущему проекту можно выбрать щелкнув на кнопке, расположенной справа от текстового поля Location;
 - щелкните левой кнопкой мыши на кнопке **OK**.
3. Щелчок запустит встроенный мастер приложений: Application Wizard, который откроет диалоговое окно **Win32 Console Application – Step 1 of 1** с предложением определиться, какой подтип консольного приложения желаете создать:
 - выберите тип: **An empty project** (пустой проект);
 - щелкните на кнопке **Finish**.

4. После щелчка появится окно: **New Project Information** (информация о новом проекте) со спецификациями проекта и информацией о каталоге, в котором будет размещен создаваемый проект:

5. Щелкните на кнопке **OK**.

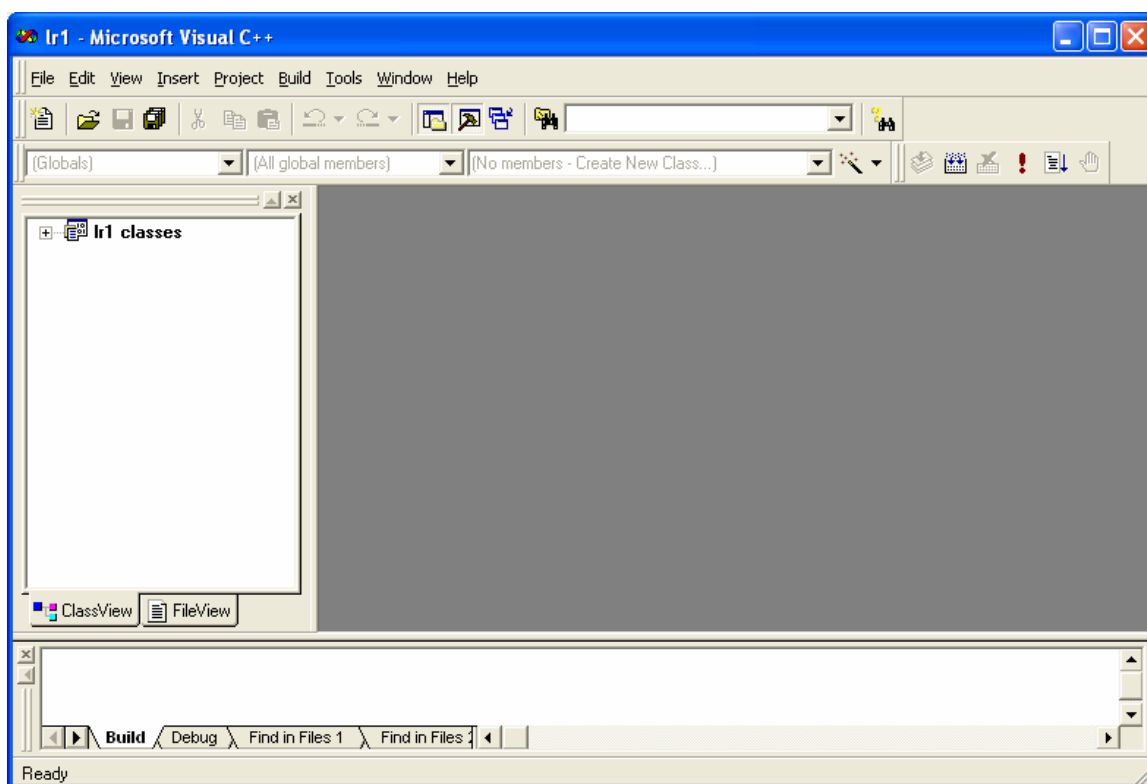


Рис. 4.2

После выполненных шагов рабочий стол примет вид, показанный на рис. 4.2, а в папке **lr1**, созданной мастером приложений – файлы **First.dsw**, **First.dsp**, **First.opt**, **First.ncb** и папка **Debug** (или **Release** – в зависимости от конфигурации проекта).

Краткое описание каждого из файлов:

First.dsw – файл рабочей области проекта, используемый внутри интегрированной среды разработки. Он объединяет всю информацию о проектах, входящих в данную рабочую область.

First.dsp – проектный файл, используемый для построения (building) отдельного проекта или подпроекта).

First.opt – файл, содержащий опции рабочей области проекта. Благодаря этому файлу при каждом открытии рабочей области проекта все параметры Developer Studio, выбранные во время последнего сеанса работы с данной рабочей областью, будут восстановлены.

First.ncb – служебный файл. Он создается компилятором и содержит информацию, которая используется в инструменте интегрированной среды под названием ClassView. Панель ClassView находится в окне Project Workspace и показывает все классы C++, для которых доступны определения в рамках данного

проекта, а также все элементы этих классов. В задачах данного практикума данная панель будет пустой, так как классы C++ них не используются.

Debug – папка, в которую будут помещаться файлы, формируемые компилятором и сборщиком. Из этих файлов нас будет интересовать, в общем-то, только один – исполняемый файл, имеющий расширение *.exe.

Как видно из рис. 4.2 окно Project Workspace теперь открыто. В нем отображены две вкладки: **Class View** и **File View**. Вкладка Class View как легко убедиться пустая. Щелчком мыши переключаемся на вкладку File View. Она предназначена для просмотра списка файлов проекта. Щелкнув мышью на значке «+» откроем список First files,. Появится дерево списка файлов, содержащее пиктограммы трех папок: **Source Files** (исходные коды), **Header Files** (заголовочные файлы), **Resource Files** (файлы ресурсов). Так как в консольных приложениях файлы ресурсов не используются, последняя папка всегда будет пустой. Первоначально и первые две папки пустые, т.к. в качестве подтипа консольного приложения был выбран пустой проект (опция An empty project).

3. Добавление к проекту файлов с исходным кодом

При создании консольного приложения можно или добавить уже существующий файл с исходным кодом (lr1.cpp, т.е. чтобы исключить путаницу, желательно, чтобы имя файла с исходным кодом совпадало с именем проекта), который был создан при помощи других оболочек, или создать новый файл в встроенном текстовом редакторе среды программирования Visual C++.

Добавление существующего файла

1. Скопируйте исходный файл (lr1.cpp) в папку рабочей области проекта (в данном случае – lr1).

2. Вернитесь к списку lr1 files в окне Project Workspace проекта и щелкните правой кнопкой мыши на папке Source Files.

3. В появившемся контекстном меню щелчком мыши выберите команду добавления файлов **Add Files to Folder....**

4. В открывшемся диалоговом окне **Insert Files...** выберите нужный файл (lr1.cpp) и щелкните на кнопке ОК.

Создание и добавление нового файла

В этом случае необходимо выполнить следующие действия:

5. Выберите в строке меню главного окна команду **File/New....** В результате откроется диалоговое окно **New**.

6. На вкладке **Files**:

– выберите тип файла (в данном случае: **C++ Source File**);

– в текстовом поле **File Name** введите нужное имя файла (в данном случае: lr1.cpp); флажок **Add to project** должен быть включен;

– щелкните на кнопке ОК.

После этого получим следующие результаты:

1) в окне Project Workspace раскроется папка Source Files списка файлов проекта и в нее будет помещен файл lr1.cpp;

2) окно редактора Editor станет белым, а в его левом верхнем углу будет мерцать черный текстовый курсор, предлагая ввести какой-нибудь текст.

Введем, например, такой текст программы:

```
#include <stdio.h>
void main(void)
{
    printf("\n Hello World! \n");
}
```

В случае необходимости переключитесь на вкладку File View. Открываем список rl1 files, папку Source Files и убеждаемся, что в проекте создан файл rl1.cpp с только что набранным кодом.

Рисунок 4.3 иллюстрирует вид главного окна Visual C++ после проделанной работы:

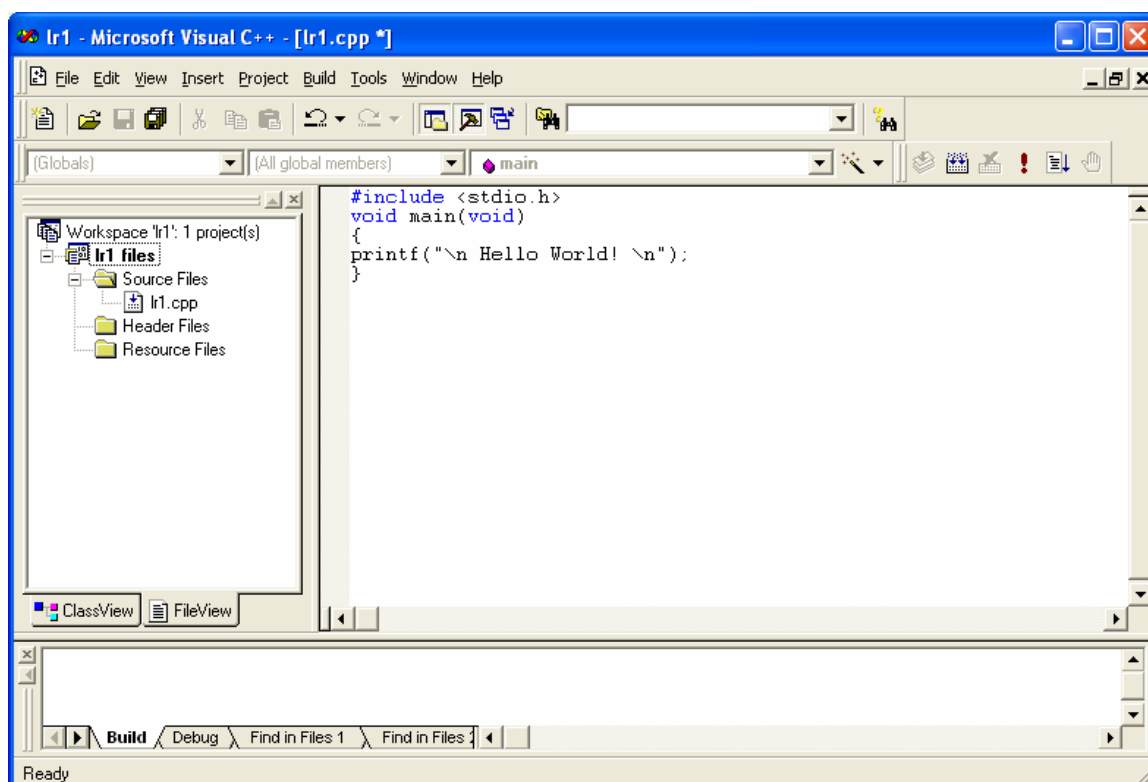


Рис. 4.3

4. Компиляция, компоновка и выполнение проекта

Эти операции можно выполнить или через подменю Build главного окна, или при помощи кнопок панели инструментов или при помощи комбинации горячих клавиш.

Данное подменю объединяет команды для компиляции, сборки и отладки программ. Основные команды меню **Build** следующие:

Compile – компиляция выбранного файла. Результаты компиляции выводятся в окно Output.

Build – компоновка проекта. Компилируются все файлы, в которых произошли изменения с момента последней компоновки. После компиляции происходит сборка (link) всех объектных модулей, включая библиотечные, в результирующий исполняемый файл. Сообщения об ошибках компоновки выводятся в окно Output. Если обе фазы компоновки завершились без ошибок, среда программирования создаст исполняемый файл с расширением .exe (для данного примера: lr1.exe), который можно запустить на выполнение.

Rebuild All – то же, что и Build, но компилируются все файлы проекта независимо от того, были ли в них произведены изменения или нет.

Execute – выполнение исполняемого файла, созданного в результате компоновки проекта. Если же в исходный текст были внесены изменения – то перекompiling, перекompiling и выполнение.

Операции Compile, Build и Execute соответствуют первая, вторая и четвертая кнопки панели инструментов Build MiniBar, которая расположена на рабочем столе (см. рис.3) справа вверху рядом с системными кнопками. Перечислим их слева направо с указанием комбинаций горячих клавиш:

Compile = Ctrl+F7

Build = F7

Execute Program = Ctrl+F5

Откомпилируйте проект, например щелчком на кнопке Build. В процессе компиляции в окне вывода Output отображаются диагностические сообщения компилятора и сборщика. И все в порядке в окне вывода получим следующую последнюю строку:

lr1.exe – 0 error(s), 0 warning(s)

Теперь запускаем приложение на выполнение щелчком, например на кнопке Execute Program (Ctrl+F5). Появится окно приложения rl1, изображенное на рис. 4.4.

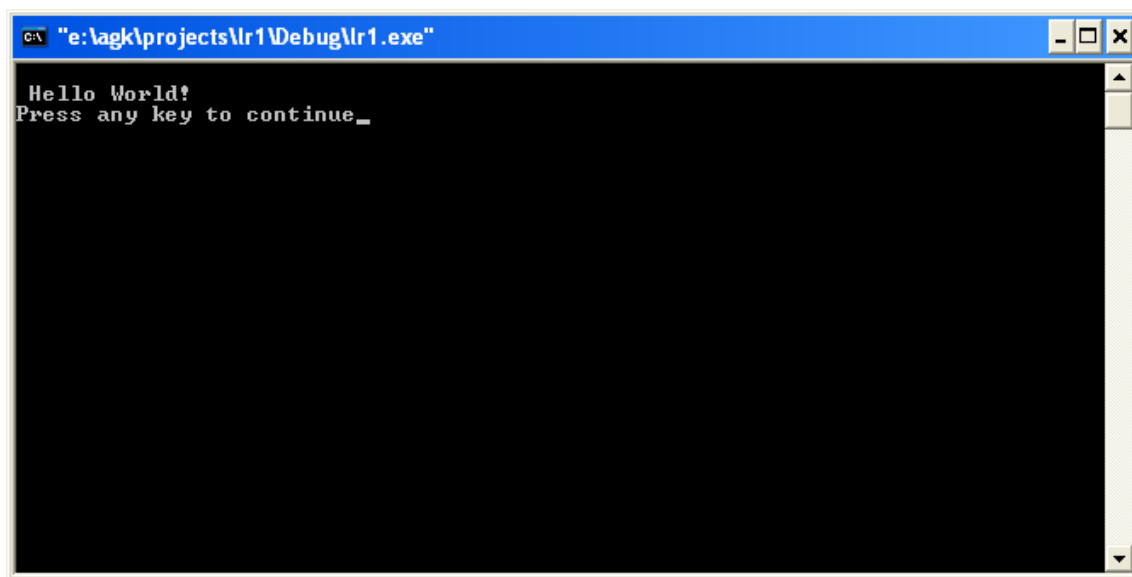


Рис. 4.4

5. Конфигурация проекта

Visual C++ позволяет строить проект либо в отладочной конфигурации (**Win32 Debug**), либо в итоговой конфигурации (**Win32 Release**). Начинать работу нужно в отладочной конфигурации, которая обычно установлена по умолчанию. Для того, чтобы проверить какая текущая конфигурация в проекте, нужно выбрать в подменю **Project** пункт **Settings...** Откроется диалоговое окно **Project Settings**. Смотрим, какое значение установлено в окне комбинированного списка **Settings For:..** Если это не **Win32 Debug**, то переключитесь на нужное значение через команду меню **Build/Set Active Configuration...** Но в отладочной конфигурации даже минимальный проект имеет очень большие размеры. Так только что созданный проект **rl1** имеет размер на диске более одного мегабайта. Поэтому после получения финальной версии проект нужно перекомпилировать в итоговой конфигурации (**Win32 Release**).

Окончание работы над проектом

Можно выбрать меню **File**, пункт **Close Workspace**. А можно просто закрыть приложение Visual C++.

6. Открытие существующего проекта

Для того, чтобы открыть существующий проект нужно проделать следующие действия:

1. Запустить на выполнение среду программирования Visual C++.
2. Выбрать в подменю **File**, пункт **Open Workspace...**
3. В открывшемся диалоговом окне найти папку с нужным проектом, в ней – файл **ProjectName.dsw** и открыть найденный файл, щелкнув по нему мышью.

Или сделать так (если нужный проект был в работе не так давно):

1. Запустить на выполнение Visual C++.
2. Выбрать подменю **File**, навести курсор мыши на пункт **Recent Workspaces**.

3. Если в появившемся меню со списком последних файлов, с которыми шла работа, есть файл **ProjectName.dsw**, щелкнуть по нему мышью.

Или следующим образом:

– не вызывая Visual C++, найти папку с нужным проектом, в ней – файл **ProjectName.dsw**;

– щелкнуть мышью на файле **ProjectName.dsw** и ОС запустит на выполнение среду Visual C++, открыв при этом нужный для работы проект.

Приложение 5. Некоторые возможности отладчика Visual C++

При создании проектов в любой среде программирования одним из важнейших этапов работы являются действия по отладке создаваемых кодов программ.

Приводим начальные возможности отладчика Visual C++.

Проще всего это можно сделать следующим образом: написать программу, содержащую несколько ошибок, и после этого при помощи отладчика показать, как найти и исправить эти ошибки.

Напомним, что синтаксические ошибки – это результат нарушения формальных правил написания программы на конкретном языке программирования.

Логические ошибки делят на ошибки алгоритма и семантические ошибки

Причина ошибки алгоритма – несоответствие построенного алгоритма ходу получения конечного результата сформулированной задачи.

Причина семантической ошибки – неправильное понимание смысла (семантики) операторов выбранного алгоритмического языка.

Создадим консольное приложение под названием Test1 . Программа должна вычислить сумму первых пяти натуральных чисел: от 1 до 5. Ответ: 15.

Текст программы:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
    int i, summa, n=5;
    int a[]={1, 2, 3, 4, 5};
    for(i=1; i<n; i++)
        summa+=a[i];
    printf("\ summa=%d", summa);
}
```

Как видно из текста, синтаксис программы правильный. Учимся отлавливать логические и семантические ошибки. Запускаем программу на выполнение и получаем на экране нечто такое:

Summa=-858993446

(на вашем компьютере может быть и другое число), что мало похоже на 15.

Начинаем отладку программы.

1. Установка точки прерывания

Точка прерывания позволяет остановить выполнение программы перед любой выполняемой инструкцией (оператором) для того, чтобы продолжать дальнейшее выполнение программы или в пошаговом режиме с целью отладки ее по-

следующих участков, в непрерывном режиме до следующей точки прерывания или до конца кода.

Для того, чтобы установить точку прерывания перед оператором, необходимо установить перед ним текстовый курсор и или нажать клавишу **F9** или щелкнуть мышью на кнопке **Insert/Remove Breakpoint** на панели инструментов **Build MiniBar** (крайняя правая кнопка). Точка прерывания обозначится в виде красного кружочка на левом поле окна редактирования перед выбранным оператором. Повторный щелчок на указанной кнопке снимает точку прерывания. В программе точек прерывания можно устанавливать столько, сколько нужно.

Выполнение программы до точки прерывания

Программа запускается в отладочном режиме с помощью команды подменю **Build: Start Debug > Go** (или нажатием клавиши **F5**).

В результате код программы выполняется до той строки, на которой установлена точка прерывания. После чего программа останавливается и отображает в окне редактора ту часть кода, где находится точка прерывания. При этом появляется желтая стрелка на левом поле, указывающая на ту строку, которая будет выполнена на следующем шаге отладки.

Далее установим точку прерывания перед оператором **for** запустим программу в отладочном режиме, нажав клавишу **F5**. Программа выполнится до данного оператора и остановится. При этом подменю **Build** заменилось на меню **Debug**. Вид экрана изображен на рис. 5.1.

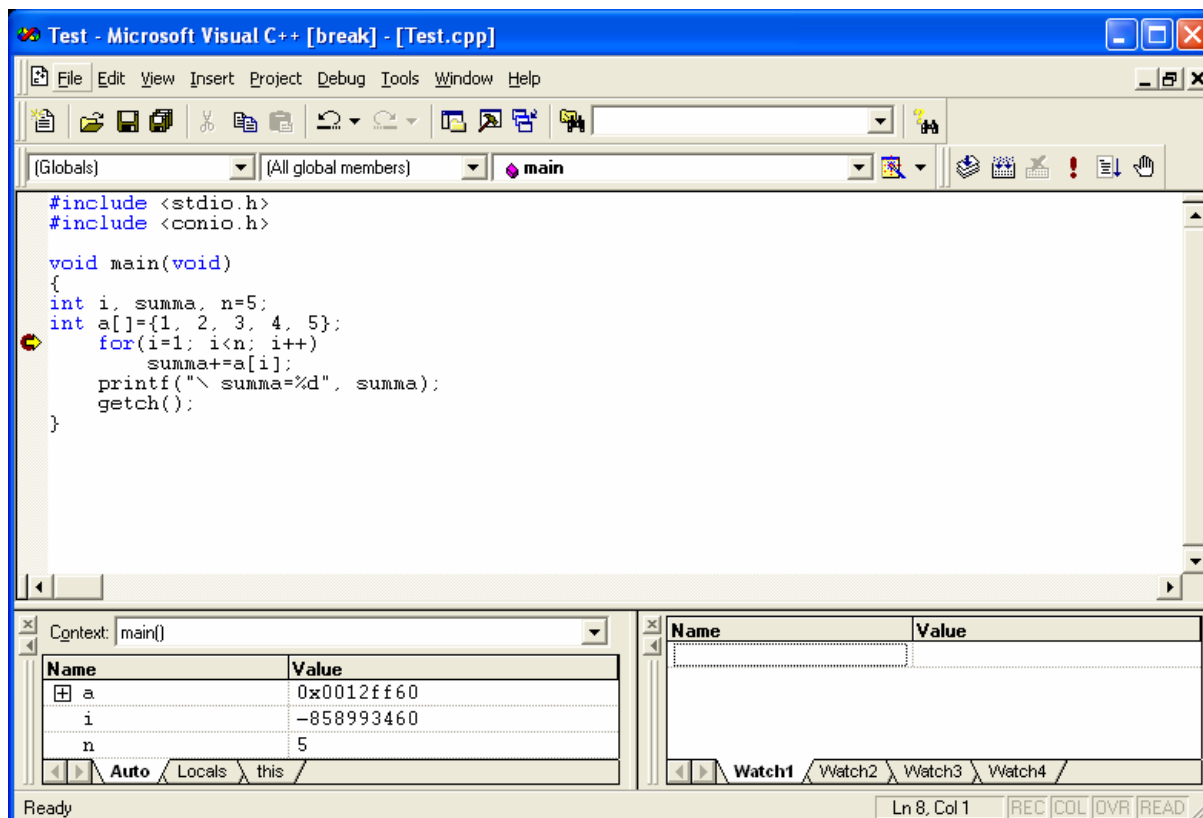


Рис. 5.1

2. Пошаговое выполнение программы

Теперь можно выполнять один оператор оставшейся части программы за другим нажимая клавишу **F10**. При этом есть следующие дополнительные возможности.

1. Предположим, что код программы содержит функцию пользователя `f1()` и что при пошаговом выполнении программы вы дошли до строки, содержащей вызов этой функции.

Теперь имеем две возможности:

– если необходимо просмотреть код вызываемой функции, то надо нажать клавишу **F11**;

– если нужен только результат ее выполнения, то надо нажать клавишу **F10**.

2. Теперь, пусть мы проходим код функции `f1()`, нажав клавишу **F11**. Нажатие клавиш **Shift+F11** обеспечит досрочный выход из нее в тоску ее вызова..

3. Существует возможность пропустить пошаговое выполнение некоторого участка программы: нужно установить текстовый курсор в нужное место программы и нажать клавиши **Ctrl+F10**.

Продолжим отладку. Нажимаем клавишу **F10**.

Указатель переместится на оператор:

Summa+a[i];

3. Проверка значений переменных во время выполнения программы

В любой момент можно узнать текущее значение любой переменной. Чтобы узнать, например значение переменной `summa`, в которой будет накапливаться сумма элементов массива `a`, установите и задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением этой переменной, например, таким (или с другим произвольным значением):

`Summa = -858993446`

Переменная `summa` уже ненулевая, хотя еще не было никакого суммирования.

Ошибка 1 – не обнулена переменная накопления `summa` до входа в цикл по организации суммирования.

Нажимаем комбинацию клавиш **Shift+F5**, чтобы выйти из отладчика, и исправляем найденную ошибку:

int i, summa=0, n=5;

снимаем точку прерывания, компилируем – **Build (F7)**, запускаем на выполнение – кнопка **Execute Program (Ctrl+F5)** и получаем новый результат:

`Summa = 14`

Все равно неправильно – есть еще ошибка. Продолжаем отладку.

Опять устанавливаем точку прерывания перед оператором `for`, запускаем программу в отладочном режиме – клавиша **F5**, нажимаем клавишу **F10** – указатель следующей выполняемой команды переместился на оператор

summa += a[i];

нажимаем клавишу **F10** – указатель следующей выполняемой команды переместится на оператор

```
for(i = 1; i<n; i++)
```

Опять указатель мыши над переменной *summa*, рядом с именем переменной на экране появится подсказка со значением этой переменной

Summa = 2

Сделана только 1-я итерация цикла: переменная накопления должна быть такой – **summa = 1** (значение первого элемента массива), а имеем число 2, то есть значение второго элемента массива.

Ошибка 2 находится в заголовке цикла `for(i = 1; i<N; i++)` – нумерация элементов массива в языке Си начинается с нуля, а переменная цикла с единицы – первоначально из цикла выдергивается второй по счету элемент, а не первый.

Исправляем: **i=0;**

Выйдя из отладчика, снимаем точку прерывания, исправляем текст программы в операторе `for`:

```
for(i = 0; i<n; i++)
```

Вновь компилируем, запускаем на выполнение и получаем:

Summa=15

Все правильно!

4. Окна Auto и Watch 1

Помимо экранной подсказки, переменная *sum* со своим значением отображается в окне Auto, расположенном в левом нижнем углу экрана (см. рис.1). В этом окне приведены значения последних переменных, с которыми работал Visual C++.

Кроме этого, в окне Watch, которое находится в правом нижнем углу, можно задать имя любой переменной, за значениями которой вы хотите понаблюдать.

В заключение напомним, что после отладки проекта необходимо построить его финальную версию, т.к. в **EXE-файле** есть дополнительный отладочный код, включенный через директивы компилятора **_DEBUG**. В папке **Debug** его размер где-то около 1 МБ. Для того, чтобы получить нормальную версию, необходимо переключиться на проект типа **Release**. Это делается в меню **Build**: далее **Set Active Configuration**, дальше надо выбрать **Release** и запустить сборку: **Rebuild all**. Весь отладочный код пропустится, в каталоге проекта появится папка **Release**. Теперь размер **EXE-файла** около 100 КБ.

И ещё, размер проекта зависит от того как используется библиотека **MFC**. Как вы уже знаете есть два варианта: **Static Library** и **Dinamic Library**. Если использовать первый вариант то код **MFC** будет встроен в проект и размер возрастет. Во втором варианте программа будет пользоваться DLL-ками и размер будет меньше.

5. Программные средства отладки

При создании приложения на базе **MFC**, в него можно включить определенные инструкции, используемые только во время его отладки. Их использование замедляет выполнение программы. Цель использования: получение дополнитель-

ных сообщений, которые облегчают процесс отладки. При создании окончательной версии приложения эти операторы следует удалить.

Режим компиляции приложения должен быть Debug (отладочный) так как инструкции – специальные функции и макросы библиотеки MFC, работающие только в отладочном режиме. Кроме того, в отладочной версии приложения Visual C++ с помощью директивы **#define** определяется константа **_DEBUG**, которая используется в директивах условной компиляции и в макросах отладки.

Макрос ASSERT(BOOLexpression). Данный макрос используется для проверки некоторых логических условий, которые должны выполняться в данной точке программы. Его работа такая: если логическое выражение BOOLexpression, передаваемое ему в качестве аргумента имеет значение **FALSE**, выполнение приложения прерывается и на экран выводится окно сообщения, показанное на рис. 5.2. В данном окне указывается имя файла и номер строки, в которой произошла ошибка.

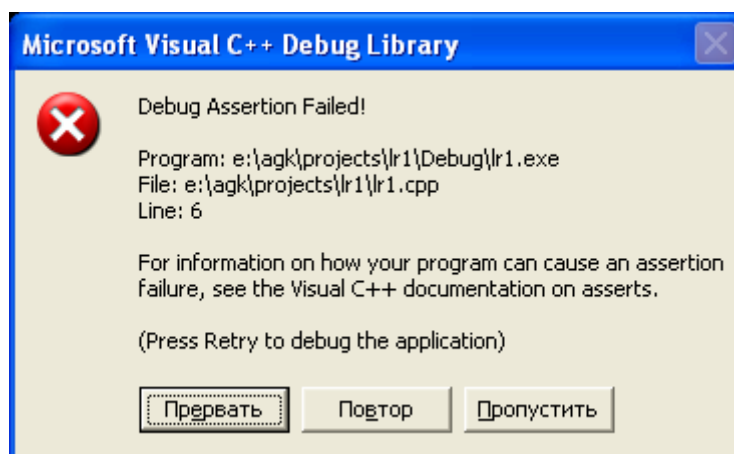


Рис. 5.2

Нажатие кнопки Повтор позволяет перейти в текст программы для ее дальнейшей отладки. Причем текущая точка останова устанавливается на строку соответствующего макроса **ASSERT**. В противном же случае ничего не происходит и программа выполняется дальше.

Макрос TRACE(exp). Служит для вывода диагностических сообщений. Синтаксис макроса TRACE аналогичен синтаксису функции printf. Здесь **exp** – переменное число параметрических аргументов, т.е. макрос позволяет выводить сообщения с любым числом параметрических аргументов. Под параметрическим аргументом понимается идентификатор переменной, значение которой должно быть преобразовано в текстовую строку в соответствии с указанным форматом.. Помимо макроса TRACE существуют TRACE1, TRACE2 и TRACE3. Число в имени макроса указывает на количество параметрических аргументов в данном макросе. Макросы TRACE0, TRACE1, TRACE2 и TRACE3 созданы исключительно с целью экономии места в сегменте данных. Все макросы TRACE посылают свои сообщения в поток afxDump.

Для посылки сообщения могут быть использованы и обычные функции вывода, однако они будут работать и в окончательной версии, что бывает нежелательно.

Следующий пример демонстрирует вывод сообщения о возникновении ошибки в файле:

```
TRACE2("\n Ошибка номер: %d в файле %s \n", nError, szFileName);
```

А следующий пример иллюстрирует работу данного макроса:

```
...  
int a=5;  
char s[]="Minsk";  
TRACE(\n a=%d, s=%s \n", a, s);  
...
```

В поле среды OutPut получим:

```
a=5, s=Minsk
```

В окончательной версии приложения Release, в которой константа `_DEBUG` не определена, макросы `ASSERT` и `TRACE` не выполняют никаких действий. Это позволяет оставлять их в тексте программы. В случае же необходимости контроля некоторых логических условий и в рабочей области, вместо `ASSERT` необходимо использовать макрос `VERIFY(BOOLexpression)`, который работает точно также но в рабочей версии проекта.

Среда программирования Visual C++ 6.0 по умолчанию создает отладочную версию проекта. И это приводит к следующему.

Создайте опять проект Win32 console например с именем Test2 со следующим кодом

```
#include "iostream.h"  
#include "afxwin.h"  
void main()  
{  
    cout << "This not debug" << endl;  
    #ifdef _DEBUG  
        cout << "debug code" << endl;  
    #endif  
    int i;  
    cin >> i;  
}
```

Запустив программу на экране должна появиться надпись:

```
This not debug  
debug code
```

Это говорит о том, что в код включена дополнительная отладочная информация. Т.е. в EXE-файле присутствуют помимо имен переменных, классов, функций, и кода программы дополнительный код, включенный через директивы компилятора `_DEBUG`

Этот файл размещен в папке **Debug** и его размер около одного мегабайта.

Для того, чтобы получить окончательную рабочую версию, необходимо переключиться на проект типа **Release**. Выбираем пункт меню **Build/** далее **Set Active Configuration/** далее **Release** и выполняем: **Rebuild all**. Теперь запустите проект на выполнение. На экране будет:

This not debug

Весь отладочный код компилятор пропустит и создаст в каталоге проекта папку **Release**. Теперь размер EXE-файла в районе 100 КБ.

И ещё: размер зависит от того какой вариант MFC используется в вашем проекте. Есть два варианта: «Static Library» и «Dinamic Library». Если использовать первый вариант то код MFC будет встроен в ваш проект, что приведет к увеличению его размера. При втором варианте размер программы будет меньше.

ЧАСТЬ 2

1. Системы счисления

1.1. Общие определения

Под системой счисления (с/с) понимают способ записи чисел с помощью цифр и символов (букв) [3]. Системы счисления делятся на *позиционные* и *непозиционные*. В позиционных с/с "вес" цифры зависит от ее местоположения, позиции в числе. Непозиционной с/с, например, является римская с/с. В вычислительной технике широко используются позиционные с/с. *Основанием с/с* называется количество цифр и символов, используемых в ней (обозначим его через P). Величина P показывает, во сколько раз численное значение единицы данного разряда больше численного значения единицы предыдущего разряда.

В позиционной с/с число R можно представить в развернутом виде:

$$R = a_l \cdot P^l + a_{l-1} \cdot P^{l-1} + \dots + a_2 \cdot P^2 + a_1 \cdot P^1 + a_0 \cdot P^0 + a_{-1} \cdot P^{-1} + \dots + a_{-k} \cdot P^{-k},$$

где R – запись P -ичной с/с; a_i – весовые коэффициенты, принимающие значения от 0 до $P-1$.

Чаще число R представляется в укороченной форме:

$$R = a_l a_{l-1} \dots a_2 a_1 a_0 a_{-1} \dots a_{-k}.$$

В ЭВМ используется двоичная, восьмеричная, десятичная и шестнадцатеричная с/с. Ниже приведены примеры записи чисел, с/с указывается в скобках.

$$386,53_{10} = 3 \cdot 10^2 + 8 \cdot 10^1 + 6 \cdot 10^0 + 5 \cdot 10^{-1} + 3 \cdot 10^{-2}$$

В памяти ПЭВМ, как и в большинстве других ЭВМ, информация представляется в двоичной системе счисления (2 с/с).

Двоичная система счисления

В двоичной с/с используются две цифры: 0 и 1. Арифметические операции в двоичной с/с выполняются с помощью таблиц по тем же правилам, что и в 10 с/с.

Сложение	Умножение
0+0=0	0*0=0
0+1=1	0*1=0
1+0=1	1*0=0
1+1=10	1*1=1

1.2. Алгоритмы перевода из одной системы счисления в другую

Перевод чисел из p -ичной с/с в q -ичную, если имеет место соотношение $p=q^k$ (k – целое положительное число), наиболее прост. В этом случае перевод из p -ичной с/с в q -ичную осуществляют поразрядно, заменяя каждую p -ичную цифру равным ей k -разрядным числом, записанным в q -ичной с/с. Перевод же из с/с с

основанием q в с/с с основанием p осуществляют следующим образом. Двигаясь от запятой вправо и влево, разбивают запись числа на группы по k цифр. Если при этом крайние группы окажутся неполными, то их дополняют до k -цифр незначащими нулями. Затем заменяют каждую группу цифр ее p -ичным изображением.

Рассмотрим примеры:

а) перевести число 536,712 из 8-ой в 2 с/с ($8=2^3$); разбиваем на триады

$$536,712_{(8)} = (101)(011)(110),(111)(001)(010) = 101011110,111001010_{(2)};$$

б) перевести число BC5,AD4 из 16-ой в 2 с/с ($16=2^4$); разбиваем на тетрады

$$BC5,AD4_{(16)} = (1011)(1100)(0101),(1010)(1101)(0100) = \\ 101111000101,101011010100_{(2)}$$

Рассмотрим более общее правило перевода чисел, пригодное и в том случае, когда не выполняется вышеуказанное соотношение $P=q^k$. В этом случае перевод производится отдельно для целой и дробной частей

Перевод целой части числа

Целую часть числа, записанную в p -ичной с/с, делят на основание новой с/с q до получения целого частного (все операции выполняются по правилам P -ичной с/с). В остатке получается число, являющееся последней (младшей) цифрой записи числа в q -ичной с/с (эта цифра записана в P -ичной с/с, надо перейти в q -ичную с/с). Полученное частное снова делят на основание q ; в остатке будет число, являющееся предпоследней цифрой искомой записи, и т.д. Операцию деления повторяют до тех пор, пока в частном не получат число, меньшее чем q . Это и будет первая (старшая) цифра записи переводимого числа в q -ичную с/с.

Следует отметить, что процесс перевода из 10 с/с в 2 с/с упростится, если число сначала перевести в 8 с/с, а затем воспользоваться правилом 1; целесообразно использовать промежуточную 8 с/с и при переводе из 2 с/с в 10 с/с.

Перевод дробной части числа

Дробную часть числа, записанную в P -ичной с/с, умножают в P -ичной системе на основание q . Целая часть произведения будет первой (старшей) цифрой изображения дроби в q -ичной с/с. Дробную часть произведения снова умножают на q . Целая часть произведения будет следующей цифрой записи дроби в q -ичной с/с. Процесс продолжают до тех пор, пока дробная часть произведения не будет нулевой, или пока не получат требуемое количество знаков записи дроби в q -ичной с/с.

При переводе смешанных чисел отдельно переводят целую и дробную части по вышеизложенным правилам, а затем записывают результаты перевода друг за другом, отделяя целую часть от дробной запятой.

Перевод чисел в 10 с/с

Для исключения выполнения операций умножения и деления в других системах, рекомендуется выполнять способом суммирования с учетом "веса" разрядов по формуле:

$$\dots a_3 a_2 a_1 a_0 a_{-1} a_{-2} (p) \dots = \dots a_3 \cdot p^3 + a_2 \cdot p^2 + a_1 \cdot p^1 + a_0 + a_{-1} \cdot p^{-1} + a_{-2} \cdot p^{-2} \dots (10) .$$

Выражение в правой части записывают и вычисляют в 10 с/с.

Алгоритм перевода из 8-ой в 2-ную с/с:

Начало алгоритма.

1. Подключаем заголовочные файлы: `stdio.h`, `string.h`, `conio.h`, `math.h`.

2. Объявляем `int i`; `char j` – переменные цикла

`l` – счетчик триад

`y` – целое восьмеричное

`y1` – целое двоичное

`char *z1, *z2`; – указатели на начало строк

3. Вводим массив с данными для замены

`s2[8][3]={"000", "001", "010", "011", "100", "101", "110", "111"};`

4. Вводим целочисленное восьмеричное число `y`

5. Преобразовываем число в строку

`itoa(y, z1, 10);`

`k=strlen(z1);` – длина строки равна числу разрядов

6. Цикл перебора «символов цифр» строки

`i=0` до `i<k` шаг по `i=1`

Цикл (выбор: какая цифра в текущем разряде)

`j='0'` до `j<='9'` шаг по `j=1`

Если `z1[i] = j`, копируем соответствующую двоичную триаду в строку `z2`

`strcat(z2, &s2[j-'0'-1][3]); l+=3;`

7. По завершении циклов `z2[l]='\0'`; – устанавливаем конец строки

8. Переводим строку в число

`y1=atoi(z2);`

9. Печатаем результат `y1`.

Конец алгоритма.

Текст программы может быть следующим:

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

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

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

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

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

```
void main(void)
```

```
{
```

```
    int i, k, l=0, y1, y;
```

```
    char *z1, *z2, j;
```



```

char s2[8][3]={“000”,”001”, ”010”, ”011”, ”100”, ”101”, ”110”, “111”};
printf(“\n input y: “); scanf(“%d”, &y);
itoa(y,z1,10);      k=strlen(z1);
printf(“\n k = %d”, k);
printf(“\n y = %s”, z1);
for(i=0; i<k; i++)
{
    for(j='0'; j<='9'; j++)
        if(z1[i]==j)
        {
            strcat(z2,&s2[j-'0'-1][3]);
            l+=3;
        }
}
z2[l]='\0';
printf(“\n z2 = %s”, z2);
y1=atoi(z2);
printf(“ %d “, y1);
getch();
}

```

Алгоритм перевода дробной части числа

Должны ввести количество цифр после запятой и постоянно определять и запоминать целую часть от произведения дробной части предыдущего умножения на основание **p** системы счисления, в которую выполняем перевод.

Начало алгоритма.

1. Подключаем необходимые заголовочные файлы.

2. Объявляем

double i1, y – целая и дробная части числа

z2[100] – результирующий массив

int p,k – основание системы и число знаков

3. Вводим число y=0,xxx и точность перевода

4. Выполняем от i=0 до i<k с шагом i=1

а) $y=y*p$ – последовательное умножение только целой части

б) выделение целой части и дробной

$y = \text{modf}(y, \&i1);$

функция $\text{modf}(\text{double } x, \text{double } *i)$ – выделяет целую и дробную части числа типа double, дробная часть возвращается функцией, а целая записывается по адресу указателя **i**;

в) заносим целую часть в массив $z2[i]=i1$, если $y=0$, то прервать!

5. Печать дробной части:

$\text{printf}(\text{“}\backslash\text{n } y1 = 0.\text{”});$

Меняя **j** от 0 до **i** с шагом 1 выводим $z2[j]$

Конец алгоритма.

Текст программы перевода дробной части числа может быть следующим:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main(void)
{
    int p, k, i;
    double y, i1;
    int z2[100];
    p=8; // для 8-ой
    printf("\n input y: "); scanf("%lf", &y);
    printf("\n input k: "); scanf("%d", &k);
    for( i=0; i<k; i++)
    {
        y=y*p;
        y=modf(y,&i1);
        z2[i]=i1;
        printf("\n xxx = %d   y = %lf", z2[i], y); // контроль
    }
    printf(" i = %d", i);
    printf("\n xxx = %d   \n", z2[i]);
    printf(" y = %lf\n", y); // результат перевода
    printf(" 0. ");
    for(i=0; i<k; i++)
        printf(" %d ", z2[i]);
    getch();
}
```

Алгоритм перевода целой части

При выполнении перевода мы должны на текущем шаге выделять остаток от деления $<p$ и запоминать его. Потом запоминать частное от деления и вывести результат в обратном порядке.

Начало алгоритма.

1. Подключаем необходимые заголовочные файлы.

2. Объявляем: целые

j=0 – переменная цикла

i=0 – счетчик разрядов нового числа

p – основание новой с\с

y – число для перевода

z2[100] – массив с разрядами нового числа

3. Вводим y и p.

4. Начало цикла, выполнять пока $y \geq p$

- а) занесение остатка от текущего шага деления
 $z2[i] = y \% p;$
- б) $y = y / p;$ – выделяем частное, полученное на текущем шаге, использовали тот факт, что результат деления целого на целое равно целому
- в) i увеличили на единицу

Конец цикла.

5. Увеличили i на единицу и запоминаем последнее частное от деления
 $z2[i] = y;$

6. Печатаем массив в обратном порядке
 j от $i-1$ до $j \geq 0$ шаг уменьшаем на 1
 печать $z2[j];$

Конец алгоритма.

Текст программы перевода целой части числа может быть следующим:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
    int i=0, j=0, p, y, z2[100];
    p = 2; // для 2-ой
    printf("\n input y: "); scanf("%d", &y);
    while (y >= p)
    {
        z2[i] = y % p;
        printf("\n xxx = %d ", z2[i]);
        y = y / p;
        i++;
    }
    i++;
    z2[i] = y;
    printf(" i = %d", i);
    printf("\n xxx = %d \n", z2[i]);
    printf(" y = %d", y); // результат перевода
    for (j = i - 1; j >= 0; j--)
        printf(" %d ", z2[j]);
    getch();
}
```

Пример перевода числа из любой системы счисления в десятичную

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

```

#include <stdlib.h>
#include <math.h>
void main(void)
{
    int i, j, k, p, y, y1=0;
    char *z2;
    p=2;
    printf("\n input y: "); scanf("%d", &y);
    itoa(y,z2,10);          // ввели число и преобразовали его в строку
    k = strlen(z2);
    for( i=0; i<k; i++)
    {
        printf ("\n %d - %c", i, z2[i]);
        j=z2[i]-'0';
/* извлекаем коэффициенты текущего разряда и добавляем слагаемое в сумму –
укороченная запись числа */
        printf ("\n j= %d ", j);
        y1+=j*pow(p,k-i-1);
    }
    printf(" %d ", y1);
    getch();
}

```

2. Организация памяти и структуры данных

Виды организации хранения данных в памяти

Практически все языки программирования предоставляют возможность манипулирования множеством разнообразных структур данных. Так, например, в языке Си рассматриваются скалярные переменные, массивы, строки, структуры и т. д. Все объекты программы требуют декларации (описания) их типа с последующим определением. Это позволяет компилятору зафиксировать в оперативной памяти (ОП) их конкретные размеры. Такие объекты, как правило, называют статические.

Механизм хранения данных в ОП может иметь или векторную или списковую организацию памяти.

Векторная память и адресная функция

Векторная память поддерживается почти всеми языками высокого уровня и предназначена для хранения массивов различной размерности. Каждому массиву выделяется непрерывный участок памяти указанного размера. При этом, например, элементы трехмерного массива X размещаются в ОП в следующем образом:

$X(1,1,1), X(1,1,2), X(1,1,3), \dots, X(n1,n2,n3-1), X(n1,n2,n3).$

Адресация элементов массива определяется некоторой функцией (адресной), связывающей адрес массива и индексы элемента.

Пример адресной функции для однородного трехмерного массива $X(n1, n2, n3)$:

$$K(i, j, k) = n1 * n2 * (i-1) + n2 * (j-1) + k; \quad i = \overline{1, n1}; \quad j = \overline{1, n2}; \quad k = \overline{1, n3}.$$

Для размещения такого массива потребуется участок ОП размером $n1 * n2 * n3$ элементов или $n1 * n2 * n3 * \text{sizeof}(type)$ байт (операция $\text{sizeof}(type)$ возвращает количество байт для размещения элемента массива типа $type$). Рассматривая эту область как вектор (одномерный массив) $Y(1, \dots, n1 * n2 * n3)$, можно установить соответствие между элементом трехмерного массива X и элементом одномерного массива Y :

$$X(i, j, k) \Leftrightarrow Y(K(i, j, k)).$$

Адресная функция двумерного массива $A(n, m)$ будет выглядеть так:

$$N1 = K(i, j) = n * (i-1) + j; \quad i = \overline{1, n}; \quad j = \overline{1, m}.$$

Тогда справедливо следующее:

$$A(i, j) \Leftrightarrow B(K(i, j)) = B(N1), \quad i = \overline{1, n}; \quad j = \overline{1, m};$$

B – одномерный массив размера $N1 = 1, \dots, n * m$.

Например, для двумерного массива $A(2, 2)$ имеем:

	$A(1, 1)$	$A(1, 2)$	$A(2, 1)$	$A(2, 2)$
	1	2	3	4
$i=1, j=1$	$N1 = 2 * (1-1) + 1 = 1$			$B(1)$
$i=1, j=2$	$N1 = 2 * (1-1) + 2 = 2$			$B(2)$
$i=2, j=1$	$N1 = 2 * (2-1) + 1 = 3$			$B(3)$
$i=2, j=2$	$N1 = 2 * (2-1) + 2 = 4$			$B(4)$

Необходимость введения адресных функций возникает лишь в случаях, когда требуется изменить способ отображения с учетом особенностей конкретной задачи.

Например, следующая адресная функция обеспечит компактное представление верхней треугольной матрицы X с диагональю порядка n :

$$K(i, j) = (n - i/2) * (i-1) + j, \quad \text{где } i = 1, 2, \dots, n; \quad j = i, i+1, \dots, n.$$

Для хранения элементов такой матрицы достаточно выделить в оперативной памяти одномерный массив: $Y(1 \dots n * (n+1)/2)$, а для доступа к его элементам использовать следующее соотношение:

$$X(i, j) \Leftrightarrow Y(K(i, j)), \quad i = \overline{1, n}; \quad j = \overline{i, n};$$

(допустимые комбинации значений индексов здесь строго фиксированы условиями задачи).

Списковая организация памяти

Некоторые задачи исключают использование структур данных фиксированного размера и требуют введения структур динамических, способных увеличивать или уменьшать свой размер уже в процессе работы программы. Основу таких структур составляют динамические переменные.

Динамическая переменная хранится в некоторой области ОП, не обозначенной именем, и обращение к ней производится через переменную-указатель.

Динамическое распределение ОП связано с операциями порождения и уничтожения объектов по запросу программы, т.е. захват и освобождение памяти производится программно в процессе работы. При этом в стандарте языка Си таких операций нет, а порождение объектов (захват памяти) и уничтожение объектов (освобождение памяти) производится при помощи библиотечных функций (см. Часть 1, п. 13).

В Си++ введены две операции по захвату и освобождению участков динамической памяти *new* и *delete*, например:

1) *type *p = new type[n]*; – возвращает указатель на ОП размером *n*sizeof(type)*;

...

delete []p; – освободили всю захваченную память;

2) *type *p = new type(5)*; – захватили участок памяти размером *sizeof(type)*, установив на него указатель, и записали в захваченную область значение 5;

...

**p=12*; – заменили 5 на 12;

delete p; – освободили захваченную под элемент память.

Как правило, динамические переменные организуются в списковые структуры данных.

Списковая организация памяти применяется в случаях, когда возможность прямого доступа к любому именованному элементу памяти не используется, а решение задачи практически удовлетворяется лишь возможностью перехода от одного элемента данных к другому. Очевидно, что в таких случаях необязательно размещать последовательно обрабатываемые элементы в смежных областях памяти.

Для адресации элементов данных достаточно каждый элемент дополнить полем ссылки (указателем) на область размещения следующего элемента, и отдельно задать ссылку на первый элемент списка.

Использование указателей для связи элементов данных позволяет легко отразить различные виды взаимосвязей данных. В графическом представлении списков указатели представляются стрелками, начало которых соответствует области хранения указателя, а окончание – адресуемому указателем элементу.

Пример из жизни: очередь на прием в какой-нибудь кабинет. Каждый запоминает человека, за которым занял очередь – все связаны в цепочку, но в пространстве распределены произвольно. Вновь пришедший человек занимает очередь за последним, запоминает его и садится на свободное место.

Так и в памяти элементы связанной структуры размещаются там, где есть свободное место и каждый элемент должен знать, за кем он стоит, т.е. содержать ссылку (указатель) на предыдущий элемент цепочки.

Пусть один покидает очередь, это не требует перемещения в пространстве остальных, стоящий за покинувшим очередь человеком просто запоминает того человека, за которым ушедший занимал очередь. Т.е. исключение элемента из це-

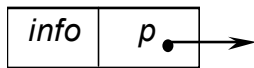
почки не требует перемещения остальных элементов в памяти – элемент удаляется из списка, освобождается память, занятая им и эту память можно использовать для других целей.

При помощи указателей легко добавить новый элемент, достаточно изменить значения двух указателей. При этом новый элемент может быть размещен в любом месте памяти.

Как правило, элементы таких данных (списков) имеют тип *struct*.

Описание связанных данных

Пусть необходимо создать линейный список, содержащий целые числа, тогда каждый элемент списка должен иметь информационную (*info*) часть, в которой будут находиться данные и адресную (*p*) часть, в которой будут размещаться указатели связей, т.е.



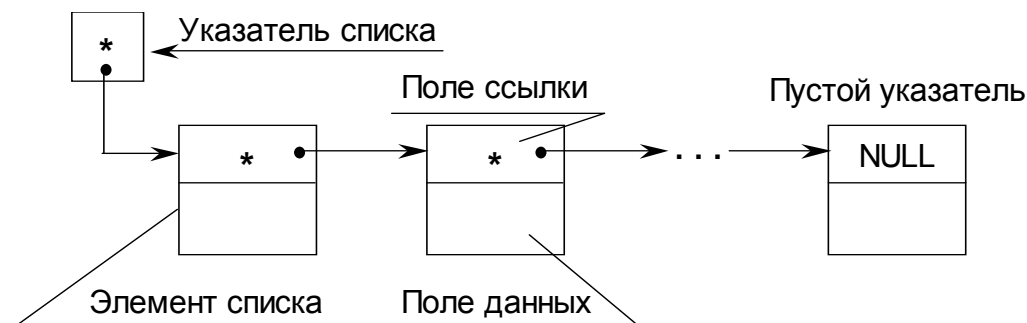
Шаблон объявления элемента такой структуры будет иметь вид:

```
struct spis {  
    int info;  
    spis *p;  
};
```

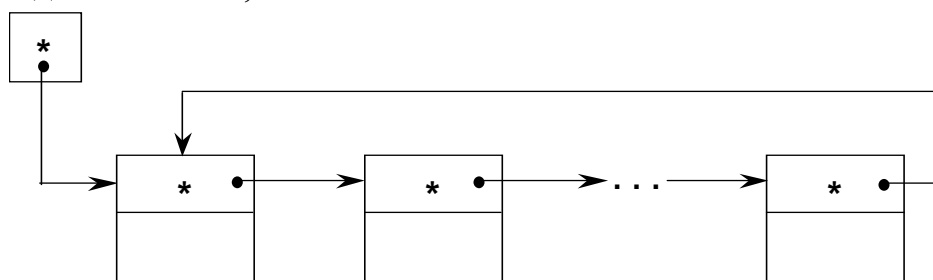
Важным понятием при манипулировании указателями является значение "пустого" указателя (*NULL*). Такой указатель используют для обнаружения окончания списковой последовательности.

Основные виды списковых структур

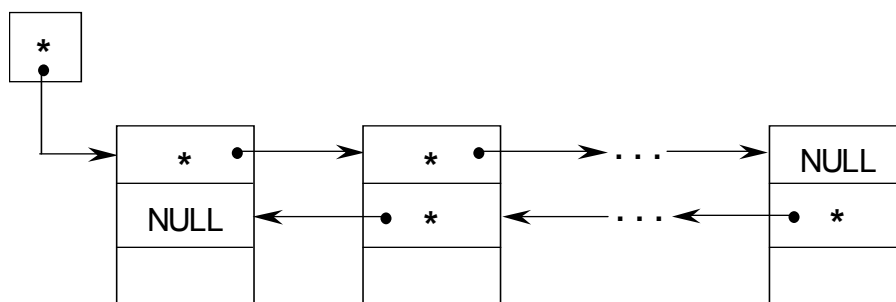
Линейный список (список) – организация некоторого множества элементов данных, при которой каждый элемент содержит собственно данные и указатель на расположение в памяти следующего элемента множества.



Циклический список, в отличие от линейного, допускает многократное прохождение списка, начиная с любого элемента.



Симметричный (двухсвязный) список позволяет организовать продвижение по связям в любом направлении, и удобен для выполнения операций включения и исключения элементов в заданном месте. Очевидно, что нет принципиальных ограничений на вид структур, представляемых списками. Если элементом списка является другой список, говорят о списковых структурах.



Таким образом, совокупность обрабатываемых элементов данных и взаимосвязей между ними без учета их физического размещения вводит новое понятие – абстрактная структура данных.

Приведем определения некоторых наиболее часто используемых абстрактных структур.

Множество – неупорядоченная совокупность элементов (данных). По определению, множество не содержит повторяющихся элементов. Если значения элементов повторяются, то говорят о комплекте.

Очереди и стеки – множества, элементы которых упорядочены по времени включения.

Очередь предполагает доступность только того элемента, который был помещен в нее ранее других.

Очередь с двумя концами – дек, допускает выполнение операций записи – выборки с обоих концов.

В стеке доступен только тот элемент, который был помещен позже других. Обращение к стеку связано с понятием – вершина стека.

Конечная последовательность элементов некоторого алфавита – строка.

Ориентированный граф – пара множеств вершин и дуг, где элементы множества дуг – упорядоченные пары вершин. Для нагруженного графа дополнительно определяется соответствующее каждой дуге значение (вес дуги).

Дерево – частный случай ориентированного графа, где имеется лишь одна вершина с нулевой степенью захода, а остальные вершины имеют единичную степень захода (степень захода – число дуг, входящих в вершину графа).

Таблица – множество элементов, организованное таким образом, что каждый элемент и, возможно, его расположение однозначно определяются его ключом. В отличие от списков, элемент таблицы идентифицирует только самого себя.

Таблица соответствует понятию файла, а ее элементы часто называют записями. Идентификация записей ключами оправдана тем, что обычно для различения записей существенна только небольшая часть записи, называемая ключом. Абстрактные структуры данных могут отображаться как в векторной, так и списковой памяти. Выбор вида организации памяти определяется задачами использования представляемых данных.

Отображение структур данных в памяти

Итак, любые структуры данных МОГУТ отображаться в памяти распределяемой как статически – объект будет создан на этапе компиляции, так и динамически – на этапе выполнения программы.

Статическое распределение реализуется средствами создания операционных объектов языка программирования на этапе трансляции и/или вызова программного модуля.

Примеры организации динамических данных

1. Создание динамической переменной

`#include <alloc.h>`

...

`double *v;` – декларировали указатель на участок памяти, в котором можно хранить вещественные данные с удвоенной точностью;

`v = (double*)malloc(sizeof(double));` – захватили участок памяти, установив на него указатель;

```

if(v==NULL) {           – контроль за возможной ошибкой;
printf("\n Error! ");
getch();
return;                // exit(1);
}
*v = 12.34;             – косвенная инициализация;
...                    – работаем с динамической памятью;
free(v);                – освобождаем память;

```

2. Создание одномерного динамического массива

`#include <alloc.h>`

```

...
double *x;
int i, n;
scanf("%d",&n);          – ввод размерности массива;
x = (double*)calloc(n, sizeof(double));
if(x==NULL) {             – контроль за возможной ошибкой (см. п.1);
... }
for ( i=0; i<n; i++) scanf("%lf", &x[i]);    – ввод элементов массива;
...
for ( i=0; i<n; i++) printf("\n %lf", x[i]);  – вывод элементов массива;
...
free(x);

```

3. Создание динамической переменной, используя операции **new** и **delete**

```

...
int *a;
a = new int(5);
...           – работаем через косвенную адресацию;
delete a;     – освобождаем память;

```

4. Создание одномерного массива, используя операции **new** и **delete**

```

...
float *x;
int i, n;
scanf("%d",&n);    – ввод размера массива;
x = new float[n];
if(x==NULL)
{
... }              // Контроль за возможной ошибкой (см. п.1);
for ( i=0; i<n; i++) scanf("%f", (x+i));    – ввод элементов массива
(адреса);
for ( i=0; i<n; i++) printf("\n %f", *(x+i)); – вывод массива (значения);

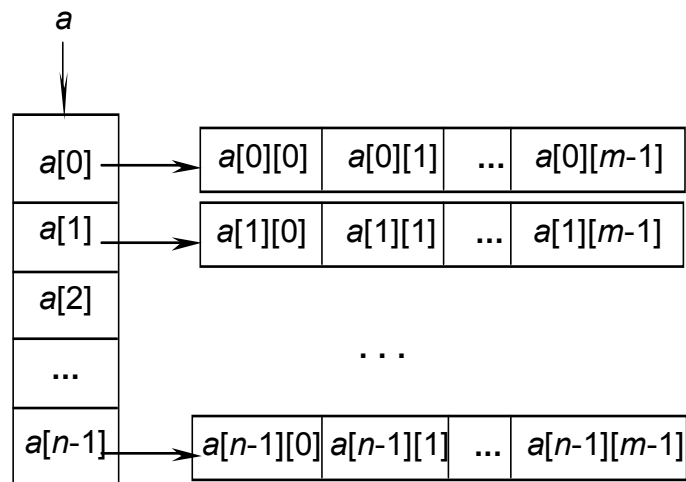
```

...
`delete []x;` – освободили память;

...

5. Создание двумерного массива, используя операции *new* и *delete*

Напомним, что связь между указателями $a[i]$ для двумерного массива $a[n][m]$ выглядит следующим образом



```

...
float **a;
int i, j, n, m;
scanf("%d %d", &n, &m);
a = new float* [n];
for ( i=0; i<n; i++) *(a+i) = new float [m];

...
for ( i=0; i<n; i++)
    for ( j=0; j<m; j++) scanf("%f", &a[i][j]);
...
delete [ ]a;
...

```

3. Линейные списки и рекурсия

Как было показано ранее простой способ связать множество элементов – сделать так, чтобы каждый элемент содержал ссылку на следующий. Такой список называют *однонаправленным (односвязным)*. Если добавить в каждый элемент ссылку на предыдущий элемент, получится *двунаправленный список (двусвязный)*, если последний элемент связать указателем с первым, получится *кольцевой список*.

Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных.

Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа. При упорядочивании такого списка по алфавиту ключом будет фамилия, а при поиске, например, ветеранов труда – ключом будет стаж работы. Как правило, ключи должны быть уникальными, но в ряде случаев ключи разных элементов списка могут совпадать. И в последнем случае лучше всего использовать схемы организации структур данных по принципам «хеширования».

Над списками можно выполнять следующие *операции*:

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- обработка (чтение, удаление и т.п.) элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- упорядочивание списка по ключу.

Если программа состоит из функций, решающих вышеперечисленные задачи, то необходимо соблюдать следующие требования:

- все параметры, не изменяемые внутри функций, должны передаваться с модификатором *const*;
- указатели, которые могут изменяться (например, при удалении из списка последнего элемента указатель на конец списка требуется скорректировать), передаются по адресу.

Удаление элемента из списка

Рассмотрим подробнее *функцию удаления элемента из списка*. Ее параметрами должны быть указатели на начало списка, на конец списка и ключ элемента, подлежащего удалению.

Удаление из списка происходит по-разному в зависимости от того, находится элемент в начале списка, в середине или в конце. Сначала нужно проверить, находится ли удаляемый элемент в начале списка – в этом случае следует скорректировать указатель *begin* на начало списка так, чтобы он указывал на следующий элемент в списке, адрес которого находится в адресном поле первого элемента. Новый начальный элемент списка должен иметь в поле указателя на предыдущий элемент значение *NULL*.

Если удаляемый элемент находится в конце списка, требуется сместить указатель *end* конца списка на предыдущий элемент. Кроме того, нужно обнулить для нового последнего элемента указатель на следующий элемент. Если удаление происходит из середины списка, то единственное, что надо сделать, – обеспечить связь предыдущего и последующего элементов. После корректировки указателей память из-под удаленного элемента освобождается.

Сортировка элементов списка

Сортировка связанного списка заключается в изменении связей между элементами. В данном случае наиболее прост линейный выбор. Алгоритм состоит в

том, что исходный список просматривается, и каждый элемент вставляется в новый список на место, определяемое значением его ключа.

При работе со списками удобно пользоваться рекурсивными алгоритмами и прежде, чем рассмотреть рекурсивные приемы работы со списками вспомним, что такое *рекурсия*.

Рекурсивные функции

Напомним, рассмотренные в первом семестре основные сведения.

Рекурсивной (само вызываемой или само вызывающей) называют функцию, которая прямо или косвенно вызывает сама себя.

Вторая основная особенность – при каждом обращении к рекурсивной функции создается новый набор объектов автоматической памяти, локализованных в коде функции.

Возможность прямого или косвенного вызова позволяет различать прямую или косвенную рекурсии. Функция называется косвенно рекурсивной в том случае, если она содержит обращение к другой функции, содержащей прямой или косвенный вызов первой функции. В этом случае по тексту определения функции ее рекурсивность (косвенная) может быть не видна. Если в функции используется вызов этой же функции, то имеет место прямая рекурсия, т.е. функция, по определению, рекурсивная.

Рекурсивные алгоритмы эффективны в задачах, где рекурсия использована в самом определении обрабатываемых данных. Поэтому изучение рекурсивных методов нужно проводить, вводя динамические структуры данных с рекурсивной структурой, что далее и будет сделано. Рассмотрим вначале только принципиальные возможности, которые предоставляет язык Си для организации рекурсивных алгоритмов.

В рекурсивных функциях необходимо выполнять следующие правила:

- при каждом ее вызове в функцию передавать модифицированные данные;
- на некотором шаге должен быть прекращен дальнейший вызов этой функции, т.е. рекурсивный процесс должен шаг за шагом так упрощать задачу, чтобы для нее появилось не рекурсивное решение (чтобы не допустить ошибку, заключающуюся в том, что функция будет вызывать себя бесконечно);
- после завершения каждого экземпляра рекурсивной функции в вызывающую функцию должен возвращаться некоторый результат для дальнейшего его использования.

Рекурсия при обработке списков

Структура, такая как динамический односвязный список, представляет собой конструкцию, которую удобно обрабатывать при помощи рекурсивных процедур. Рассмотрим формирование списка и вывод списка на экран дисплея в виде рекурсивных функций. Сделаем это с целью показать на этом простом примере особенности рекурсивной обработки динамических информационных структур.

Например, пусть, как и раньше данная структура имеет вид:

```

struct Spis {
    int info;
    Spis *Next;
};

```

В каждом звене списка имеется информационная часть (данные – поле *info*) и адресная часть (адрес на следующее звено списка – *Next*). Если адресная часть нулевая (*NULL*), то список пройден до конца. Для начала просмотра списка нужно знать только адрес его первого элемента.

Рассмотрим, какие действия должны выполнять функция рекурсивного формирования списка и функция его рекурсивного просмотра с выводом данных о звеньях списка на экран дисплея.

Функция рекурсивного заполнения списка *input()* должна возвращать указатель на сформированный и заполненный данными с клавиатуры динамический список. Предполагается, что при каждом вызове этой функции формируется новый элемент списка.

Функция заканчивает работу, если для очередного звена списка для информационной части (*info*) введено нулевое значение переменной.

В противном случае заполненная с клавиатуры структура подключается к списку, а ее элементу (*Spis *Next*) присваивается значение, которое вернет вызванная рекурсивно функция *input()*.

Прежде чем рассматривать текст функции *input*, помещенный в приведенной ниже программе, еще раз сформулируем использованные в ней конструктивные решения.

1. При каждом обращении к этой функции в памяти формируется новый элемент списка, указатель на начало которого, возвращает эта функция.

2. Выделение памяти для элементов списка и заполнение их значениями полностью определяет пользователь.

3. Если для очередного звена списка вводится нулевое значение элемента *info*, то звено не подключается к списку, процесс формирования списка заканчивается. Такой набор данных, введенных для элемента очередного звена, считается терминальным.

4. Если нулевое значение элемента *info* введено для первого звена списка, то функция *input* возвращает значение *NULL*.

5. Список определяется рекурсивно как первое (головное) звено, за которым следует присоединяемый к нему список.

6. Псевдокод рекурсивного алгоритма формирования списка:

Если введены терминальные данные для звена:

- освободить память, выделенную для звена;
- вернуть нулевой указатель.

Иначе:

- элементу связи звена присвоить результат формирования продолжения списка (использовать тот же алгоритм).

Все-если вернуть указатель на звено.

Для структуры типа *Spis* выделяется память. После того как пользователь введет данные, выполняется их анализ. В случае терминальных значений операция *delete* освобождает память от ненужного звена списка и выполняется оператор *return NULL*;

В противном случае элементу связи (указатель *Spis *Next*) присваивается результат рекурсивного вызова функции *input*. Далее все повторяется для нового экземпляра этой функции. Когда при очередном вызове *input* пользователь введет терминальные данные, функция *input* вернет значение *NULL*, которое будет присвоено указателю связи предыдущего звена списка, и выполнение функции будет завершено.

Функция рекурсивного просмотра и печати списка. Аргументом функции является адрес начала списка (или адрес любого его звена), передаваемый как значение указателя – параметра *Spis *begin*. Если параметр имеет значение *NULL* – список исчерпан, и исполнение функции прекращается. В противном случае выводятся на экран значения элементов той структуры, которую адресует параметр, и функция *print* рекурсивно вызывается с параметром *begin->Next*. Тем самым выполняется продвижение к следующему элементу списка. Конец известен – функция печатает "Список исчерпан!" и больше не вызывает себя, а завершает работу.

Текст программы с рекурсивными функциями:

```
#include <stdlib.h>
#include <stdio.h>
struct Spis {
    int info;
    Spis *Next;
};

//----- Функция ввода и формирования списка -----
Spis* input(void) {
    int i;
    Spis *t = new Spis;
    printf("\n info ? : ");
    scanf("%d", &i);
    if ( i == 0 ) {
        delete t;
        return NULL;
    }
    t->info = i;
    t->Next = input ();
    return t;
}

//----- Функция вывода списка на экран -----
void print (Spis *t) {
    if (t == NULL) {
        printf ("\n Spisok - END! ");
    }
```

```

    return ;
}
printf ("\n Info = %d ", t->info) ;
print ( t -> Next );           // Рекурсивный вызов
}

//-----
void main () {
    Spis *begin = NULL;        // Начало списка
    printf ("\n Vvedi Info : ") ;
    begin = input () ;         // Ввод списка
    printf ("\n Spisok : ") ;  // Напечатать список
    print ( begin );
    getch();
}

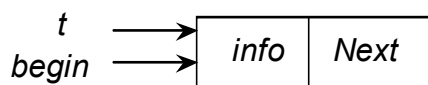
```

Рассмотрим подробно работу программы формирования списка с использованием функции *input*.

1. Объявлен указатель на начало списка и изначально установлен в *NULL*. Запустив на выполнение функцию *main*, через *begin = input()*; будет вызвана функция *input* первый раз (*input* возвращает указатель на структуру).

2. Теперь работает функция *input()*; (первый шаг):

1) захват памяти: *t = new Spis*;



2) формирование информационной части *info*;

3) если введен нуль: (*i = 0*), то освобождаем захваченную память и возвращаем *NULL*, т.е. *begin = NULL*; – список закончился так и не начавшись.

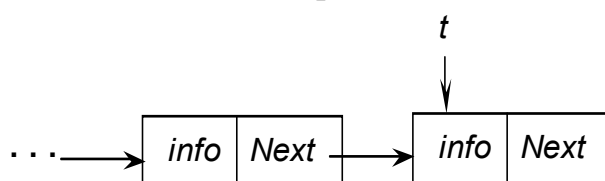
4) иначе, т.е. *i ≠ 0*, используя указатель структуры *t* на тип *Spis* вызывается функция *input* на следующий шаг *t -> Next = input()*; и возвращает значение указателя *t* на первый созданный элемент: *return t*; таким образом, указатель на начало *begin* установлен на этот элемент.

Последующие вызовы рекурсивной функции.

1. Работает функция *input*, вызванная через указатель последнего созданного элемента.

1) текущий захват памяти под новый элемент (напомним, элементы добавляются в список с конца):

t = new Spis;



- 2) формирование информационной части *info*;
- 3) если $i = 0$, то освобождаем захваченную память для созданного элемента (*delete t*;) и возвращаем *NULL*, т.е. формирование очереди закончилось.
- 4) иначе, т.е. $i \neq 0$, формируем информационную часть *i*, используя адресную часть только что созданного элемента, опять обращаемся к функции *input*:
 $t \rightarrow \text{Next} = \text{input}()$;
 и возвращаем указатель на только что созданный (последний) элемент, который будет размещен в адресной части теперь предпоследнего элемента.

4. Динамическая структура данных – СТЕК

Рассмотрим алгоритмы работы со спискавыми динамическими данными на примере структуры под названием «СТЕК».

Стек – упорядоченный набор данных, в котором размещение новых элементов и удаление существующих производится только с одного его конца, который называют вершиной стека, т.е. стек это список с одной точкой доступа к его элементам. Графически это можно изобразить так:



Стек – структура типа **LIFO** (*Last In, First Out*) – последним вошел, первым вышел: последний добавленный элемент, который первым выйдет. Стек получил свое название из за схожести с оружейным магазином с патронами (обойма) – когда в стек добавляется новый элемент, то прежний проталкивается вниз и временно становится недоступным. Когда верхний элемент удаляется из стека, следующий за ним поднимается вверх и становится опять доступным.

Максимальное число элементов стека не должно ограничиваться, т.е. по мере вталкивания в стек новых элементов, память под него должна динамически запрашиваться и освобождаться также динамически при удалении элемента из стека. Таким образом, стек – динамическая структура данных, состоящая из переменного числа элементов одинакового типа.

Состояние стека рассматривается только по отношению к его вершине, а не по отношению к количеству его элементов, т.е. только вершина стека характеризует его состояние.

Операции, выполняемые над стеком, имеют специальные названия:

push – добавление элемента в стек (вталкивание);

pop – выталкивание элемента из стека, верхний элемент стека удаляется (не может применяться к пустому стеку).

Кроме этих обязательных операций часто нужно прочесть значение элемента в вершине стека, не извлекая его оттуда. Такая операция получила название *peek*.

Стек можно реализовать или на базе динамических операций с ОП или на базе массивов. Рассмотрим реализацию списковой структуры стек на базе динамической памяти. Информационная часть элементов стека будет содержать целые числа. Тип *int* взят только для простоты рассмотрения, хотя информационная часть может состоять из любого количества объектов любого допустимого типа за исключением файлов.

Ниже приведены основные словесные алгоритмы для работы со стеком.

Алгоритм формирования первых 2-х элементов

1. Описать структуру переменной (рекомендуется шаблон структуры описывать глобально):

struct Stack →

info	Next
------	------

```
struct Stack {
    int info;
    Stack *Next;
};
```

2. Объявить два указателя на структуру:

Stack *begin (вершина стека), *t (текущий элемент);

3. Стек пуст, поэтому вершина в NULL: begin = NULL; (такое значение необходимо для первого вталкиваемого в стек элемента); графически это можно изобразить так:

begin →

NULL

4. Захват памяти под первый (текущий) элемент:

t = new Stack;

формируется конкретный адрес (A1) для первого элемента, т.е. t равен A1.

5. Считываем информацию (i1);

а) формируем информационную часть:

t -> info = i1;

t →

info = i1	Next
-----------	------

б) значение вершины – в адресную часть (там был NULL)

t -> Next = begin;

6. Вершину стека переносим на созданный (1-ый) элемент:

begin = t;

получили:

begin (=A1) →

info = i1	NULL
-----------	------

7. Опять захватываем память (под второй элемент):

t = new Stack;

имеем конкретный адрес (A2) для второго элемента, т.е. по шагам

8. Считываем для него информацию (i2);

а) формируем информационную часть:

t -> info = i2;

б) в адресную часть – значение вершины, т.е. адрес предыдущего (первого) элемента (A1):

t -> Next = begin;

t (=A2) →

info = i2	Next = A1
-----------	-----------

9. Вершину стека снимаем с первого и устанавливаем на новый элемент (A2):

begin = t;

получим:

begin (=A2) →

Info=i2	Next=A1
---------	---------

 →

Info=i1	Next=NULL
---------	-----------

Алгоритм добавления текущего элемента

1. Захват памяти под текущий элемент (формирование конкретного адреса).
2. Формирование информационной части.
3. Формирование адресной части:
 - в адресную часть значение вершины;
 - в вершину значение адреса нового элемента.

Алгоритм извлечения элемента из стека

Рассмотрим данный алгоритм с обработкой и уничтожением.

1. Используя вершину стека, выдавливаем последний занесенный элемент:

t = begin;

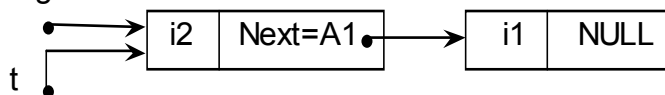
2. Обрабатываем информационную часть (t -> info) (выводим на экран).

3. Вершине стека присваиваем значение адресной части, т.е. вершину переставляем на следующий элемент: begin = t -> Next;

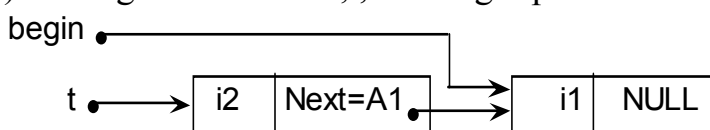
4. Уничтожаем обработанный элемент, т.е. освобождаем занятую под него память: delete t;

Для первых двух элементов рассмотрим подробно:

begin=A2



- 1) t = begin;
- 2) t -> info; например, печать i
- 3) begin = t -> Next; ,т.е. begin равен A1

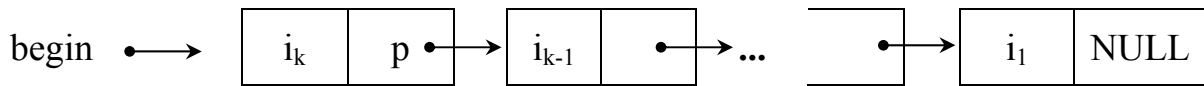


- 4) delete t;

А теперь давайте рассмотрим алгоритмы решения конкретного примера.

Задача. Сформировать стек, для простоты состоящий из целых положительных чисел (признаком окончания может быть отрицательное число, его в стек не заносим). Просмотреть стек, не изменяя вершины. Найти в стеке максимальный элемент и его порядковый номер, одновременно освобождая память, занятую просмотренным элементом.

В программе должна получиться следующая структура данных:



Декларируем структурный тип данных глобально:

```

struct Stack {
    int info;
    Stack *Next;
} ;
  
```

Формирование стека из k элементов

1. Декларируем объекты `int i, kol = 0;` где i – дополнительная переменная, kol – счетчик элементов;

`Stack *begin = NULL, (стек пуст) *t;`

2. Начало бесконечного цикла, например `while(1)`

а) считывание информации для текущего элемента (ввод i);

б) если $i < 0$ выход из цикла – `break`;

в) захват памяти под текущий элемент:

`t = new Stack;`

г) формирование информационной части

`t->info = i;`

д) значение вершины стека заносим в адресную часть текущего элемента (для первого элемента `NULL`):

`t->Next = begin;`

е) вершину стека переставляем на начало только что созданного элемента:

`begin = t;`

ж) счетчик элементов увеличиваем на единицу: `kol++;`

3. Конец цикла.

4. Вывод сообщения: «`Kol elementov = ...`».

Просмотр стека

1. Устанавливаем текущий указатель на вершину. Проверяем, не пуст ли стек. Если `begin=NULL`, то стек пуст (выдаем сообщение, и либо завершаем работу, либо переходим на его формирование).

2. Если стек не пуст начинаем цикл, выполняя до тех пор, пока `t != NULL`, т.е. не дошли до последнего.

а) выводим на печать информационную часть:

`printf("\n Элемент: %d", t->info);`

б) переставляем текущий указатель на следующий элемент:

t = t -> Next;

3. Конец цикла.

Поиск максимального элемента

1. Устанавливаем указатель текущего элемента на вершину:

t = begin;

(можно проверить, не пустой ли стек).

2. Положим, что максимальный – это элемент вершины:

int max = t -> info;

int nom_max = k; – номер элемента вершины, т.е. последнего.

3. Начало цикла, используем конструкцию do-while: do (выполнять)

а) begin = t -> Next; – переставляем вершину стека на следующий элемент, а указатель t остался на k-ом;

б) проверяем, если (t->info > max), заменяем max=t->info; и nom_max=k;

в) освобождаем память просмотренного элемента t : delete t;

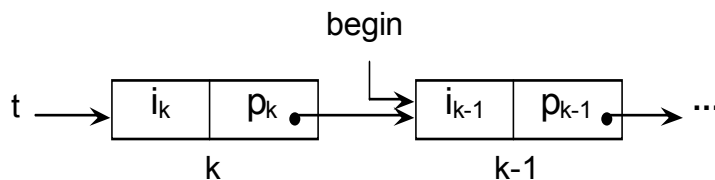
г) t = begin; теперь и вспомогательный указатель устанавливаем на следующий элемент и уменьшаем номер текущего элемента: k--;

4. Конец цикла, выполняющегося до тех пор пока t != NULL.

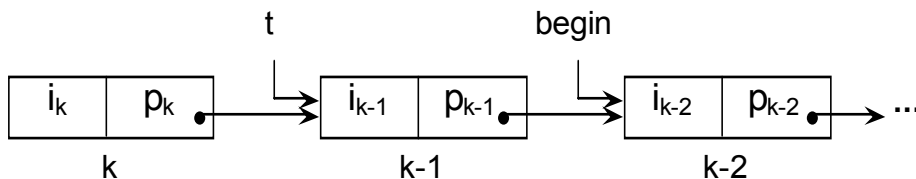
5. Вывод на экран найденных значений max и nom_max.

Т.е. для данной конструкции:

1-ый прогон:



2-ой прогон:



Завершится цикл тогда, когда p_1 станет равно NULL (на предпоследнем шаге $U_k = \text{NULL}$).

Текст программы может быть следующим:

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

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

```
struct Stack {  
    int info;  
    Stack *Next;
```

// Описание структурного типа Stack

```

    } ;
void main(void) {
Stack *begin = NULL, *t;
int i, kol = 0;
    while (1) { // Формирование стека
        printf(" Input info : ");
        scanf("%d", &i);
        if(i<0) break; // Выход из while(1)
        t = new Stack;
        t -> info = i;
        t -> Next = begin;
        begin = t;
        kol++;
    }
    printf("\n Kol elementov = %d", k);
//===== Просмотр стека =====
t = begin; // Установили текущий указатель на вершину if(t == NULL)
{
    printf("\n NO Steck!");
    return;
}
i = k;
while (t != NULL) {
    printf("\n Element %d: %d", i--, t -> info);
    t = t -> Next; // Текущий указатель на следующий элемент
}
//===== Поиск максимума =====
t = begin; // Установили текущий указатель на вершину
int max = t -> info;
int nom_max = k;
do {
    begin = t -> Next;
    if (t->info > max) {
        max = t->info;
        nom_max = k;
    }
    delete t; // Освобождаем память
    t = begin;
    k--;
} while (t != NULL);
printf("\n Max = %d , number = %d ", max, nom_max);
getch();
}

```

Пример использования стека для проверки правильности расстановки скобок в арифметическом выражении.

Скобки расставлены верно, если:

- а) число открывающихся и закрывающихся скобок совпадает;
- б) каждой открывающейся скобке соответствует закрывающаяся скобка.

Для выражения: $(A - (B + C) - D / (A + C) -$ нарушено первое условие, а для выражения: $(A - B) + C) - (D / (A + C) -$ нарушено второе условие.

Алгоритм анализа следующий:

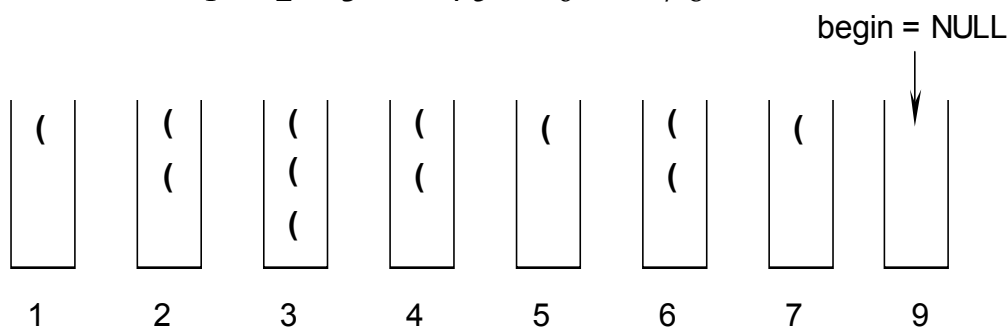
Просматриваем выражение слева направо.

1. Если обнаружена открывающая скобка, то записываем ее в стек.
2. Если первоначально обнаружена закрывающая скобка, то выражение составлено неверно, выводим сообщение об этом и завершаем работу.
3. Если теперь обнаружена открывающаяся скобка, то анализируем содержимое стека:
 - а) если стек пуст, то открывающая скобка отсутствует и выражение составлено неверно, выводим сообщение об этом и завершаем работу;
 - б) если стек не пуст, выбираем скобку – она открывающая, обе скобки выбрасываем из рассмотрения.
4. После того, как будет просмотрено все выражение, проверяем стек и, если он не пустой, то баланс скобок нарушен и выражение составлено неверно, выводим сообщение об этом и завершаем работу.

По такому принципу работают все компиляторы, проверяя арифметические выражения, вложенные блоки {...}, вложенные циклы и т.п.

Пример выражения:

$$(x + (y - (a + b)) * c - (d + e)) * k$$



5. Динамическая структура данных – ОЧЕРЕДЬ

Очередь – упорядоченный набор данных (структура данных), в котором в отличие от стека извлечение данных происходит из начала цепочки, а добавление данных – в конец этой цепочки.

Очередь также называют структурой данных, организованной по принципу **FIFO** (*First In, First Out*) – первый вошел (первый созданный элемент очереди), первый вышел.

Количество элементов очереди, как и в случае стека не лимитируется – память должна запрашиваться динамически по мере того, как в очередь добавляются новые элементы, и так же динамически освобождаться, когда из нее удаляются элементы.

Пример очереди – некоторый механизм обслуживания, который может выполнять заказы только последовательно один за другим. Если при поступлении нового заказа данное устройство свободно, оно немедленно приступит к выполнению этого заказа, если же оно выполняет какой-то ранее полученный заказ, то новый заказ поступает в конец очереди из других ранее пришедших заказов. Когда устройство освобождается, оно приступает к выполнению заказа из начала очереди, т.е. этот заказ удаляется из очереди и первым в ней становится следующий за ним заказ.

Заказы, как правило, поступают нерегулярно и очередь то увеличивается, то укорачивается и даже может оказаться пустой.

Работу с очередью можно организовать двумя способами.

1. Использовать помимо рабочего указателя для начала и окончания очереди дополнительные переменные-указатели, т.е. первый указатель установить на начало очереди, а второй – на ее конец.

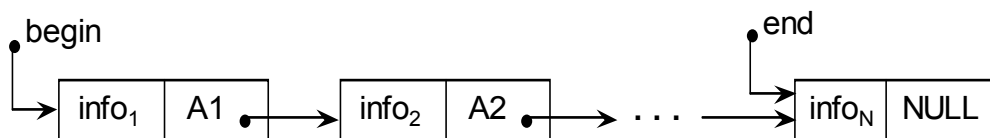
2. Использовать указатель-переменную только для начала очереди, а перемещаться по ней на основе внутренних связей между ее элементами.

Основные операции с очередью следующие:

- первоначальное формирование очереди;
- добавление нового элемента в конец очереди;
- удаление элемента из начала очереди.

В языке Си работа с очередью, как и со стеком, реализуется при помощи структур, указателей на структуры и операций динамического выделения и освобождения памяти.

Далее мы организуем структуру данных следующего вида:



где каждый элемент очереди имеет информационную *info* и адресную *Next* (A1, A2 ...) части.

Шаблон элемента структуры (аналогично стеку) может иметь такой вид:

```
struct Spis {
```



```

int info;
Spis *Next;
    } *begin, *end;

```

begin, end – указатели на начало и конец очереди.

Формирование очереди

Первоначальное формирование очереди осуществляется в два этапа.

Этап 1. Первоначальное формирование очереди

Этот этап заключается в создании первого элемента, для которого адресная часть должна быть нулевой (*NULL*). Для этого нужно:

1) ввести информацию для первого элемента (для простоты – это какое-то целое положительное число – значение переменной *i*);

2) захватить, используя рабочий указатель память под первый элемент:

```
t = new Spis;
```

сформирован конкретный адрес (*A1*) для первого элемента;

3) в информационную часть заносим введенную с клавиатуры информацию:

```
t -> info = i; ( обозначим ее как i1 )
```

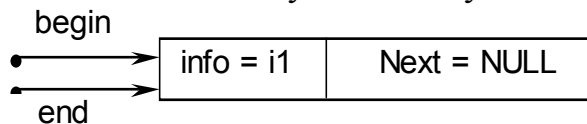
4) в адресную часть заносим *NULL*:

```
t -> Next = NULL;
```

5) указателям на начало и на конец очереди присваиваем значение *t*:

```
begin = end = t;
```

На этом этапе получили следующее:



Этап 2. Формирование элементов очереди, начиная с первого

Рассмотрим только для второго элемента, а ниже обобщим для произвольного числа элементов.

1. Начало бесконечного цикла *while* (1), окончанием которого будет отрицательное число.

2. Ввод информации для текущего элемента – значение *i*.

3. Если значение не подходит, т.е. *i* < 0 – завершение цикла (*break*).

4. Иначе захватываем память под текущий элемент:

```
t = new Spis;
```

5. Формируем информационную часть (для второго элемента введенное значение обозначим *i2*):

```
t -> info=i;
```

6. В адресную часть созданного элемента (текущего) – *NULL*, т.к. этот элемент теперь последний:

```
t -> Next = NULL;
```

7. Элемент добавляется в конец очереди, поэтому в адресную часть последнего элемента $end \rightarrow Next$ вместо $NULL$ заносим адрес созданного:

$end \rightarrow Next = t$;

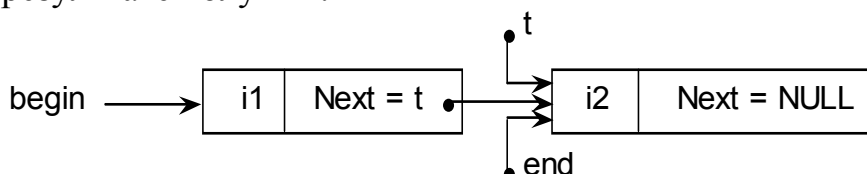
бывший последний элемент теперь стал предпоследним.

8. Переставляем указатель последнего элемента на добавленный:

$end = t$;

9. Конец цикла

В результате получим:



Алгоритм добавления нового элемента

1. Захватываем память под новый элемент структуры.

2. Устанавливаем на него через указатель конца очереди указатель теперь уже предпоследнего элемента:

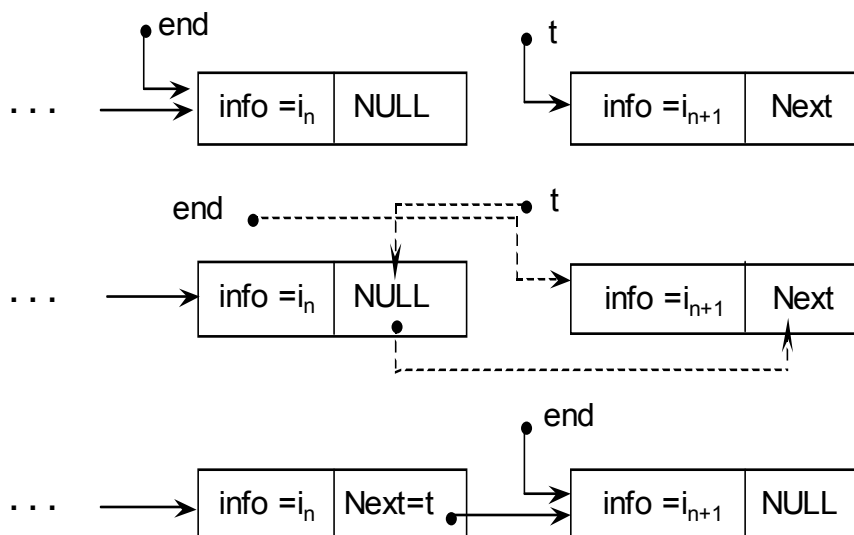
$end \rightarrow Next = t$;

3. Считываем данные и заносим в информационную часть.

4. Записываем в адресную часть $NULL$.

5. Устанавливаем на новый элемент указатель конца очереди.

Графически это выглядит так:



Алгоритм удаления первого элемента из очереди

Имеем непустую очередь (можно организовать проверку указателей на $NULL$).

1. Устанавливаем текущий указатель на начало очереди: $t = begin$;

2. Обрабатываем информационную часть первого элемента очереди, например, печатаем.

3. Указатель на начало очереди переставляем на следующий (2-ой) элемент
`begin = begin->Next;`

4. Освобождаем память, захваченную под 1-ый элемент: `delete t;`

5. Выводим сообщение, например, «Элемент удален!».

Алгоритм просмотра очереди

1. Устанавливаем текущий указатель на начало очереди: `t = begin;`

2. Начало цикла, выполняющегося пока $t \neq NULL$ (работает, пока не дойдем до конца очереди).

3. Информационную часть текущего элемента $t->info$ выводим на печать.

4. Переставляем текущий указатель на следующий элемент очереди:

`t = t->Next;`

5. Конец цикла.

Алгоритм установки указателя на нужный элемент очереди

1. Ввести номер нужного элемента k .

2. Устанавливаем текущий указатель на начало очереди: `t = begin;`

3. Начало цикла: перебор элементов от $i=1$ до $i < k$ с шагом 1.

4. Переставляем текущий указатель на следующий элемент очереди:

`t = t->Next;`

5. Конец цикла.

В итоге t будет указывать на начало k -го элемента очереди.

Пример. Создать очередь из целых положительных чисел, организовать просмотр, добавление элементов в конец и удаление элементов из начала очереди.

Текст программы может иметь вид:

```
#include <stdio.h>
#include <conio.h>
struct Spis {                      // Описание структуры
    int info;
    Spis *Next;
} *begin, *end;
void Make(int);                   // Описание прототипов функций
void Del();                       // Удаление одного первого элемента
void Del_All();                   // Освобождение всей памяти
void View();
void main(void) {
    Spis *t;
    int i;
    while(1) {
        puts("\n Creat - 1\n View - 2\n Add - 3\n Del - 4\n EXIT - 0");
```

```

switch (getch()) {
    case '1': Make(0); break; // kod=0 – Первоначальное формирование
    case '2': View(); break;
    case '3': if(begin == NULL) Make(0);
                else Make(1); // kod=1 – Добавление
    break;
    case '4': Del(); break;
    case '0': Del_All();return;
} // switch()
} // while(1)
}
//-----Создание очереди (kod=0) и Добавление в конец (kod=1)-----
void Make(int kod) {
    Spis *t;
    int i;
    if ( kod==0 ) { // Формирование первого элемента
        printf(" Input info 1: ");
        scanf("%d", &i);
        t = new Spis;
        t -> info = i;
        t -> Next = NULL;
        begin = end = t;
    }
    while (2) { // Добавление положительных элементов
        printf(" Input info : ( >= 0 ) ");
        scanf("%d", &i);
        if(i<0) break; // Выход из цикла while(2)
        t = new Spis;
        t -> info = i;
        t -> Next = NULL;
        end -> Next = t;
        end = t;
    }
}
//----- Просмотр очереди -----
void View() {
    Spis *t = begin;
    puts(" Ochered ");
    if(t == NULL) {
        printf("\n NO !");
        return;
    }
    while (t != NULL) {
        printf("\n Element %d ", t -> info);
        t = t -> Next;
    }
}

```

```

    }
}

//----- Удаление элемента (одного) -----
void Del() {
    Spis *t = begin;
    if(t==NULL) {
        printf("\n NO !");
        return;
    }
    printf("\n Info del = %d ",begin -> info);
    begin = begin -> Next;
    delete t;
    printf("\n Element Deleted!");
}

//----- Удаление всех элементов очереди -----
void Del_All() {
    Spis *t;
    while (begin != NULL) {
        t = begin;
        begin = begin -> Next;
        delete t;
    }
    printf("\n Deleted!");
}

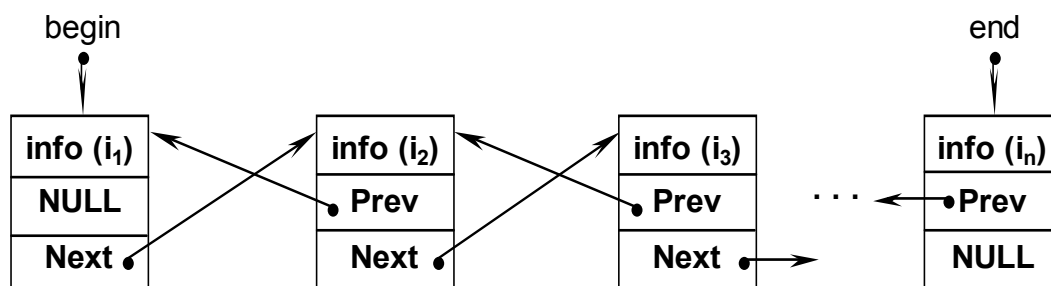
```

6. Двухнаправленный линейный список

Мы познакомились с довольно простой списочной структурой данных односторонним (односвязным) списком. Рассмотрим теперь более универсальный список – двухнаправленный (двухсвязный).

Это список, каждый элемент которого кроме первого и последнего имеет еще два указателя: на предыдущий и следующий элементы.

Графически это выглядит следующим образом:



Можно, кроме того, поместить впереди списка дополнительный элемент, который называется «заголовок списка». В нем хранят информацию обо всем

списке, в частности, размещают счетчик числа элементов в списке. Наличие заголовка может, как усложнить программу, так и упростить ее, в зависимости от поставленной задачи.

Для работы со списком достаточно одного указателя на его начало *begin*, но для упрощения введем и указатель на конец списка – *end*.

Введем структуру, в которой (для простоты, как и раньше) информационной частью *info* будут целые числа:

```
struct Spis {
    int info;
    Spis *Prev, *Next;
} *begin, *end;
```

begin, *end* – указатели на начало и конец списка, соответственно.

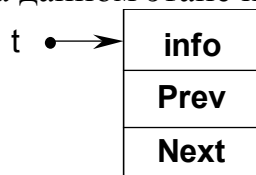
Формирование такого списка проводится в два этапа.

Этап I. Формирование первого элемента

1. Захватить память на текущий элемент:

```
Spis *t = new Spis;
```

На данном этапе имеем элемент:



2. Формируем первый элемент списка:

а) в информационную часть вводим значение (с клавиатуры):

```
scanf("%d", &t->info);
```

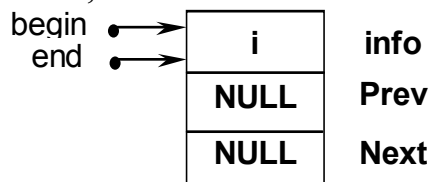
б) в адресные части заносим NULL:

```
t->Prev = t->Next = NULL;
```

в) указатели на начало и конец списка устанавливаем на 1-ый элемент:

```
begin = end = t;
```

После выполнения указанных действий получили элемент, указатели которого в NULL, т.е. список состоит из одного элемента:



Этап II. Формирование списка добавлением в конец

1. Начало цикла.

2. Захват памяти под текущий элемент:

```
t = new Spis;
```

3. Вводим в информационную часть значение:

```
scanf("%d", &t->info);
```

4. В адресную часть указателя на следующий элемент (Next) заносим NULL:

t -> Next = NULL;

5. В адресную часть указателя на предыдущий элемент (Prev) через указатель конца (end) заносим адрес предыдущего элемента

t -> Prev = end;

6. В адресную часть предыдущего элемента, т.е. последнего, через end заносим адрес только что созданного (текущего):

end -> Next = t;

7. Указатель конца списка устанавливаем на созданный элемент

end = t;

8. Продолжаем с п. 2, до тех пор пока не обработаем признак завершения.

Алгоритм просмотра списка

С начала

1. Устанавливаем текущий указатель на

начало списка:

t = begin;

2. Начало цикла, работающего, до тех пор пока t != NULL.

3. Информационную часть текущего элемента t -> info – на печать.

4. Устанавливаем текущий указатель на

следующий элемент, адрес которого находится в поле Next текущего элемента

t = t -> Next;

5. Конец цикла.

С конца

конец списка:

t = end;

предыдущий элемент, адрес которого находится в поле Prev текущего элемента

t = t -> Prev;

Алгоритм поиска элемента в списке по ключу

Ключом может быть любое интересующее значение (в зависимости от поставленной задачи). Поэтому, уточним задачу: найдем конкретное значение info в списке и его порядковый номер.

1. Введем с клавиатуры ключ поиска, т.е. искомое значение i_p

2. Установим текущий указатель на начало списка

t = begin;

3. Счетчик элементов k=1;

4. Начало цикла (выполнять пока t != NULL, т.е. не дойдем до конца)

5. Сравниваем информационную часть текущего элемента с искомым:

5.1) если они совпадают (t -> info = i_p), выводим на печать элемент, его номер k и завершаем поиск (break);

5.2) иначе, переставляем текущий указатель на следующий элемент и увеличиваем счетчик на 1:

t = t -> Next;

k++;

6) Конец цикла.

В общем случае, не плохо было бы оформить случай, когда такого элемента в списке нет. В примере программы это предусмотрено.

Алгоритм удаления элемента в списке по ключу

Найти в списке и удалить элемент, значение информационной части (ключ) которого совпадает с введенным с клавиатуры.

Решение данной задачи проводим в два этапа: поиск, удаление.

Изменим алгоритм поиска, т.к. в дальнейшем понадобится дополнительный указатель для удаления и добавим контроль на случай отсутствия в списке искомого элемента.

Этап 1. Поиск

1. Установим дополнительный указатель в:

Spis *key = NULL;

2. Введем с клавиатуры искомое значение (ключ поиска): i_p

3. Установим текущий указатель на начало списка:

t = begin;

4. Начало цикла (выполнять пока t != NULL).

5. Сравниваем информационную часть текущего элемента с искомым:

1) если они совпадают (t -> info = i_p), то, например, выводим сообщение об успехе;

2) запоминаем адрес найденного элемента:

key = t;

3) завершаем поиск – досрочный выход из цикла (break);

4) иначе, переставляем текущий указатель на следующий элемент:

t = t -> Next;

6. Конец цикла.

7. *Контроль*, если key = NULL, т.е. искомый элемент не найден, то сообщаем о неудаче и функцию удаления не выполняем (return или exit).

Этап 2. Удаление

1. Иначе, т.е. key != NULL – продолжаем алгоритм. Удаление элемента из списка проводится в зависимости от его местонахождения.

2. Проверяем, находится ли удаляемый элемент в начале списка, т.е. если key=begin, то создадим новый начальный элемент:

1) указатель на начало списка переставляем на следующий за первым (второй) элемент:

begin = begin -> Next;

2) а в поле Prev элемента, который был вторым, а теперь стал первым, заносим NULL, т.е. нет предыдущего

begin -> Prev = NULL;

3. Иначе, если удаляемый элемент в конце списка, т.е. key = end, то

1) указатель на конец списка переставляем на предыдущий элемент, адрес которого в поле Prev последнего:

2) обнулить для нового последнего элемента, т.е. предпоследнего указатель на следующий элемент в поле Next
end -> Next = NULL;

Адреса элементов:

key->Prev

key

key->Next

...

info

Prev

Next

...

Порядковые номера:

k-1

k

k+1

```
( key -> Prev ) -> Next = key -> Next;
```

```
( key -> Next ) -> Prev = key -> Prev;
```

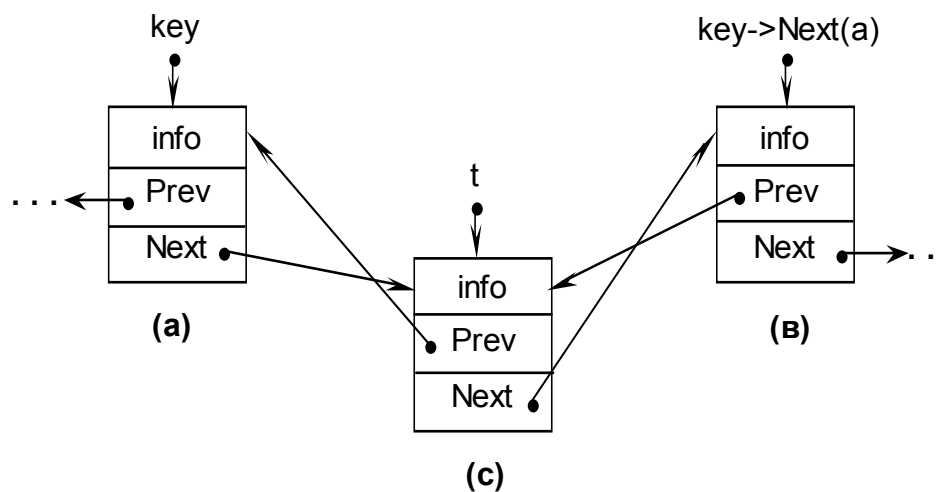
Решение данной задачи проводим в два этапа: поиск, вставка.

Используем алгоритм поиска, рассмотренный ранее, т.е. этап 1 – пропускаем (для удобства, поиск оформить в виде функции), предполагаем, что указатель на искомый элемент (`key != NULL`) – найден.

129

3. Связываем новый элемент с предыдущим
 $t \rightarrow \text{Prev} = \text{key};$
 4. Связываем новый элемент со следующим
 $t \rightarrow \text{Next} = \text{key} \rightarrow \text{Next};$
 5. Связываем предыдущий элемент с новым
 $\text{key} \rightarrow \text{Next} = t;$
 6. Если элемент добавляется не в конец списка, т.е. $\text{key} \neq \text{end}$, то
 $(t \rightarrow \text{Next}) \rightarrow \text{Prev} = t;$
- по адресу (в), который в поле Next(c) $[(t \rightarrow \text{Next})] - \text{в поле Prev(в)} [(t \rightarrow \text{Next}) \rightarrow \text{Prev}]$ заносим адрес (с);
7. Иначе $\text{end} = t;$

Схема вставки элемента (с):



Пример: написать программу обработки двунаправленного списка из целых чисел. Реализовать функции: создание, просмотр, поиск введенного с клавиатуры значения и его номера, удаление элемента, вставку элемента по номеру (после), освобождение памяти при выходе из программы.

Текст программы может иметь вид:

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

struct Spis {
    int info;
    Spis *Prev,*Next;
} *begin, *end;

void View (void );
void Del_Spis(void);

void main(void) {
    Spis *t, *key;
    int i, k;
    while(1) {
        printf("\n Creat - 1\n View - 2\n Find - 3\
        // Описание структуры
```

```

        \n Del  - 4\n Ins  - 5\n EXIT  - 0\n");
switch (getch()) {
//===== Формирование первого элемента =====
    case '1': i=0;
        t = new Spis;
        printf(" Input info 1 : "); scanf("%d", &t->info);
        t->Next = t->Prev = NULL;
        begin = end = t;
//===== Формирование списка со второго =====
    while (2) {
        t = new Spis;
        printf(" Input info : "); scanf("%d", &t->info);
        t->Next = NULL;
        t->Prev = end;
        end -> Next = t;
        end = t;
        printf("\n Repeat - y ");
        if(getch() != 'y') break; // Продолжение - 'y', иначе - выход
    }
    break;
//===== Просмотр списка =====
    case '2': View ( ); break;
//===== Поиск элемента =====
    case '3': t = begin;
        printf("\n Find Info : "); scanf("%d", &i);
        k=1;
        while ( t != NULL) {
            if (t->info == i) {
                printf("\n Element = %d - number = %d", t->info, k);
                getch(); break;
            }
            t = t->Next;
            k++;
        }
        if( t == NULL ) {
            puts("\n Not Found !");
            getch();
        }
        break;
//===== Поиск элемента для удаления =====
    case '4': t = begin;
        key = NULL;
        k = 1;
        printf( "\n Delet Info : "); scanf("%d", &i);
        while ( t != NULL) {

```

```

    if (t->info == i) {
        printf("\n Delete element = %d - number = %d", t->info, k);
        getch();
        key = t;           // Запомнили адрес удаляемого элемента
        break;
    }
    t = t->Next;
    k++;
}
if( key == NULL ) {
    puts("\n NO Element !");
    getch();
    break;
}

//===== Удаляем найденный =====
if ( key == begin) {
    begin = begin->Next;
    begin->Prev = NULL;
}
else if ( key == end) {
    end = end->Prev;
    end->Next = NULL;
}
else {
    ( key-> Prev) -> Next = key->Next;
    ( key-> Next) -> Prev = key->Prev;
}

delete key;
break;

//===== Вставка элемента по порядковому номеру (после) =====
case '5':    t = begin;
    printf( "\n Ins Nom : "); scanf("%d", &k);
    for ( i=1; i<k; i++)
        if ( t != NULL) t = t->Next;
    key = t;
    if ( key == NULL) {
        printf( "\n NO element ");
        getch(); break;
    }
    t = new Spis;
    printf( "\n Input info "); scanf("%d", &t -> info);
    t -> Prev = key;
    t -> Next = key -> Next;
    key -> Next = t;
    if ( key != end ) ( t -> Next ) -> Prev = t;

```

```

        else end = t;
    break;
    case '6': Sort(); break;
    case '0': Del_Spis(); return;
} // switch()
} // while(1)
} // main()

//===== Просмотр списка с начала =====
void View ( )      {
    int k=1;
    Spis *t = begin;
    if ( t== NULL ) {
        printf ("\n NO elements !");    return;
    }
    while (t != NULL) {
        printf("\n %d element - %d", k, t->info);
        t = t->Next;
        k++;
    }
}

//===== Освобождение памяти =====
void Del_Spis(void) {
    Spis *t = begin;
    while( t!=NULL) {
        begin = begin -> Next;
        delete t;
        t = begin;
    }
    puts(" Delete!");
}

```

Алгоритм вставки в отсортированный по возрастанию список

Этот алгоритм можно использовать для формирования уже отсортированного по выбранному (ключевому) полю списка. При вставке элемента возможны три случая: в начало, в конец и в середину списка.

1. Установим текущий указатель на начало:
t = begin;
2. Введя дополнительный указатель, захватим память под новый элемент
Spis *t1 = new Spis;
3. Формируем информационную часть (с клавиатуры):
scanf("%d", & t1 -> info);
4. Начало цикла (выполнять до тех пор, пока t !=NULL) поиска места для вставки.

5. Если $t1 \rightarrow \text{info} < t \rightarrow \text{info}$, то вставляем новый элемент перед текущим:

1) в адресную часть Next нового элемента заносим адрес текущего
 $t1 \rightarrow \text{Next} = t$;

2) если это начало списка, т.е. $t1 = \text{begin}$, то

$t1 \rightarrow \text{Prev} = \text{NULL}$;
 $\text{begin} = t1$;

3) иначе, вставляем новый элемент в середину списка

а) $(t \rightarrow \text{Prev}) \rightarrow \text{Next} = t1$; – предыдущий в поле Next получает адрес нового;

б) $t1 \rightarrow \text{Prev} = t \rightarrow \text{Prev}$;

в) $t1 \rightarrow \text{Next} = t$;

г) $t \rightarrow \text{Prev} = t1$;

4) Сообщив об успехе, выходим из функции, (return;) – это необходимо сделать, чтобы обеспечить вставку элемента, если он окажется больше последнего.

6. Если $t1 \rightarrow \text{info} > t \rightarrow \text{info}$, то идем дальше, переставляя текущий указатель на следующий элемент

$t = t \rightarrow \text{Next}$;

7. Конец цикла.

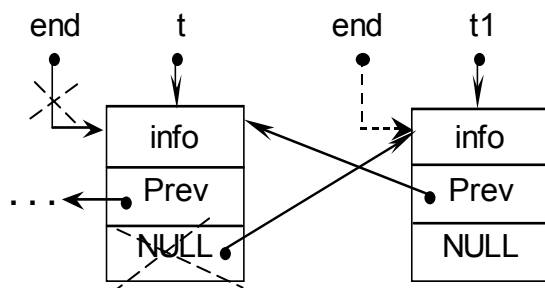
8. Этот пункт будет выполняться только в том случае, если вставляемый элемент больше последнего, т.е. приписываем его в конец, изменяя последний:

1) $t1 \rightarrow \text{Next} = \text{NULL}$;

2) $t1 \rightarrow \text{Prev} = \text{end}$;

3) $\text{end} \rightarrow \text{Next} = t1$;

4) $\text{end} = t1$;



// ===== Функция Sort =====

```
void Sort() {
    Spis *t = begin, *t1 = new Spis;
    printf( "\n Input info ");
    scanf("%d", &t1 -> info);           // Добавляем новый элемент
    while ( t != NULL ) {
        if ( t1 -> info < t -> info ) { // Поставить перед текущим
            t1 -> Next = t;
            if( t == begin ) { // В начало списка
                t1 -> Prev = NULL;
                begin = t1;
            }
            else { // В середину списка
```

```

        ( t -> Prev ) -> Next = t1;
        t1 -> Prev = t -> Prev;
    }

    t -> Prev = t1;
    puts("\n Insert! "); // Выводим сообщение
    return;
}
t = t -> Next;
}
t1 -> Next = NULL;           // В конец списка
t1 -> Prev = end;
end -> Next = t1;
end = t1;
puts("\n Insert End! ");    // Сообщение - вставили в конец
return;
}

```

7. Построение обратной польской записи

Одной из главных причин, лежащих в основе появления языков программирования высокого уровня, явились вычислительные задачи, требующие больших объемов вычислений. Поэтому к разработчикам языков программирования были предъявлены жесткие требования: максимальное приближение формы записи в коде программы математических выражений к естественному языку математики. Поэтому одной из первых областей системного программирования составили исследования способов трансляции математических выражений. И здесь были получены многочисленные результаты данных исследований, но наибольшее распространение получил метод трансляции при помощи обратной польской записи, которую предложил польский математик Я.Лукашевич. Ниже приводятся преимущества такого промежуточного постфиксного представления математического выражения. Но прежде давайте рассмотрим два основных алгоритма получения обратной польской записи.

Алгоритм 1. Данный алгоритм основан на представлении математического выражения в виде дерева и использовании третьего способа его обхода, который подробно рассмотрен в [1].

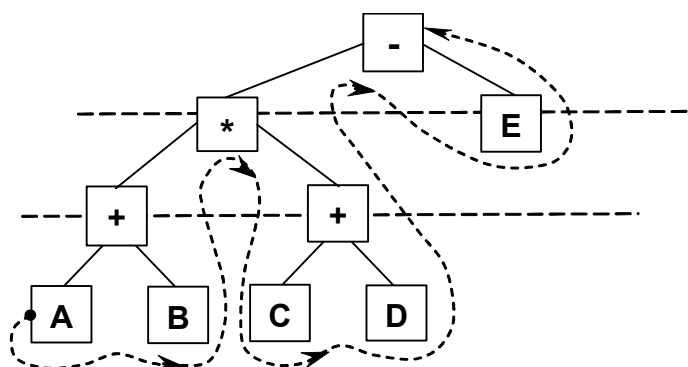
Напомним его на конкретном примере.

Пусть задано простое арифметическое выражение вида:

$$(A+B)*(C+D)-E . \quad (1)$$

Представим это выражение в виде дерева, в котором узлам соответствуют операции, а листьям – операнды. Построение начинается с корня, в качестве которого выбирается операция, которая выполняется последней.левой ветви соответ-

ствует левый операнд операции, а правой ветви – правый. Дерево выражения (1) имеет вид:



Совершим обход дерева, под которым понимается формирование строки из символов узлов и ветвей дерева. Обход будем совершать от самой левой ветви вправо и узел переписывать в выходную строку только после рассмотрения всех его ветвей. Обход совершаем строго по уровням.

Уровень 1: имеем **AB+CD+**

Поднялись на Уровень 2: имеем ***E**

И наконец Корень.

Результат обхода дерева таким образом имеет вид:

$$\mathbf{AB+CD+*E-} \quad (2)$$

Характерные особенности выражения (2) состоят в следовании символов операций за символами операндов и в отсутствии скобок. Такая запись и называется обратной польской записью.

Алгоритм 2. Использование стека. Получение обратной польской записи из исходного выражения может осуществляться весьма просто на основе простого алгоритма, предложенного Дейкстрой. Для этого он ввел понятие стекового приоритета операций, который приведен в таблице:

Операция	Приоритет
(0
)	1
+ -	2
* /	3

Суть алгоритма в следующем. Просматривается исходная строка символов слева направо, операнды переписываются в выходную строку, а знаки операций заносятся в стек на основе следующих соображений:

- если стек пуст, то операция из исходной строки переписывается в стек;
- операция выталкивает из стека все операции с большим или равным приоритетом в выходную строку;

в) если очередной символ из исходной строки есть открывающая скобка, то он проталкивается в стек;

г) закрывающая круглая скобка выталкивает все операции из стека до ближайшей открывающей скобки, сами скобки в выходную строку не переписываются, а уничтожают друг друга.

Процесс получения обратной польской записи выражения (1) представлен в таблице:

Просматриваемый символ	Входная строка	Состояние стека			Выходная строка
1	((
2	A	(A
3	+	+	(
4	B	+	(B
5)				+
6	*	*			
7	((*		
8	C	(*		C
9	+	+	(*	
10	D	+	(*	D
11)	*			+
12	–	–			*
13	E	–			E
14					–

Получим: **AB+CD+*E–**

Обратная польская запись обладает рядом замечательных свойств, которые превращают ее в идеальный промежуточный язык при трансляции.

Во-первых, вычисление выражения, записанного в обратной польской записи, может проводиться путем однократного просмотра, что является весьма удобным при генерации объектного кода программ.

Например, вычисление полученного выражения может быть проведено следующим образом:

Шаг	Анализируемая строка	Действие
1	AB+CD+*E–	R1=A+B
2	R1CD+*E–	R2=C+D
3	R1R2*E–	R1=R1*R2
4	R1E–	R1=R1–E
5	R1	

Здесь R1 и R2 – вспомогательные переменные.

А теперь несколько примеров на использование только что рассмотренных алгоритмов.

Пример 1. Пусть опять задано математическое выражение $a+b*c+(d*e+f)*g$. Необходимо записать это выражение в постфиксной форме. Правильным ответом будет выражение $abc*+de*f+g*+$. Решаем эту задачу, используя стек.

Пусть исходная информация хранится в строке $S = "a+b*c+(d*e+f)*g"$. Результат будем получать в строке V .

Алгоритм решения

В начале стек пуст, V – пустая строка, $S = "a+b*c+(d*e+f)*g"$.

Первым читается символ «а», т.к. он не является операцией, то помещается в строку V . Затем читаем «+» и помещаем в стек. Следующий символ «b» помещаем в строку V . На этот момент стек и строка V выглядят следующим образом:

+

 $V = "ab"$

Следующий символ «*». Элемент в вершине стека имеет более низкий приоритет, чем «*», поэтому в строку V ничего не помещаем, а «*» помещаем в стек. Далее идет символ «с», он помещается в строку V . После этого имеем:

*
+

 $V = "abc"$

Следующий символ в строке S «+». Анализируем стек и видим, что элемент в вершине стека «*» имеет более высокий приоритет, чем «+». Поэтому извлекаем этот элемент из стека и помещаем в строку V .

Следующий элемент стека «+» имеет тот же приоритет, что и текущий символ строки S . Следовательно, извлекаем элемент из стека и помещаем его в строку V , а прочитанный из строки S элемент помещаем в стек. В итоге имеем:

+

 $V = "abc*+"$

В строке S следующий символ «(». Он имеет самый высокий приоритет и помещается в стек. Читаем символ «d» и помещаем в строку V .

(
+

 $V = "abc*+d"$

Читаем строку S дальше – это символ «*». Так как открывающую скобку нельзя извлечь из стека до тех пор, пока не встретилась закрывающая, то в строку V ничего не помещаем, а знак «*» помещаем в стек. Следующий символ строки S «е» помещаем в строку V .

	B = "abc*+de"
*	
(
+	

Следующий прочитанный символ «+». Так как элемент «*» имеет более высокий приоритет, то извлекаем его из стека и помещаем в строку В. Символ «+» помещаем в стек, а следующий символ строки S «f» помещаем в строку В.

	B = "abc*+de*f"
+	
(
+	

Далее в строке S идет закрывающая скобка. В этом случае все элементы стека до символа «)» включительно помещаем в строку В (это элемент «+»). Символ «(» просто игнорируется после того, как был извлечен из стека, и читается очередной символ входной строки S.

	B = "abc*+de*f+"
+	

Читаем символ «*», помещаем его в стек, а следующий за ним символ «g» – в строку В.

	B = "abc*+de*f+g"
*	
+	

Закончив чтение символов строки S, переписываем элементы из стека в строку В.

	B = "abc*+de*f+g*+"

Таким образом, просмотрев исходную информацию только один раз, мы решили поставленную задачу.

Пример 2. Пусть задано выражение, записанное в постфиксной форме $A = "6523+8*+3*"$, операндами которого являются цифры. Вычислить его значение.

Решение. В постфиксной записи каждая операция относится к двум предшествующим операндам. Причем один из этих операндов может быть результатом операции, выполненной ранее.

Записываем в стек каждый встретившийся операнд.

Если встречается операция, то ее операндами будут два верхних элемента стека.

Извлекаем эти элементы, выполняем над ними текущую операцию, и результат помещаем в стек.

Выполнение алгоритма на примере

В начале организуем пустой стек.

Первые четыре символа выражения – операнды, читаем их и помещаем в стек.

3
2
5
6

Следующий элемент «+». Извлекаем из стека соответствующие операнды («3» и «2»), выполняем действие $3+2$, результат «5» помещаем в стек.

5
5
6

Далее в стек помещаем очередной операнд входной строки «8».

8
5
5
6

Теперь встретился знак операции «*». Извлекаем из стека для него операнды «8» и «5», выполняем умножение $8*5$, результат «40» помещаем в стек.

40
5
6

Следующим символом идет «+». Извлекаем из стека «40» и «5», выполняем сложение операндов $40+5$, результат «45» помещаем в стек.

45
6

Читаем и помещаем в стек очередной операнд входной строки «3».

3
45
6

Следующий символ строки А «+». Извлекаем из стека операнды для этой операции («3» и «45») и выполняем сложение $3+45$, результат «48» помещаем в стек.

48
6

Последний символ в строке А «*». Операнды для этой операции – «48» и «6». В стек помещаем результат выполнения этой операции – $48*6=288$, это и есть значение заданного выражения.

Пример реализации алгоритма получения обратной польской записи на основании приоритета операций, который предложен Дейкстрой.

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

```

#include<stdlib.h>
#include<conio.h>
#include<string.h>
struct Stack {
    char c;
    Stack *Next;
};
int Prior (char);
void main () {
    Stack *t;
    char a, In[50], Out[50];           // Входная и выходная строки
    Stack *Op = NULL;                 // Стек операций Op – пуст
    int k, l;                          // Текущие индексы для строк
    puts(" Input formula : ");
    gets(In);
    k=l=0;
    while ( In[k] != '\0') {
//===== Часть 1 =====
        if ( In[k] == ')' ) { // Если очередной символ ")", выталкиваем из
                                // стека все знаки операций в выходную строку
            while ( (Op -> c) != '(' ) { // до первой открывающей скобки "("
                t = Op;                // Заносим в него вершину стека
                a = t -> c;             // В а записываем значение элемента вершины
                Op = t -> Next;         // В вершину заносим адресную часть
                delete t;               // Сняли указатель с элемента (извлекли)
                if ( !(Op) ) a = '\0';
                Out[l++] = a;           // Информационную часть – в выходную строку
            }
            t = Op;                    // Удаляем из стека открывающуюся скобку
            Op = t -> Next;
            if ( Op == NULL ) {        // Стек пустой
                puts (" End of Stack!");
                break;
            }
            delete t;
        }
//===== Часть 2 =====
        if ( (In[k] >= 'a') && (In[k] <= 'z')) // Если символ буква,
            Out[l++] = In[k];               // то заносим ее в выходную строку
//===== Часть 3 =====
        if ( In[k] == '(' ) { // Если символ входной строки открывающаяся
                                // скобка, то заталкиваем ее в стек
            t = new Stack;
            t -> c = In[k];
            t -> Next = Op;

```

```

    Op = t;
}
//===== Часть 4 =====
    if ( In[k] == '+' || In[k] == '-' || In[k] == '*' || In[k] == '/') {
// Если знак операции, то переписываем в выходную строку все операции,
    while ( (Op !=NULL) && (Prior (Op -> c) >= Prior (In[k]))) {
// находящиеся в стеке с большим или равным приоритетом
        t = Op;
            a = t -> c;
            Op = t -> Next;
            delete t;                // Извлекли элемент из стека
            // if ( !(Op) ) a = '\0';
            Out[l++] = a;
        }
        t = new Stack;           // Записываем в стек поступившую информацию
        t -> c = In[k];
        t -> Next = Op;
        Op = t;
    }
    k++;
}                                // Конец цикла анализа входной строки
//===== Часть 5 =====
while ( Op !=NULL) {           // После рассмотрения всей входной информации
    t = Op;                     // переписываем операции из стека
    a = t -> c;                 // в выходную строку
    Op = t -> Next;
    delete t;
    Out[l++] = a;
}
Out[l] = '\0';
printf("\n Polish = %s", Out); // Печатаем полученную строку
getch();
}
//===== Функция реализации приоритета операций =====
int Prior ( char a ) {
    switch ( a ) {
        case '*': case '/': return 3;
        case '-': case '+': return 2;
        case '(': return 1;
    }
    return 0;
}

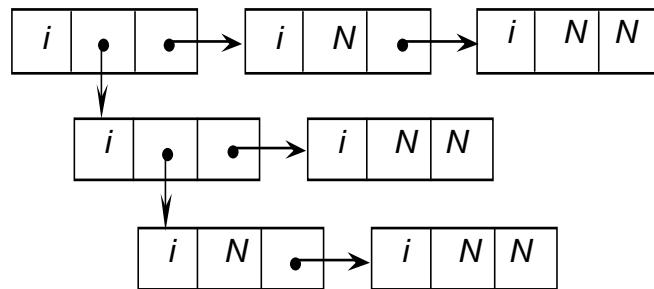
```

8. Нелинейные структуры данных

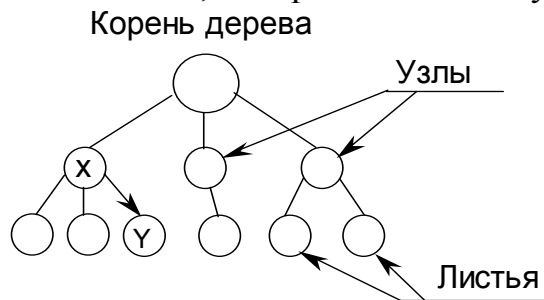
Мы рассмотрели связные структуры данных или списки – это линейные структуры динамических списковых данных.

Введение в динамическую переменную двух и более «полей-указателей» позволяет получить нелинейные структуры данных. Наиболее распространенные:

а) «текст» – переменная с тремя полями, «первое поле» – информация (i), «второе» – либо указывает на первый элемент следующей строки, либо имеет значение NULL (N), «третье» либо на следующий элемент текущей строки, либо «обнулено»:



б) довольно часто при работе с данными бывает удобно использовать структуры с иерархическим представлением, изображенные следующим образом:



Такая конструкция данных получила название «дерево».

Дерево состоит из элементов, называемых узлами (вершинами). Узлы соединены между собой направленными дугами. В случае $X \rightarrow Y$ вершина X называется предком (родителем), а Y – потомком (сыном).

Дерево имеет единственный узел, у которого нет предков. Такой узел называют **корнем**. Любой другой узел имеет ровно одного предка. Узел, не имеющий потомков, называется **листом**.

Внутренний узел – это узел, не являющийся ни листом ни корнем. **Порядок узла** – количество его узлов-потомков. **Степень дерева** – максимальный порядок его узлов. **Глубина узла** равна числу его предков плюс один. **Глубина дерева** – это наибольшая глубина его узлов.

Бинарные деревья

Бинарное дерево – это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на бинарные

Начальный же узел, как уже известно, называется корнем дерева.

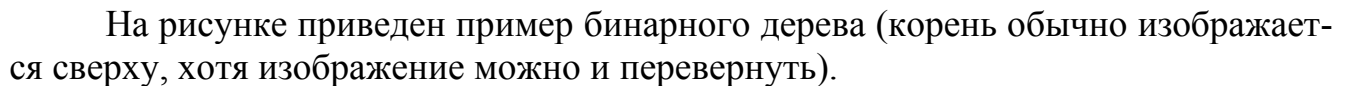
[illegible]

Diagram illustrating a B-tree structure. The root node is labeled "info" and has two pointers labeled "Left" and "Right". Each pointer points to an internal node, also labeled "info". Each internal node has two pointers that point to leaf nodes. Each leaf node is labeled "info" and contains two slots, each containing the letter "N". There are four leaf nodes in total, two under each internal node.

Если дерево организовано так, что для каждого узла все ключи его левого поддеревья меньше ключа этого узла, а все ключи его правого поддеревья – больше, оно называется деревом поиска. Одинаковые ключи здесь не допускаются.

144

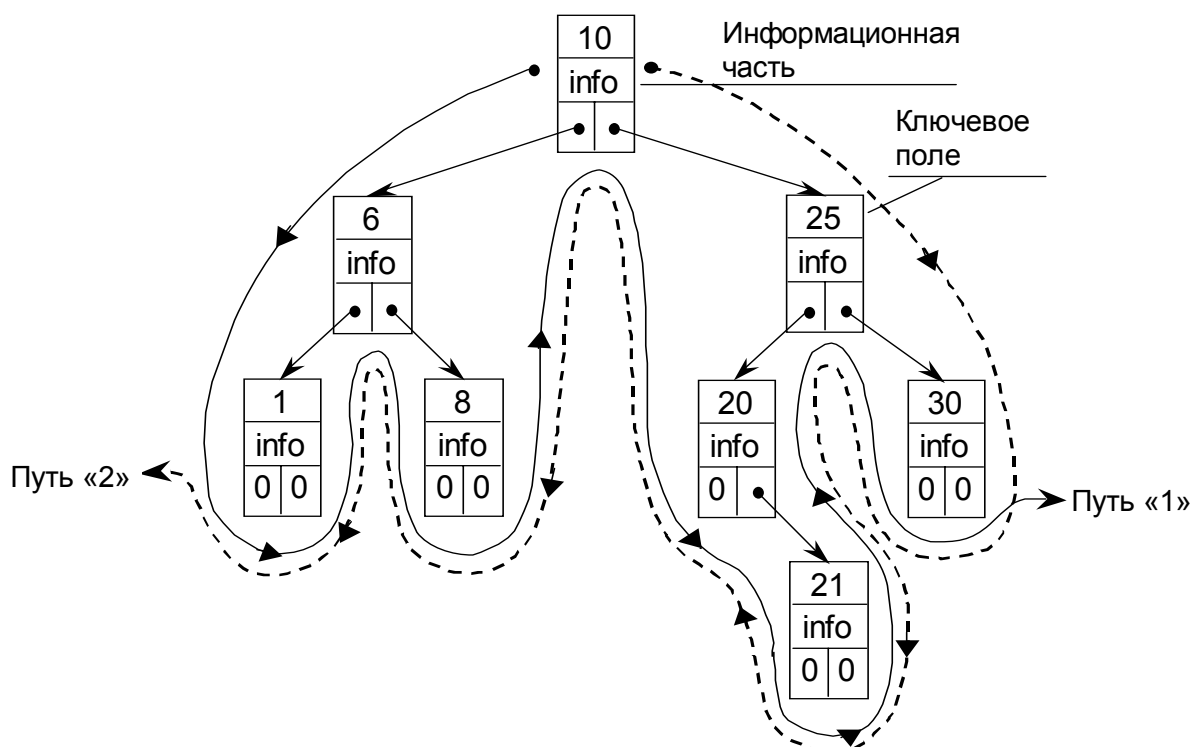
В дереве поиска можно найти элемент по ключу, двигаясь от корня, и, переходя на левое или правое поддерево, в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна логарифму количества узлов – $\log_2 N$. Время поиска по сравнению с линейной структурой сокращается с N до $\log_2 N$.

Для так называемого сбалансированного дерева, в котором количество узлов справа и слева отличается не более чем на единицу, высота дерева как раз и равна двоичному логарифму количества узлов. Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно ровно половине длины этого списка.

Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью рекурсивных алгоритмов. Например, функцию обхода всех узлов дерева можно в общем виде описать так:

```
type way_around (дерево) {
    way_around (левое поддерево);
    посещение корня;
    way_around (правое поддерево);
}
```

Можно обходить дерево и в другом порядке, например, сначала корень, потом поддеревья, но приведенная функция позволяет получить на выходе отсортированную последовательность ключей, поскольку сначала посещаются вершины с меньшими ключами, расположенные в левом поддереве, а потом вершины правого поддерева с большими ключами. На приведенном ниже рисунке это Путь «1».



Результат обхода дерева, изображенного на рисунке:

Путь «1»: $1 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 20 \rightarrow 21 \rightarrow 25 \rightarrow 30$

Если в функции обхода первое обращение идет к правому поддереву, результат обхода будет таким:

Путь «2»: $30 \rightarrow 25 \rightarrow 21 \rightarrow 20 \rightarrow 10 \rightarrow 8 \rightarrow 6 \rightarrow 1$

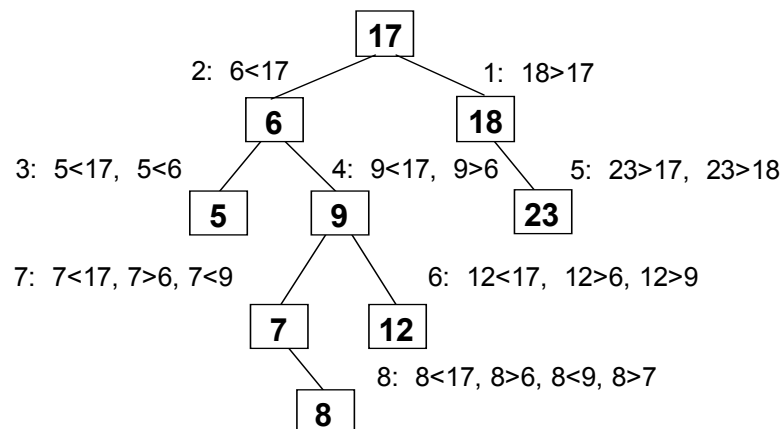
Таким образом, деревья поиска можно применять для сортировки значений (при обходе дерева узлы не удаляются).

Рассмотрим основные алгоритмы работы с бинарным деревом

Необходимо уметь:

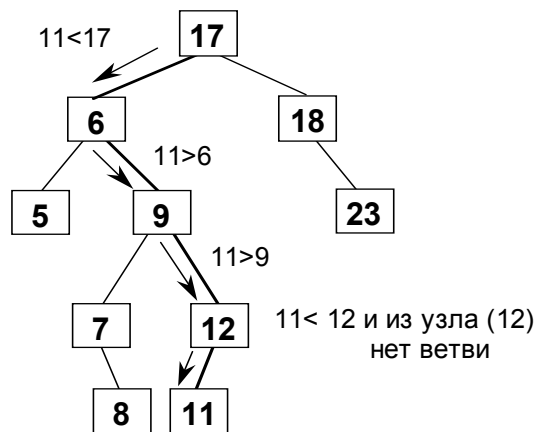
- построить (создать) дерево;
- вставить новый элемент;
- обойти все элементы дерева, например, для просмотра или чтобы произвести некоторую операцию;
- выполнить поиск элемента с указанным значением в узле;
- удалить заданный элемент.

Обычно бинарное дерево строится сразу упорядоченным, т.е. узел левого сына имеет значение меньше, чем значение родителя, а узел правого сына – большее. Например, если приходят числа 17, 18, 6, 5, 9, 23, 12, 7, 8, то построенное по ним дерево будет выглядеть следующим образом (для упрощения приводим только ключи):



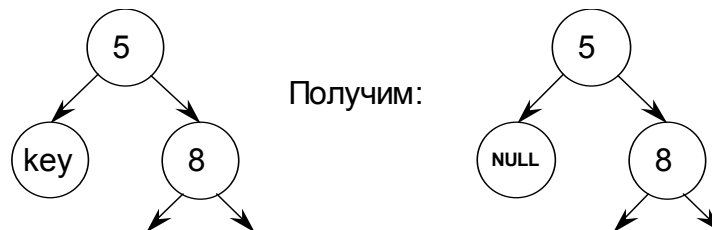
Для того чтобы вставить новый элемент в дерево, необходимо найти для него место. Для этого, начиная с корня, сравниваем значения узлов (Y) со значением нового элемента (New). Если $New < Y$, то идем по левой ветви, в противном случае – по правой ветви. Когда дойдем до узла, из которого не выходит нужная ветвь для дальнейшего поиска, это означает, что место под новый элемент найдено.

Путь поиска места в построенном дереве для числа **11**:

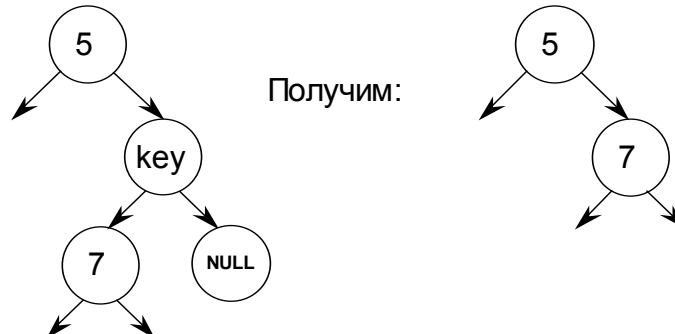


При удалении узла из дерева возможны три ситуации:

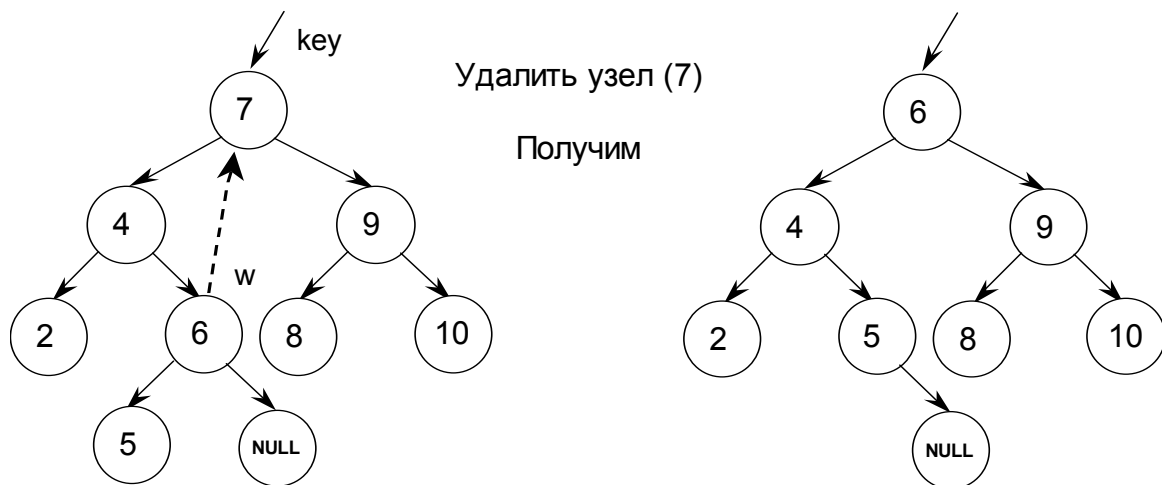
1. Удаляемый узел является листом – просто удаляем ссылку на него;
Удаление листа с ключом *key*:



2. Из удаляемого узла выходит только одна ветвь;
Удаление узла имеющего одного потомка:

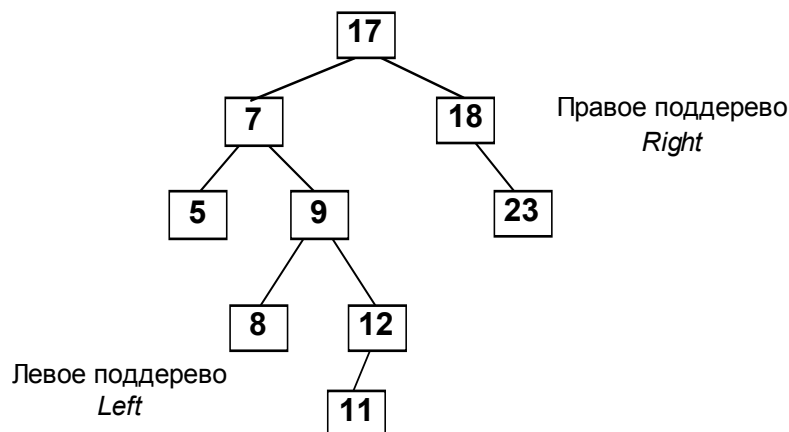


3. Удаление узла, имеющего двух потомков, значительно сложнее. Если *key* – исключаемый узел, то его следует заменить узлом *w*, который содержит либо наибольший ключ в левом поддереве, либо наименьший ключ в правом поддереве. Такой узел *w* является либо листом, либо самым правым узлом поддерева *key*, у которого имеется только левый потомок:



Таким образом, если из удаляемого узла выходит две ветви (в данном случае на место удаляемого узла надо поставить либо самый правый узел левой ветви, либо самый левый узел правой ветви для сохранения упорядоченности дерева).

Например, построенное дерево после удаления узла 6 может стать таким:



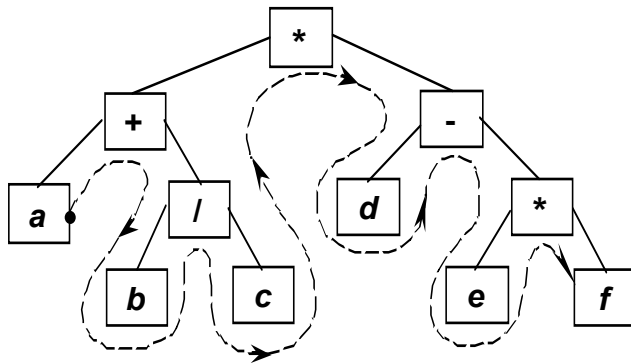
Рассмотрим задачу обхода дерева. Существуют три алгоритма обхода деревьев, которые естественно следуют из самой структуры дерева.

- 1) Обход слева направо: *Left-Root-Right* (сначала посещаем левое поддерево, затем – корень и, наконец, правое поддерево).
- 2) Обход сверху вниз: *Root-Left-Right* (посещаем корень до поддерева).
- 3) Обход снизу вверх: *Left-Right-Root* (посещаем корень после поддерева).

Интересно проследить результаты этих трех обходов на примере записи формулы в виде дерева, так как они и позволяют получить различные формы записи арифметических выражений.

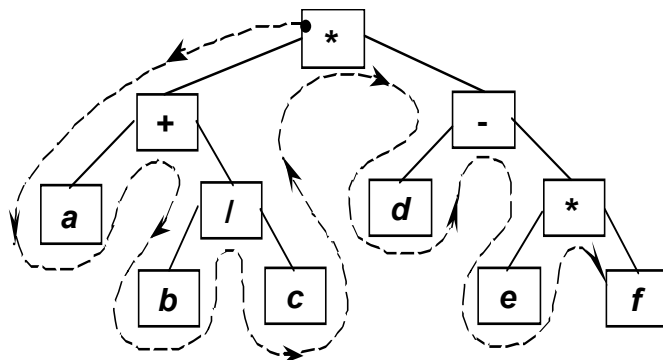
Рассмотрим на примере формулы: $((a+b/c)*(d-e*f))$. Дерево формируется по принципу:

- в корне размещаем операцию, которая выполнится последней;
- далее узлы – операции, операнды – листья дерева.



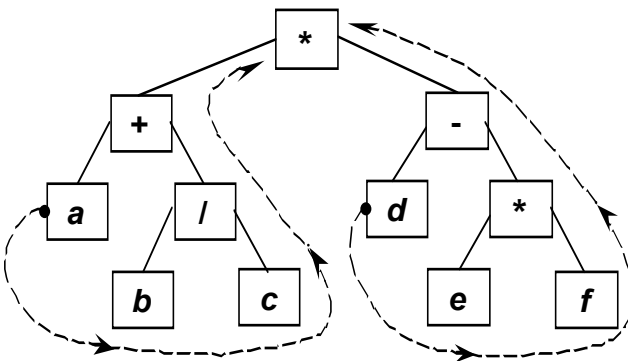
Обход 1 (*Left-Root-Right*) дает обычную инфиксную запись выражения (без скобок):

$$a + b / c * d - e * f .$$



Обход 2 (*Root-Left-Right*) – префиксную запись выражения (без скобок):

$$* + a / b c - d * e f .$$



Обход 3 (*Left-Right-Root*) – постфиксную (ПОЛИЗ – польская инверсная запись):

$$a b c / + d e f * - * .$$

Пример. Сформировать дерево из целых чисел, выполнить просмотр, добавление, удаление по ключу и освобождение памяти при выходе из программы.

Для каждого рекурсивного алгоритма можно создать его не рекурсивный эквивалент, и в приведенной ниже программе реализован не рекурсивный алгоритм поиска по дереву с включением и рекурсивная функция обхода дерева. При-

мер построения бинарного дерева, осуществляя поиск элемента с заданным ключом и, если элемент не найден – включает его в соответствующее место дерева. Для включения элемента необходимо помнить пройденный по дереву путь на один шаг назад и знать, выполняется ли включение нового элемента в левое или правое поддерево его предка.

Рекурсии удалось избежать, сохранив всего одну переменную (*Prev*), и, повторив, при включении операторы, определяющие, к какому поддереву присоединяется новый узел.

В функции обхода дерева вторым параметром передается переменная, определяющая, на каком уровне (*level*) находится узел. Корень находится на уровне «0». Значения узлов выводятся по горизонтали так, что корень находится слева. Перед значением узла для имитации структуры дерева выводится количество пробелов, пропорциональное уровню узла. Если закомментировать цикл печати пробелов, вводимые значения ключей будут выведены в столбик.

Текст программы:

```
#include <stdio.h>
#include <conio.h>
struct Tree {
    int info;
    Tree *Left, *Right;
} *Root;           // Root – указатель на корень
void Make(int);
void Print (Tree*, int);
void Del(int);
void Del_All(Tree*);
Tree* List(int);
void main() {
    int b, found, key;
    // b – для ввода ключей, found – код поиска, key – удаляемый ключ
    while(1) {
        puts(" Creat  - 1\n View   - 2\n Add    - 3 \nDel Key - 4\n EXIT   - 0");
        switch (getch()) {
            case '1': Make(0); break;
            case '2': if( Root == NULL ) puts ("\t END TREE !");
                       else Print(Root, 0);
            break;
            case '3': if(Root==NULL) Make(0);
                       else Make(1);
            break;
            case '4': puts("\n Input Del Info ");   scanf("%d", &key);
                       Del(key);
            break;
            case '0': Del_All(Root);
                       puts("\n Tree Delete!");
        }
    }
}
```

```

        return;
    } // End switch
} // End while(1)
}

//===== Создание дерева =====
void Make(int kod) {
    Tree *Prev, *t, *t1;
    int b, found;
    if ( kod == 0 ) { // Формирование первого элемента
        puts( "\n Input Root :");
        scanf("%d", &b);
        Root = List(b); // Установили указатель корня
    }

//===== Вставка остальных элементов =====
    while(1) {
        puts( "\n Input Info :"); scanf("%d", &b);
        if (b<0) break;
        t = Root;
        found = 0;
        while ( t && !found ) {
            Prev = t;
            if( b == t->info) found = 1;
            else
                if ( b < t-> info ) t = t-> Left;
                else t = t-> Right;
        }
        if (!found) {
            t1 = List(b); // Создаем новый узел
            if ( b < Prev-> info ) Prev-> Left = t1;
            else Prev-> Right = t1;
        }
    } // Конец цикла
}

//===== Удаление элемента по ключу (не корень) =====
void Del(int key) {
    Tree *Del, *Prev_Del, *R, *Prev_R;
    // Del, Prev_Del - удаляемый элемент и его предок;
    // R, Prev_R - элемент, на который заменяется удаленный и его предок;
    Del = Root; Prev_Del = NULL;
    //----- Поиск удаляемого элемента и его предка по ключу key-----
    while (Del != NULL && Del-> info != key) {
        Prev_Del = Del;
        if (Del->info >key) Del=Del->Left;
        else Del=Del->Right;
    }
}

```

```

        }
        if (Del==NULL){                                // В дереве такого ключа нет
puts ( "\n NO Key!");
return;
}

//----- Поиск элемента для замены R -----
//-----1. Если удаляемый элемент имеет одного потомка, или ЛИСТ -----
        if (Del -> Right == NULL) R = Del->Left;
        else
            if (Del -> Left == NULL) R = Del->Right;
            else {
//-----Иначе, ищем самый правый узел в левом поддереве-----
                Prev_R = Del;

                R = Del->Left;

                while (R->Right != NULL) {

                    Prev_R=R;
                    R=R->Right;
                }
//-----2. Если удаляемый элемент имеет одного потомка-----
                if( Prev_R == Del) R->Right=Del->Right;
                else {
//-----3. Если удаляемый элемент имеет двух потомков -----
                    R->Right=Del->Right;
                    Prev_R->Right=R->Left;
                    R->Left=Prev_R;
                }
            }
        }

// Устанавливаем связь с предком удаляемого (Prev_Del) и заменой (R):
        if (Prev_Del==NULL) { Root = Prev_Del = R; }
        else
            if (Del->info < Prev_Del->info) Prev_Del->Left=R;
            else Prev_Del->Right=R;

        printf("\n Delete %d element ",Del->info);
        delete Del;
    }

//===== Формирование (создание) элемента - листа =====
Tree* List(int i) {
    Tree *t = new Tree;                                // Захват памяти
    t -> info = i;
    t -> Left = t -> Right = NULL;

    return t;
}

//===== Функция вывода на экран =====
void Print ( Tree *p, int level ) {
    if ( p ) {

```



```

        Print ( p -> Right , level+1); // Вывод левого поддерева
        for ( int i=0; i<level; i++) printf("    ");
            printf("%d \n", p->info);
        Print( p -> Left , level+1);      // Вывод правого поддерева
    }
}

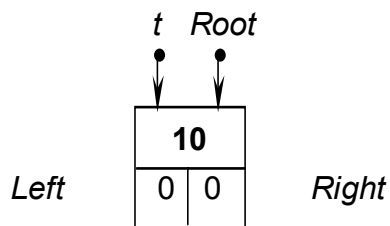
//===== Освобождение памяти =====
void Del_All(Tree *t) {
    if ( t != NULL) {
        Del_All ( t -> Left);
        Del_All ( t -> Right);
        delete t;
    }
}

```

Пояснение к программе

Вводим ключи: 10, 25, 20, 6, 21, 8, 1, 30.

1. Формирование первого элемента (корня) со значением 10 приведет к следующему виду (0 – *NULL*):



Далее, при определении, в какое место (в левое или правое поддерево) добавлять очередной элемент, воспользуемся тем, что бинарное дерево строится упорядоченным по ключевому полю:

- узел левого сына (потомка) должен быть меньше узла родителя (предка);
- узел правого сына – больше узла родителя.

Рассмотрим несколько шагов цикла *while*.

Шаг 1: b = 25;

1. Установили текущий указатель на корень: Tree *t = Root;
2. Флаг для контроля поиска int found=0; чтобы исключить повторение ключей.
3. Внутренний цикл *while*, выполнять пока текущий указатель *t* != *NULL* и искомый ключ не найден (!*found*);
4. Установили указатель на предка на текущий элемент: Prev = t;
5. Проверяем значение текущего ключа:
 - 1) если b = t -> info, то found = 1; *else* – игнорируется, вставку не производим, т.к. такой уже есть;
 - 2) если found=1, уходим на новый виток цикла;
 - 3) иначе (25 не равно 10),

если $b < t \rightarrow \text{info}$, то создаем левую ветвь ($t = t \rightarrow \text{Left}$),
иначе – создаем правую ветвь ($t = t \rightarrow \text{Right}$).

6. Создаем новый элемент:

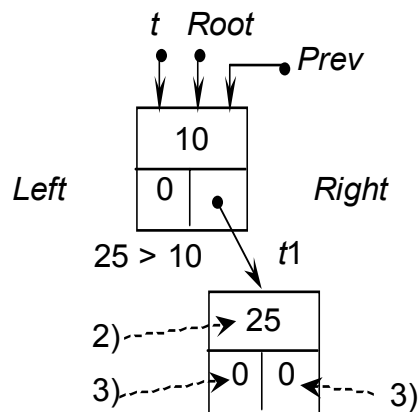
1) захватываем память: $\text{Tree } *t1 = \text{new Tree};$

2) формируем информационную часть: $t1 \rightarrow \text{info} = b;$ (25)

3) указатели на потомков в *NULL*:

$t1 \rightarrow \text{Left} = t1 \rightarrow \text{Right} = \text{NULL};$

На этом этапе получили следующее:



Шаг 2: $i = 2, b = 20;$

1. $\text{Tree } *t = \text{Root};$

2. $\text{found} = 0;$

3. Внутренний цикл *while*, выполнять пока $t \neq 0$ и $! \text{found};$

$\text{Prev} = t;$

4. Проверяем значение текущего ключа:

1) (шаг 1.1): если $b[i] \neq t \rightarrow \text{info}$ ($10 \neq 20$), – Нет!

иначе: если $b[i] < t \rightarrow \text{info}$ ($20 < 10$) – Нет! значит:

$t = t \rightarrow \text{Right}$ – создаем правую ветвь;

2) (шаг 1.2): если $b[i] \neq t \rightarrow \text{info}$ ($20 \neq 25$), – Нет!

иначе: если $b[i] < t \rightarrow \text{info}$ ($20 < 25$) – Да! значит:

$t = t \rightarrow \text{Left} = \text{NULL};$ – указатель на правую ветвь;

3) конец цикла *while*.

5. Создается по уже рассмотренному алгоритму новый элемент и присоединяется слева от последнего:

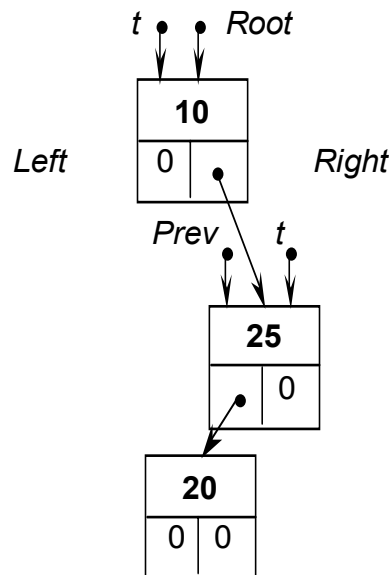
$t1 = \text{new Tree};$

$t1 \rightarrow \text{info} = b;$ (20)

$t1 \rightarrow \text{Left} = t1 \rightarrow \text{Right} = \text{NULL};$

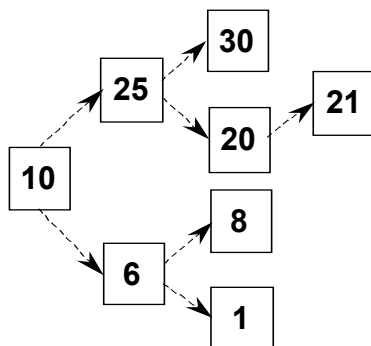
$\text{Prev} \rightarrow \text{Left} = t1;$

Графически это выглядит следующим образом:



и т.д.

Результат работы программы для ключей: 10, 25, 20, 6, 21, 8, 1, 30 (стрелками показана связь между выводимыми значениями):



9. Понятие хеширования

До сих пор среди данных большого объема остается проблемой поиск необходимого элемента. Если эти данные расположены беспорядочно в массиве (линейном списке, файле и т.п.), то осуществляют **линейный поиск**, эффективность которого $O(n/2)$. Если же данные в массиве упорядочены (например, сбалансированное двоичное дерево), то возможен **двоичный поиск** с эффективностью $O(\log_2 n)$.

Однако при работе с двоичным деревом и упорядоченным массивом затруднены операции вставки и удаления элементов, так, например, дерево – разбалансируется, и вновь требуется его балансировка. Что можно придумать в данном случае более эффективное?

В данном случае был предложен алгоритм **хеширования** (*hashing* – перемешивание), при котором создаются ключи, определяющие данные массива и на их основании данные записываются в таблицу, названную **хеш-таблицей**. Ключи

для записи определяются при помощи функции $i = h(key)$, называемой **хеш-функцией**. Алгоритм хеширования определяет положение искомого элемента в хеш-таблице по значению его ключа, полученного хеш-функцией.

Хеширование – это способ сведения хранения большого множества к более меньшему.

Возьмем, например, словарь или энциклопедию. В этом случае буквы алфавита могут быть приняты за ключи поиска, т.е. основным элементом алгоритма хеширования является **ключ** (*key*)! В большинстве приложений ключ обеспечивает косвенную ссылку на данные.

Термин «хеширование» в литературе по программированию появился не так давно – в 1967 году, ввел его Хеллерман (*Hellerman*). Дословный перевод означает – рубить, крошить, но академик А.П.Ершов предложил довольно удачный эквивалент – «расстановка».

Фактически хеширование – это специальный метод адресации данных для быстрого поиска нужной информации **по ключам**.

Или **хеширование – это разбиение общего (базового) набора уникальных ключей элементов данных на непересекающиеся наборы с определенным свойством**.

Если базовый набор содержит N элементов, то его можно разбить на 2^N различных подмножеств.

9.1. Хеш-функция и хеш-таблица

Функция, которая описывает определенное свойство подмножеств, т.е. преобразует ключ элемента данных в некоторый индекс в таблице (**хеш-таблица**), называется **функцией хеширования** или **хеш-функцией**:

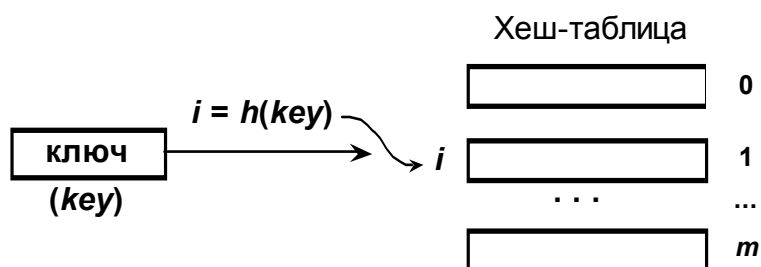
$$i = h(key);$$

где *key* – преобразуемый ключ, *i* – получаемый индекс таблицы, т.е. ключ отображается во множество, например, целых чисел (**хеш-адреса**), которые впоследствии используются для доступа к данным.

Однако функция расстановки может для **нескольких** значений ключа давать одинаковое значение позиции *i* в таблице. Ситуация, при которой два или более ключа получают один и тот же индекс (хеш-адрес) называется **коллизией** при хешировании.

Хорошей хеш-функцией считается такая функция, которая минимизирует коллизии и распределяет данные равномерно по всей таблице.

Совершенная хеш-функция – это функция, которая не порождает коллизий:



Разрешить коллизии при хешировании можно двумя методами:

- методом открытой адресации с линейным опробыванием;
- методом цепочек.

Хеш-таблица

Хеш-таблица представляет собой обычный массив с необычной адресацией, задаваемой хеш-функцией.

Хеш-структуру считают обобщением массива, который обеспечивает быстрый прямой доступ к данным по индексу. С точки зрения хеширования, массив задает отображение A множества индексов I на множество элементов E , т.е. $A:I \rightarrow E$ и позволяет по индексу быстро найти нужный элемент.

Имеется множество схем хеширования, различающихся как выбором удачной функции $h(key)$, так и алгоритма разрешения конфликтов. Эффективность решения реальной практической задачи будет существенно зависеть от выбираемой стратегии.

Примеры хеш-функций

Функция хеширования является важной частью процесса хеширования. Она используется для преобразования ключей в адреса таблицы. Она должна легко вычисляться и преобразовывать ключи (обычно целочисленные или строковые значения) в целые числа в интервале до m – максимальный размер формируемой таблицы.

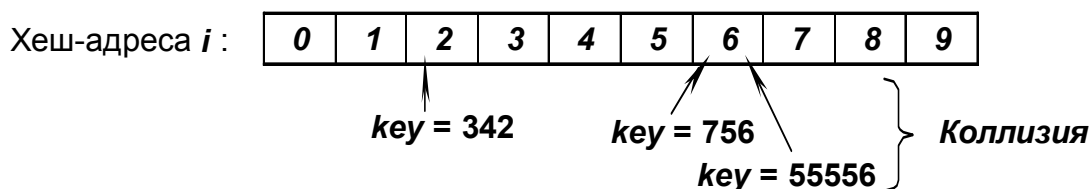
Выбираемая функция должна создавать как можно меньше коллизий, т.е. должна равномерно распределять ключи на имеющиеся индексы в таблице. Конечно, нельзя определить, будет ли некоторая конкретная хеш-функция распределять ключи правильно, если эти ключи заранее не известны. Однако, хотя до выбора хеш-функции редко известны сами ключи, некоторые свойства этих ключей, которые влияют на их распределение, обычно известны. Рассмотрим наиболее распространенные методы задания хеш-функции.

Метод деления. Исходными данными являются – некоторый целый ключ key и размер таблицы m . Результатом данной функции является остаток от деления этого ключа на размер таблицы. Общий вид функции:

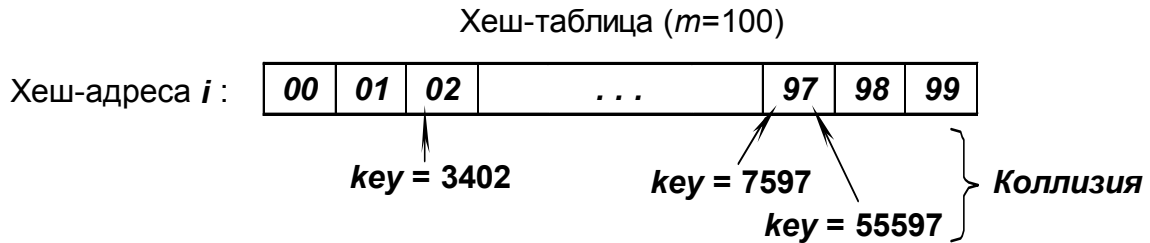
```
int h(int key, int m) {  
    return key % m;           // Значения  
}
```

Для $m = 10$ хеш-функция возвращает младшую цифру ключа.

Хеш-таблица ($m=10$)



Для $m = 100$ хеш-функция возвращает две младших цифры ключа.



Аддитивный метод, в котором ключом является символьная строка C++. В хеш-функции строка преобразуется в целое, суммированием всех символов и возвращается остаток от деления на m (обычно размер таблицы $m=256$).

```
int h(char *key, int m) {
    int s = 0;
    while(*key)
        s += *key++;
    return s % m;
}
```

Коллизии возникают в строках, состоящих из одинакового набора символов, например, *abc* и *cab*.

Данный метод можно несколько модифицировать, получая результат, суммируя только первый и последний символы строки-ключа.

```
int h(char *key, int m) {
    int len = strlen(key), s = 0;
    if(len < 2)                // Если длина ключа равна 0 или 1,
        s = key[0];           // вернуть key[0]
    else
        s = key[0] + key[len-1];
    return s % m;
}
```

В этом случае коллизии будут возникать только в строках, например, *abc* и *amc*.

Метод середины квадрата, в котором ключ возводится в квадрат (умножается сам на себя) и в качестве индекса используются несколько средних цифр полученного значения.

Например, ключом является целое 32-битное число, а хеш-функция возвращает средние 10 бит его квадрата:

```
int h(int key) {
    key *= key;
    key >>= 11;                // Отбрасываем 11 младших бит
    return key % 1024;         // Возвращаем 10 младших бит
}
```

Метод исключаящего ИЛИ для ключей-строк (обычно размер таблицы $m=256$). Этот метод аналогичен аддитивному, но в нем различаются схожие слова. Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее ИЛИ».

В **мультипликативном методе** дополнительно используется случайное действительное число r из интервала $[0,1[$, тогда дробная часть произведения $r*key$ будет находиться в интервале $[0,1]$. Если это произведение умножить на размер таблицы m , то целая часть полученного произведения даст значение в диапазоне от 0 до $m-1$.

```
int h(int key, int m) {  
    double r = key * rnd();  
    r = r - (int)r;           // Выделили дробную часть  
    return r * m;  
}
```

В общем случае при больших значениях m индексы, формируемые хеш-функцией, имеют большой разброс. Более того, математическая теория утверждает, что распределение получается более равномерным, если m является простым числом.

В рассмотренных примерах хеш-функция $i = h(key)$ только определяет позицию, начиная с которой нужно искать (или первоначально – поместить в таблицу) запись с ключом key . Поэтому схема хеширования должна включать **алгоритм решения конфликтов**, определяющий порядок действий, если позиция $i = h(key)$ оказывается уже занятой записью с другим ключом.

9.2. Схемы хеширования

В большинстве задач два и более ключей хешируются одинаково, но они не могут занимать в хеш-таблице одну и ту же ячейку. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш-таблицы отдельный список, в который помещаются все ключи, отображающиеся в этот индекс.

Эти варианты и представляют собой две классические схемы хеширования:

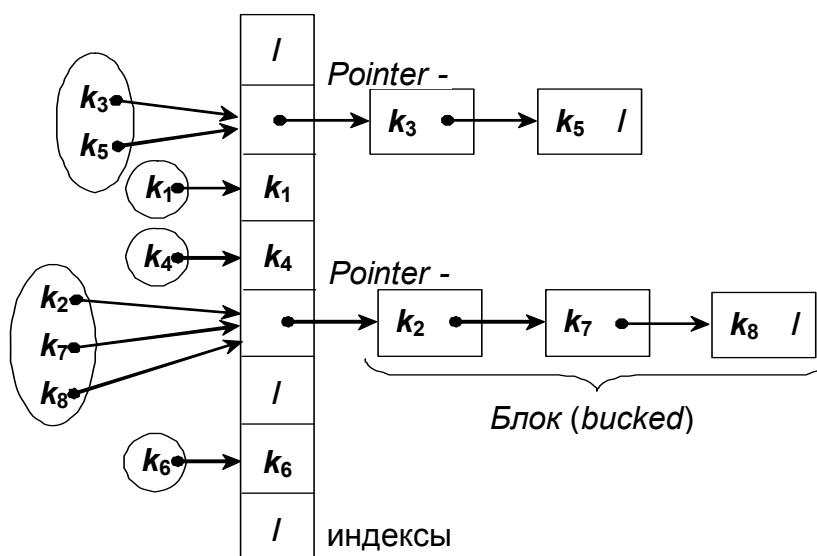
- хеширование методом цепочек (со списками), или так называемое, многомерное хеширование – *chaining with separate lists*;
- хеширование методом открытой адресацией с линейным опробыванием – *linear probe open addressing*.

Метод открытой адресацией с линейным опробыванием. Изначально все ячейки хеш-таблицы, которая является обычным одномерным массивом, помечены как не занятые. Поэтому при добавлении нового ключа проверяется, занята ли данная ячейка. Если ячейка занята, то алгоритм осуществляет осмотр по кругу до тех пор, пока не найдется свободное место («открытый адрес»).

Т.е. либо элементы с однородными ключами размещают вблизи полученного индекса, либо осуществляют двойное хеширование, используя для этого разные, но взаимосвязанные хеш-функции.

В дальнейшем, осуществляя поиск, сначала находят по ключу позицию i в таблице, и, если ключ не совпадает, то последующий поиск осуществляется в соответствии с алгоритмом разрешения конфликтов, начиная с позиции i по списку.

Метод цепочек является доминирующей стратегией. В этом случае i , полученной из выбранной хеш-функцией $h(key)=i$, трактуется как индекс в хеш-таблице списков, т.е. сначала ключ key очередной записи отображается на позицию $i = h(key)$ таблицы. Если позиция свободна, то в нее размещается элемент с ключом key , если же она занята, то отработывается алгоритм разрешения конфликтов, в результате которого такие ключи помещаются в список, начинающийся в i -той ячейке хеш-таблицы. Например



В итоге имеем таблицу массива связанных списков или деревьев.

Процесс заполнения (считывания) хеш-таблицы прост, но доступ к элементам требует выполнения следующих операций:

- вычисление индекса i ;
- поиск в соответствующей цепочке.

Для улучшения поиска при добавлении нового элемента можно использовать алгоритма вставки не в конец списка, а – с упорядочиванием, т.е. добавлять элемент в нужное место.

При решении задач на практике необходимо подобрать хеш-функцию $i = h(key)$, которая по возможности равномерно отображает значения ключа key на интервал $[0, m-1]$, m – размер хеш-таблицы. И чаще всего, если нет информации о вероятности распределения ключей по записям, используя метод деления, берут хеш-функцию $i = h(key) = key \% m$.

При решении обратной задачи – доступ (поиск) к определенному подмножеству возможен из хеш-таблицы (хеш-структуры), которая обеспечивает по хеш-адресу (индексу) быстрый доступ к нужному элементу.

Пример реализации метода прямой адресации с линейным опробыванием. Исходными данными являются 7 записей (для простоты информационная часть состоит только из целочисленных данных), объявленного структурного типа:

```
struct zap {
    int key;           // Ключ
    int info;          // Информация
} data;
```

{59,1}, {70,3}, {96,5}, {81,7}, {13,8}, {41,2}, {79,9}; размер хеш-таблицы $m=10$.

Хеш-функция $i = h(data) = data.key \% 10$; т.е. остаток от деления на 10 – $i \in [0,9]$.

На основании исходных данных последовательно заполняем хеш-таблицу.

Хеш-таблица ($m=10$)

Хеш-адреса i :	0	1	2	3	4	5	6	7	8	9
key :	70	81	41	13	79		96			59
info :	3	7	2	8	9		5			1
проба :	1	1	2	1	6		1			1

Хеширование первых пяти ключей дает различные индексы (хеш-адреса):

$$i = 59 \% 10 = 9;$$

$$i = 70 \% 10 = 0;$$

$$i = 96 \% 10 = 6;$$

$$i = 81 \% 10 = 1;$$

$$i = 13 \% 10 = 3.$$

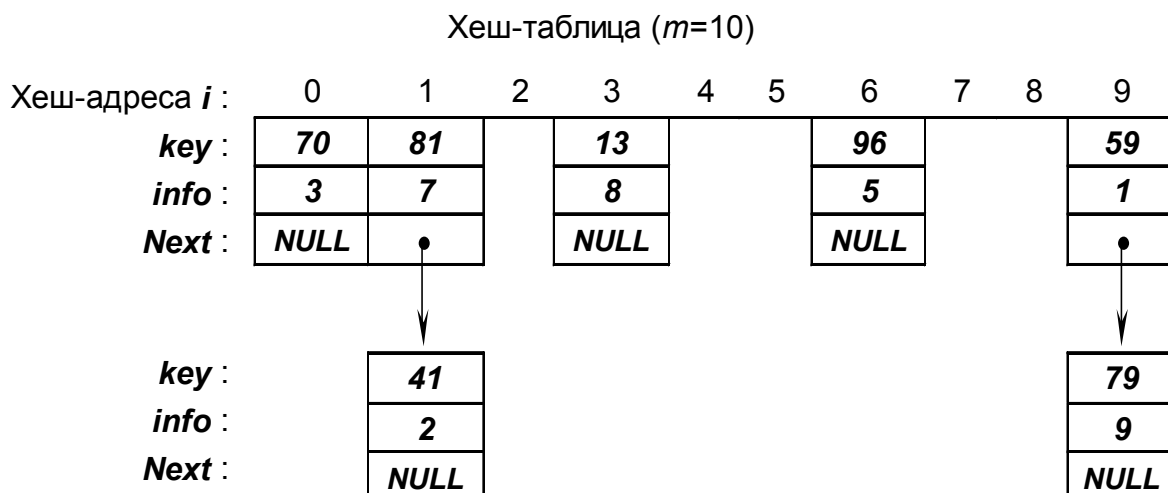
Первая коллизия возникает между ключами 81 и 41 – место с индексом 1 занято. Поэтому просматриваем хеш-таблицу с целью поиска ближайшего свободного места, в данном случае – это $i = 2$.

Следующий ключ 79 также порождает коллизия: позиция 9 уже занята. Эффективность алгоритма резко падает, т.к. для поиска свободного места понадобилось 6 проб (сравнений), свободным оказался индекс $i = 4$. Общее число проб – 1,9 пробы на элемент.

Реализация метода цепочек для предыдущего примера. Объявляем структурный тип для элемента списка (однонаправленного):

```
struct zap {
    int key;           // Ключ
    int info;          // Информация
    zap *Next;         // Указатель на следующий элемент в списке
} data;
```

На основании исходных данных последовательно заполняем хеш-таблицу, добавляя новый элемент в конец списка, если место уже занято.



Хеширование первых пяти ключей, как и в предыдущем случае, дает различные индексы (хеш-адреса): 9, 0, 6, 1, и 3.

При возникновении коллизии, новый элемент добавляется в конец списка. Поэтому элемент с ключом 41, помещается после элемента с ключом 81, а элемент с ключом 79 – после элемента с ключом 59.

10. Элементы теории погрешностей

Реальные инженерные и физические задачи во всех областях науки и техники обычно решаются посредством использования двух подходов:

- физического эксперимента;
- предварительного анализа конструкций, схем, явлений с целью выбора каких-то их оптимальных параметров.

Первый подход связан с большими и не всегда оправданными затратами материальных и временных ресурсов.

Второй подход связан с *математическим моделированием*, в основе которого заложены знания фундаментальных законов природы и построение на их основе *математических моделей* для произвольных технических и научных задач.

Математические модели представляют собой упрощенное описание исследуемого явления с помощью *математических символов и операций над ними*. Математические модели разрабатываются с соблюдением корректности и адекватности по отношению к реальным процессам, но, как правило, с учетом простоты их технической реализации.

Практика показывает, что возникающие и истребованные технические решения во многом однозначны, что определяет ограниченное число существенно полезных математических моделей, извлекаемых из стандартного справочника «Курс высшей математики». К примеру из арсенала этих моделей можно назвать такие как линейные и нелинейные уравнения, системы линейных и нелинейных уравнений, дифференциальные уравнения (ДУ), разновидности интегралов, функциональные зависимости, «целевые» функции для решения задач оптимизации и др.

При математическом моделировании важным моментом является первоначальная *математическая постановка задачи*. Она предполагает описание математической модели и указания цели ее исследования. Для одной и той же математической модели могут быть сформулированы и решаться различные математические задачи. Например, для наиболее распространенной модели, такой как функциональная зависимость $y = f(x)$ могут быть сформулированы следующие математические задачи:

- 1) найти экстремальное значение функции $f(x)$: $\max f(x)$ или $\min f(x)$;
- 2) найти значение x , при котором $f(x) = 0$;
- 3) найти значение производной $f'(x)$, значение интеграла $\int_a^b f(x)dx$ и т.д.

Вычислительная математика изучает построение и исследование численных методов решения математических задач, которые моделируют различные процессы.

10.1. Методы реализации математических моделей

Методы реализации математических моделей можно разделить на:

- графические;
- аналитические;
- численные.

Указанные методы используются как самостоятельно, так и совместно.

Графические методы позволяют оценивать порядок искомых величин и направление расчетных алгоритмы.

Аналитические методы (точные, приближенные) упрощают фрагментарные расчеты и позволяют успешно решать задачи оценки корректности и точности численных решений.

Основным инструментом реализации математических моделей являются численные методы.

Важнейшим моментом при математическом моделировании является обеспечение достоверности полученных решений. Но из практики известно, что лишь в редких случаях удастся найти метод решения, приводящий к точному результату. Как правило, приближенные решения используются совместно с точными. Поэтому наряду с выбором вычислительного метода с точки зрения оптимальности алгоритма его реализации важной задачей является оценка точности получаемого решения. Ее принято оценивать некоторой численной величиной, называемой *погрешностью*.

При решении любой практической задачи необходимо всегда указывать требуемую *точность результата*. В связи с этим необходимо уметь:

- 1) зная заданную точность исходных данных, оценивать точность результата (прямая задача теории погрешностей);
- 2) зная требуемую точность результата, выбирать необходимую точность исходных данных (обратная задача теории погрешностей).

10.2. Источники погрешностей

На рассмотренных выше этапах математического моделирования имеют место следующие источники погрешностей:

- 1) погрешность математической модели;
- 2) погрешность исходных данных (неустраняемая погрешность);
- 3) погрешность численного метода;
- 4) вычислительная погрешность.

Погрешность математической модели возникает из-за стремления обеспечить сравнительную простоту ее технической реализации и доступности исследования. Нужно иметь в виду, что конкретная математическая модель (ММ), прекрасно работающая в одних условиях, может быть совершенно неприменима в других. С точки зрения потребителя, важным является правильная оценка области ее (ММ) применения.

Погрешность численного метода (погрешность аппроксимации), связанная, например, с заменой интеграла суммой, с усечением рядов при вычислении функций, с интерполированием табличных значений функциональных зависимостей и т.п. Как правило, погрешность численного метода регулируема и может быть уменьшена до любого разумного значения путем изменения некоторого параметра.

Вычислительная погрешность возникает из-за округления чисел, промежуточных и окончательных результатов счета. Она зависит от правил и необходимости округления, а также от алгоритмов численного решения.

В основе процессов округления лежит идея минимальности разности значения s и его округления s^* .

10.3. Приближенные числа и оценка их погрешностей

При численном решении задач приходится оперировать двумя видами чисел – *точными* и *приближенными*. К точным относятся числа, которые дают истинное значение исследуемой величины. К приближенным относятся числа, близкие к истинному значению, причем степень близости определяется погрешностью вычислений.

Результатами вычислений являются, как правило, только приближенные числа. Поэтому для указания области неопределенности результата вводятся некоторые специальные понятия, широко используемые при подготовке исходных данных или (и) оценке погрешности численных решений.

Если x – точное, вообще говоря, неизвестное значение некоторой величины, а a – его приближение, то разность $x-a$ называется *ошибкой*, или *погрешностью приближения*. Часто знак ошибки $x-a$ неизвестен, поэтому используется так называемая **абсолютная погрешность** $\Delta(X)$ приближенного числа a , определяемая равенством

$$\Delta(X) = |x-a|, \quad (10.1)$$

откуда имеем

$$x = a \pm \Delta(X). \quad (10.2)$$

Изучаемая числовая величина x именованная, т.е. определяется в соответствующих единицах измерения, например, в сантиметрах, килограммах и т.п. Погрешность (10.1) имеет ту же размерность.

Однако часто возникает необходимость заменить эту погрешность безразмерной величиной – **относительной погрешностью**. При этом из-за незнания точного значения изучаемой величины принято называть относительной погрешностью величину

$$\delta(X) = \frac{\Delta(X)}{|a|} = \left| \frac{x - a}{a} \right|. \quad (10.3)$$

Относительную погрешность часто выражают в процентах: $\delta(X) = \frac{\Delta(X)}{|a|} \cdot 100\%$. Это погрешность на единицу измеряемой физической величины. Она сопоставима в идентичных экспериментах.

В связи с тем, что точное значение x , как правило, неизвестно, то формулы 1–3 носят сугубо теоретический характер.

Для практических целей вводится понятие **предельной погрешности**. *Предельная абсолютная погрешность* Δa – это верхняя оценка модуля абсолютной погрешности числа x , т.е.

$$|\Delta x| \leq \Delta a.$$

При произвольном выборе Δa всегда стремятся каким-либо образом взять **наименьшим**.

Истинное значение числа x будет находиться в интервале с границами $(a - \Delta a)$ – с недостатком и $(a + \Delta a)$ – с избытком, т.е.

$$(a - \Delta a) \leq x \leq (a + \Delta a).$$

Предельная относительная погрешность $\delta(a) = \frac{\Delta a}{|a|}$ может выражаться и в процентах.

10.4. Прямая задача теории погрешностей

Пусть в некоторой области G n -мерного числового пространства рассматривается непрерывно дифференцируемая функция

$$y = f(x_1, \dots, x_n).$$

Пусть в точке (x_1, \dots, x_n) , принадлежащей (\in) области G нужно вычислить ее (функции) значение. Известны лишь приближенные значения аргументов $(a_1, \dots, a_n) \in G$, и их погрешности. Естественно, что это будет приближенное значение

$$y^* = f(a_1, a_2, \dots, a_n).$$

Нужно оценить его абсолютную погрешность

$$\Delta y^* = |y - y^*| \lesssim \sum_{i=1}^n \Delta a_i \left| \frac{\partial}{\partial a_i} f(a_1, \dots, a_n) \right|.$$

Для функции одного аргумента $y = f(x)$ ее абсолютная погрешность, вызываемая достаточно малой погрешностью Δa , оценивается величиной

$$\Delta y^* = \lesssim |f'(a)| \cdot \Delta a.$$

10.5. Обратная задача теории погрешностей

Она состоит в определении допустимой погрешности аргументов по допустимой погрешности функции.

Для функции одной переменной $y = f(x)$ абсолютную погрешность можно вычислить приближенно по формуле

$$\Delta a = \frac{1}{|f'(a)|} \cdot \Delta y, \quad f'(a) \neq 0.$$

Для функций нескольких переменных $y = f(x_1, \dots, x_n)$ задача решается при следующих ограничениях.

Если значение одного из аргументов значительно *труднее измерить* или вычислить с той же точностью, что и значение остальных аргументов, то погрешность именно этого аргумента и согласовывают с требуемой погрешностью функции.

Если значения всех аргументов можно одинаково легко определить с любой точностью, то применяют **принцип равных влияний**, т.е. учитывают, что все слагаемые

$$\left| \frac{\partial f}{\partial x_i} \right| \cdot \Delta a_i, \quad i=1, \dots, n$$

равны между собой. Тогда абсолютные погрешности всех аргументов определяются формулой

$$\Delta a_i = \frac{\Delta y}{n \cdot |\partial f / \partial x_i|}, \quad i=1, \dots, n.$$

10.6. Понятия устойчивости, корректности и сходимости

Пусть в результате решения задачи по исходному значению величины x находится значение искомой величины y . Если исходная величина имеет абсолютную погрешность Δx , то решение y имеет погрешность Δy .

Задача называется **устойчивой по исходному параметру** x , если решение y непрерывно зависит от x , т.е. малое приращение исходной величины x приводит к малому приращению искомой величины y . Другими словами, малые погрешности в исходной величине приводят к малым погрешностям в результате расчетов.

Отсутствие устойчивости означает, что даже незначительные погрешности в исходных данных приводят к большим погрешностям в решении или вовсе к неверному результату.

Задача называется поставленной **корректно**, если для *любых* значений исходных данных ее решение существует, единственно и устойчиво по исходным данным.

Понятие сходимости численного решения вводится для итерационных процессов. По результатам многократного повторения итерационного процесса получаем последовательность приближенных значений $\overline{x_1}, \overline{x_2}, \dots, \overline{x_n}, \dots$. Говорят, что эта последовательность сходится к точному решению $\overline{x} = \overline{a}$, если $\lim_{n \rightarrow \infty} \overline{x} = \overline{a}$.

Таким образом, для получения решения задачи с необходимой точностью ее постановка должна быть корректной, а используемый численный метод должен обладать устойчивостью и сходимостью. Эти понятия будут рассматриваться в последующих разделах курса.

Некоторые обобщенные требования к выбору численных методов

Рассмотренные выше вопросы о погрешностях являются одними из важнейших моментов при выборе численного метода. В основе выбора численного метода лежат следующие соображения.

1. Можно утверждать, что нет ни одного метода, пригодного для решения всех задач одного и того же класса. Метод должен реализовываться с помощью меньшего числа действий. Поэтому всегда стоит задача выбора численного метода, сообразуясь из конкретной задачи.

2. Численный метод можно считать удачно выбранным:

- если его погрешность в несколько раз меньше неустранимой погрешности, а погрешность округлений в несколько раз меньше погрешности метода;
- если неустранимая погрешность отсутствует, то погрешность метода должна быть несколько меньше заданной точности;
- дальнейшее снижение погрешности численного метода приводит не к повышению точности результатов, а к необоснованному увеличению объема вычислений.

3. Предпочтение отдается методу, который:

- реализуется с помощью меньшего числа действий;
- требует меньшего объема памяти ЭВМ;
- логически является более простым.

4) Численный метод должен обладать устойчивостью и сходимостью.

11. Вычисление интегралов

Задачи численного интегрирования являются частным случаем общей задачи аппроксимации операторов, состоящей в том, что заданный на некотором множестве функций оператор $U = \Phi[f(x)]$ требуется заменить более простым, удобным для последующего использования, оператором $\check{U} = \Lambda[f(x)]$, близким в некотором смысле к исходному.

Аппроксимацию операторов обычно осуществляют следующим образом: функцию $f(x)$ аппроксимируют такой функцией $\varphi(x, \vec{c})$, от которой оператор Φ легко вычисляется, после чего полагают $\check{U} = \Lambda[f(x)] = \Phi[\varphi(x, \vec{c})]$.

При аппроксимации операторов численного интегрирования наибольшее распространение, ввиду своей простоты, нашли интерполяционные формулы Ньютона.

Ниже рассматривается аппроксимация $f(x)$ интерполяционным многочленом.

11.1. Формулы численного интегрирования

$$U = \int_a^b f(x) dx$$

Формулы для вычисления интеграла получают следующим образом. Область интегрирования $[a, b]$ разбивают на малые отрезки $\{a = x_1 < x_2 < \dots < x_{m+1} = b, h_i = x_{i+1} - x_i\}$, в общем случае разной длины. Значение интеграла по всей области равно сумме интегралов на отрезках

$$\int_a^b f(x) dx = \sum_{i=1}^m \int_{x_i}^{x_{i+1}} f(x) dx$$

. Выбирают на каждом отрезке $[x_i, x_{i+1}]$ 1-5 узлов и строят для каждого отрезка интерполяционный многочлен соответствующего порядка. Вычисляют интеграл от этого многочлена на отрезке. В результате получают выражение интеграла (формулу численного интегрирования) через значения подынтегральной функции в выбранной системе точек. Такие выражения называют **квадратурными формулами**.

Приведем наиболее часто используемые квадратурные формулы для равных отрезков длины $h = (b - a) / m$, $x_i = a + (i - 1) \cdot h$, $i = 1 \dots m$.

11.2. Формула средних

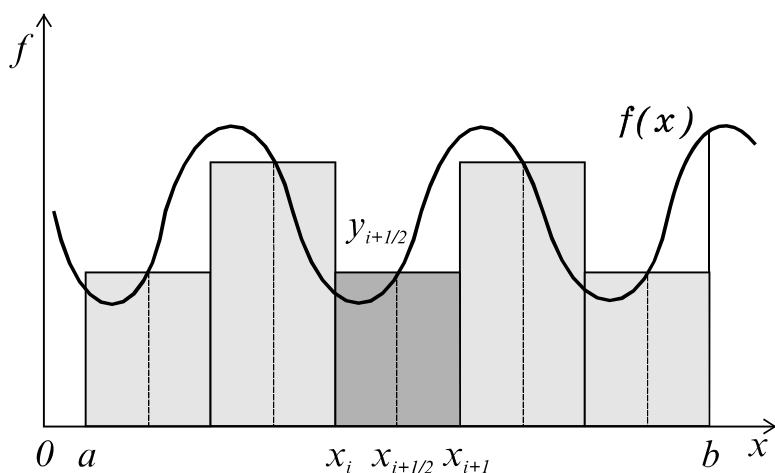


Рис. 1

Формула средних получается, если на каждом i -том отрезке взять один центральный узел $x_{i+1/2} = (x_i + x_{i+1}) / 2$, соответствующий середине отрезка. Функция на каждом отрезке аппроксимируется многочленом нулевой степени (константой)

$P_0(x) = y_{i+1/2} = f(x_{i+1/2})$. Замена площадь криволинейной фигуры площадью прямоугольника высотой $y_{i+1/2}$ и ос-

нованием h , получим (рис. 1):

$$\int_a^b f(x) dx \approx \sum_{i=1}^m \int_{x_i}^{x_{i+1}} P_0(x) dx = h \sum_{i=1}^m y_{i+1/2} = \Sigma_{cp} f \quad (11.1)$$

11.3. Формула трапеций

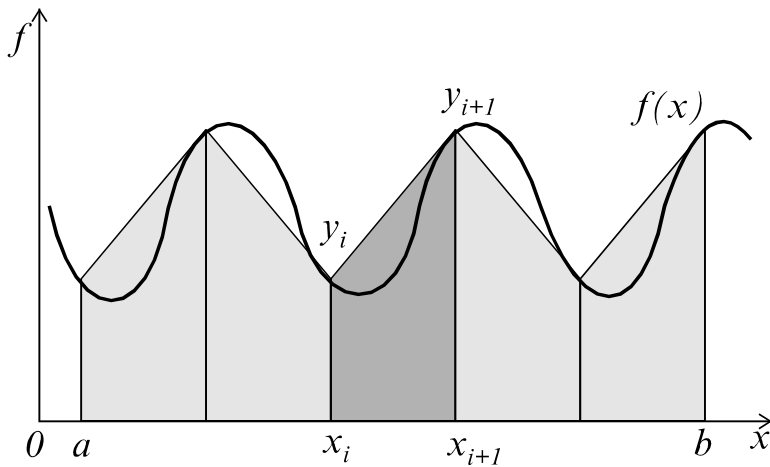


Рис. 2

Формула трапеций получается при аппроксимации функции $f(x)$ на каждом отрезке $[x_i, x_{i+1}]$ интерполяционным многочленом первого порядка, т.е. прямой, проходящей через точки (x_i, y_i) , (x_{i+1}, y_{i+1}) . Площадь криволинейной фигуры заменяется площадью трапеции с основаниями y_i, y_{i+1} и высотой h (рис. 2):

$$\int_a^b f(x)dx \approx \sum_{i=1}^m \int_{x_i}^{x_{i+1}} P_1(x)dx = h \sum_{i=1}^m \frac{y_i + y_{i+1}}{2} = h \left[\frac{y_1 + y_{m+1}}{2} + \sum_{i=2}^m y_i \right] = \Sigma_{mp} f \quad (11.2)$$

Погрешность формулы трапеций в два раза больше, чем погрешность формулы средних:

$$\varepsilon_{mp} = \max \left| \int_a^b f(x)dx - \Sigma_{mp} f \right| \leq \frac{h^2}{12} \int_a^b f''(x)dx \quad (11.3)$$

11.4. Формула Симпсона

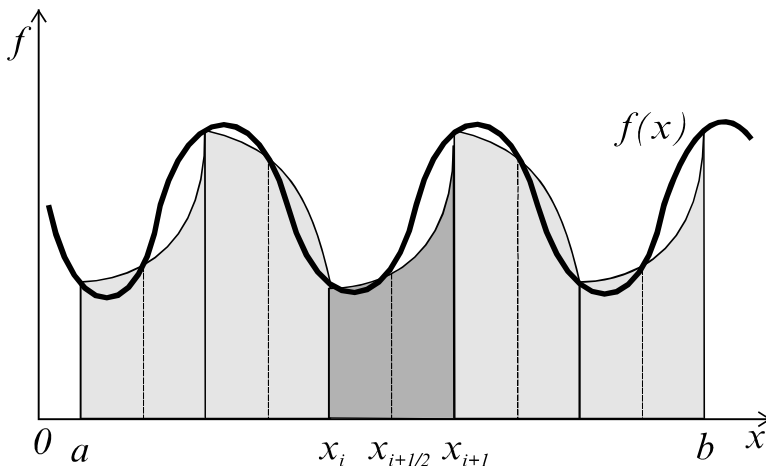


Рис. 3

Формула Симпсона получается при аппроксимации функции $f(x)$ на каждом отрезке $[x_i, x_{i+1}]$ интерполяционным многочленом второго порядка (параболой) с узлами $x_i, x_{i+1/2}, x_{i+1}$. После интегрирования параболы получаем (рис. 3):

$$\int_a^b f(x)dx \approx \sum_{i=1}^m \int_{x_i}^{x_{i+1}} P_2(x)dx = \frac{h}{6} \sum_{i=1}^m (y_i + 4y_{i+1/2} + y_{i+1}) = \Sigma_{cu} f \quad (11.4)$$

После приведения подобных членов формула (11.4) приобретает удобный для программирования вид:

$$\Sigma_{cu} f = \frac{h}{3} \cdot \left[\frac{y_1 + 4y_{1+0.5} + y_{m+1}}{2} + \sum_{i=2}^m (2y_{i+0.5} + y_i) \right].$$

Погрешность формулы Симпсона имеет четвертый порядок по h :

$$\varepsilon = \max \left| \int_a^b f dx - \Sigma_{cu} f \right| \leq \frac{h^4}{180} \int_a^b f^{(4)}(x) dx.$$

11.5. Схема с автоматическим выбором шага по заданной точности

Анализируя полученные выше формулы, выяснили, что точное значение интеграла находится между значениями Σ_{cp} и Σ_{mp} , при этом имеет место соотношение

$$\Sigma_{cu} = (\Sigma_{mp} + 2\Sigma_{cp})/3. \quad (11.5)$$

Это соотношение часто используется для контроля погрешности вычислений. Расчет начинается с $m=2$ и производится по двум методам, в результате получают Σ_{cp} , Σ_{mp} . Если $|\Sigma_{cp} - \Sigma_{mp}| \geq \delta$ (δ – заданная погрешность), то шаг h уменьшают вдвое ($m=m \cdot 2$) и расчет повторяют. Если точность достигается, то окончательное значение интеграла получается по формуле (11.5). При существенном уменьшении шага h начинают сказываться ошибки округления, поэтому шаг должен быть ограничен снизу некоторой величиной, зависящей от разрядной сетки ЭВМ ($m \leq n$).

Схема алгоритма с автоматическим выбором шага представлена на рис. 4.4 [4].

12. Методы решения нелинейных уравнений

Математической моделью многих физических процессов является функциональная зависимость $y=f(x)$. Поэтому задачи исследования различных свойств функции $f(x)$ часто возникают в инженерных расчетах. Одной из таких задач является нахождение значений x , при которых функция $f(x)$ обращается в ноль, т.е. решение уравнения

$$f(x)=0. \quad (12.1)$$

Точное решение удастся получить в исключительных случаях, и обычно для нахождения корней уравнения применяются численные методы. Решение уравнения (12.1) при этом осуществляется в два этапа:

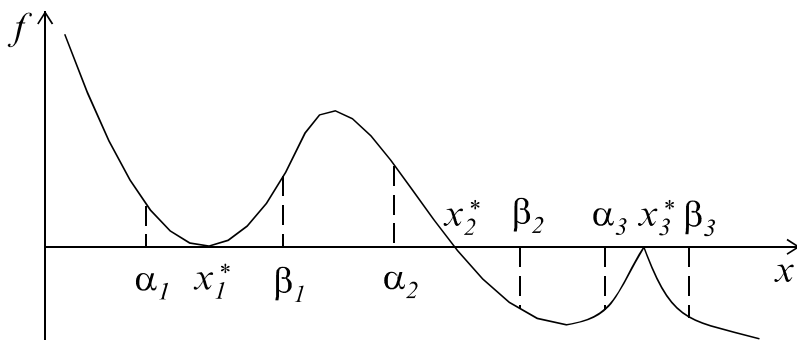


Рис. 4

1. Приближенное определение местоположения, характер и выбор интересующего нас корня.

2. Вычисление выбранного корня с заданной точностью ε .

Первая задача решается графическим методом: на заданном отрезке $[a,b]$ вычисляется таблица значений функции с некоторым шагом h , строится ее график и определяются интервалы (α_i, β_i) длиной h , на которых находятся корни. На рис.4 представлены три наиболее часто встречающиеся ситуации:

а) кратный корень: $f'(x_1^*) = 0$, $f(\alpha_1) \cdot f(\beta_1) > 0$;
 б) простой корень: $f'(x_2^*) \neq 0$, $f(\alpha_2) \cdot f(\beta_2) < 0$;
 в) вырожденный корень: $f'(x_3^*)$ не существует, $f(\alpha_3) \cdot f(\beta_3) > 0$.

Как видно из рис. 4, в случаях а) и в) значение корня совпадает с точкой экстремума функции и для нахождения таких корней рекомендуется использовать методы поиска минимума функции.

На втором этапе вычисление значения корня с заданной точностью осуществляется одним из итерационных методов. При этом, в соответствии с общей методологией m -шагового итерационного метода, на интервале (α, β) , где находится интересующий нас корень x^* , выбирается m начальных значений x_0, x_1, \dots, x_{m-1} (обычно $x_0 = \alpha, x_1 = \beta$), после чего последовательно находятся члены $(x_m, x_{m+1}, \dots, x_{n-1}, x_n)$ рекуррентной последовательности порядка m по правилу $x_k = \varphi(x_{k-1}, \dots, x_{k-m})$ до тех пор, пока $|x_n - x_{n-1}| < \varepsilon$. Последнее x_n выбирается в качестве приближенного значения корня ($x \approx x_n$).

Многообразие методов определяется возможностью большого выбора законов φ . Наиболее часто используемые на практике методы описаны ниже.

12.1. Итерационные методы уточнения корней

Очень часто в практике вычислений встречается ситуация, когда уравнение (12.1) записано в виде разрешенном, относительно x :

$$x = \varphi(x). \quad (12.2)$$

Переход от записи уравнения (12.1) к эквивалентной записи (12.2) можно сделать многими способами, например, положив

$$\varphi(x) = x + \psi(x) \cdot f(x), \quad (12.3)$$

где $\psi(x)$ – произвольная, непрерывная, знакопостоянная функция (часто достаточно выбрать $\psi = \text{const}$).

В этом случае корни уравнения (12.2) являются также корнями (12.1), и наоборот.

12.2. Метод Ньютона

Этот метод часто называется методом касательных. Если $f(x)$ имеет непрерывную производную, тогда, выбрав в (3) $\psi(x) = 1/f'(x)$, получаем эквивалентное уравнение $x = x - f(x)/f'(x) = \varphi(x)$. Скорость сходимости рекуррентной последовательности метода Ньютона

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})} = \varphi(x_{k-1}) \quad (12.4)$$

вблизи корня очень большая, погрешность очередного приближения примерно равна квадрату погрешности предыдущего $\varepsilon_k \cong |\varphi''(x^*)| \varepsilon_{k-1}^2$.

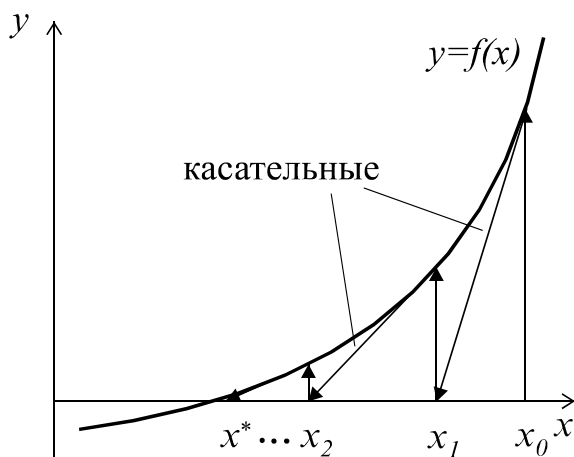


Рис. 5

Из (12.4) видно, что этот метод одношаговый ($m=1$) и для начала вычислений требуется задать одно начальное приближение x_0 из области сходимости, определяемой неравенством $|f \cdot f''|/(f')^2 < 1$. Метод Ньютона получил название метод касательных благодаря геометрической иллюстрации его сходимости, представленной на рис. 5. Этот метод позволяет находить как простые, так и кратные корни. Основным его недостаток – малая область сходимости и необходимость вычисления производной

$f'(x)$.

12.3. Метод секущих

Данный метод является модификацией метода Ньютона, позволяющей избавиться от явного вычисления производной путем ее замены приближенной

формулой. Это эквивалентно тому, что вместо касательной проводится секущая (рис. 6). Тогда вместо процесса (12.4) получаем

$$x_k = x_{k-1} - \frac{f(x_{k-1})h}{f(x_{k-1}) - f(x_{k-1} - h)} = \varphi(x_{k-1}) \quad (12.5)$$

Здесь h – некоторый малый параметр метода, который подбирается из условия наиболее точного вычисления производной.

Метод одношаговый ($m=1$), и его условие сходимости при правильном выборе h такое же, как у метода Ньютона.

12.4. Метод Вегстейна

Этот метод является модификацией метода секущих. В нем предлагается при расчете приближенного значения производной по разностной формуле использовать вместо точки $x_{k-1}-h$ в (12.5) точку x_{k-2} , полученную на предыдущей итерации. Расчетная формула метода Вегстейна:

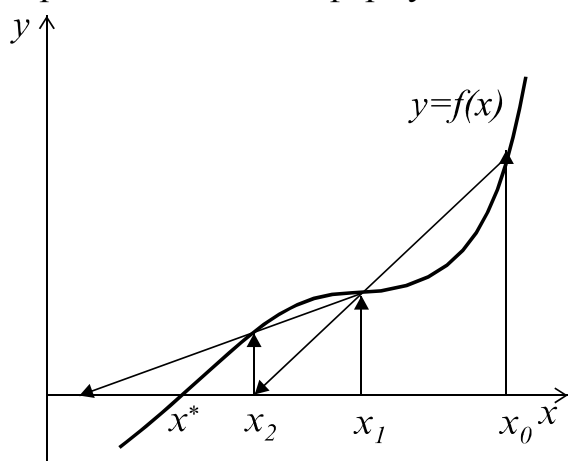


Рис. 6

$$x_k = x_{k-1} - \frac{f(x_{k-1})(x_{k-1} - x_{k-2})}{f(x_{k-1}) - f(x_{k-2})} = \varphi(x_{k-1}, x_{k-2}) \quad (12.6)$$

Метод является двухшаговым ($m=2$), и для начала вычислений требуется задать 2 начальных приближения x_0, x_1 . Лучше всего $x_0=\alpha$, $x_1=\beta$. Метод Вегстейна сходится медленнее метода секущих, однако, требует в 2 раза меньшего числа вычислений $f(x)$ и за счет этого оказывается более эффектив-

ным.

Схема алгоритма метода Вегстейна представлена на рис. 5.6 [4].

12.5. Метод парабол

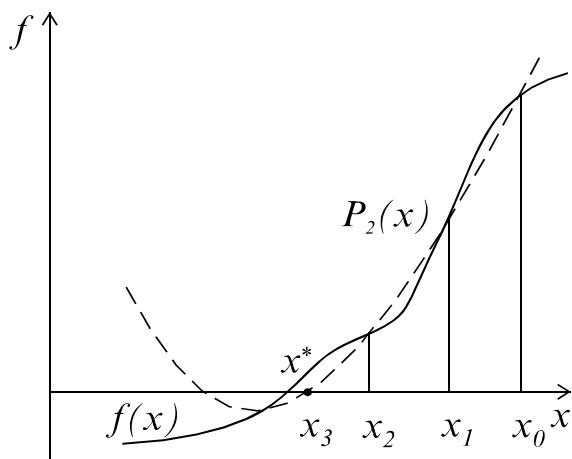


Рис. 7

Предыдущие методы основаны на том, что исходная функция $f(x)$ аппроксимируется линейной зависимостью вблизи корня и в качестве следующего приближения выбирается точка пересечения аппроксимирующей прямой с осью абсцисс. Ясно, что аппроксимация будет лучше, если вместо линейной зависимости использовать квадратичную. На этом и основан один из самых эффективных методов – метод парабол. Суть его в следующем: задаются три начальные точки x_0, x_1, x_2 (обычно $x_0=\alpha$,

$x_2=\beta$, $x_1=(\alpha+\beta)/2$, в этих точках рассчитываются три значения функции $y=f(x)$, y_0, y_1, y_2 и строится интерполяционный многочлен второго порядка (рис.7), который удобно записать в форме

$$P_2 = a(x - x_2)^2 + b(x - x_2) + c = az^2 + bz + c \quad (12.7)$$

Коэффициенты этого многочлена вычисляются по формулам:

$$z = x - x_2; \quad z_0 = x_0 - x_2; \quad z_1 = x_1 - x_2; \quad c = y_2;$$

$$a = \frac{(y_0 - y_2)z_1 - (y_1 - y_2)z_0}{z_0 z_1 (z_0 - z_1)}; \quad b = \frac{(y_0 - y_2)z_1^2 - (y_1 - y_2)z_0^2}{z_0 z_1 (z_1 - z_0)} \quad (12.8)$$

Полином (12.7) имеет два корня:

$$z_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

из которых выбирается наименьший по модулю z_m и рассчитывается следующая точка $x_3 = x_2 + z_m$, в результате получается рекуррентная формула метода парабол:

$$x_k = x_{k-1} + z_m(x_{k-1}, x_{k-2}, x_{k-3}) = \varphi(x_{k-1}, x_{k-2}, x_{k-3}). \quad (12.9)$$

Метод трехшаговый ($m=3$). Скорость сходимости его больше, чем у метода Вегстейна, однако, не лучше, чем у метода Ньютона вблизи корня. Схема алгоритма метода парабол представлена на рис. 5.8 [4].

12.6. Метод деления отрезка пополам

Все вышеописанные методы могут работать, если функция $f(x)$ является непрерывной и дифференцируемой вблизи искомого корня. В противном случае они не гарантируют получение решения.

Для разрывных функций, а также, если не требуется быстрая сходимость, для нахождения простого корня на интервале $[\alpha, \beta]$ применяют надежный метод деления отрезка пополам. Его алгоритм основан на построении рекуррентной последовательности по следующему закону: в качестве начального приближения выбираются границы интервала, на котором точно имеется один простой корень $x_0=\alpha$, $x_1=\beta$, далее находится его середина $x_2=(x_0+x_1)/2$; очередная точка x_3 выбирается как середина того из смежных с x_2 интервалов $[x_0, x_2]$ или $[x_2, x_1]$, на котором находится корень. В результате получается следующий алгоритм метода деления отрезка пополам:

- 1) вычисляем $y_0=f(x_0)$;
- 2) вычисляем $x_2=(x_0+x_1)/2$, $y_2=f(x_2)$;
- 3) если $y_0 \cdot y_2 > 0$, тогда $x_0=x_2$, $y_0=y_2$,
иначе $x_1=x_2$;
- 4) если $x_1 - x_0 > \varepsilon$, тогда повторять с п.2;
- 5) вычисляем $x^*=(x_0+x_1)/2$;
- 6) конец.

За одно вычисление функции погрешность уменьшается вдвое, то есть скорость сходимости невелика, однако метод устойчив к ошибкам округления и всегда сходится.

Рассмотрим функцию определения корня уравнения $f(x)=0$ на отрезке $[a,b]$ с заданной точностью eps . Предположим для простоты, что исходные данные задаются без ошибок, т.е. $\text{eps}>0$, $f(a)*f(b)<0$, $b>a$, и вопрос о возможности существования нескольких корней на отрезке $[a,b]$ нас не интересует. Не очень эффективная рекурсивная функция для решения поставленной задачи приведена в следующей программе:

```

...
int counter = 0;          // Счетчик обращений к тестовой функции
//----- Нахождение корня уравнения методом деления отрезка пополам -----
double Root(double f(double), double a, double b, double eps)
{
    double fa = f(a), fb = f(b), c, fc;
    if ( fa * fb > 0)
    {
        printf("\n На интервале a,b НЕТ корня!");
        exit(1);
    }
    c = (a + b) / 2.0;
    fc = f(c);
    if (fc == 0.0 || (b - a) < eps) return c;
    return (fa * fc < 0.0) ? Root(f, a, c, eps) : Root(f, c, b, eps);
}

//-----

void main()
{
    double x, a=0.1, b=3.5, eps=0.00001;
    double fun(double);          // Прототип тестовой функции
    x = Root (fun, a, b, eps);
    printf ("\n Число обращений к тестовой функции = %d ", counter);
    printf ("\n Корень = %f ", x);
    getch();
}

//----- Определение тестовой функции fun -----
double fun (double x)
{
    counter++;          // Счетчик обращений – глобальная переменная
    return (2.0/x * cos(x/2.0));
}

```

Значения А, В и EPS заданы постоянными только для тестового анализа полученных результатов, хотя лучше данные вводить с клавиатуры.

В результате выполнения программы с определенными в ней конкретными данными получим:

Число обращений к тестовой функции = 60

Корень = 3.141591

На рис. 5.9 [4] приведена схема определения всех корней функции $f(x)$ в указанном интервале $[a, b]$ – на экран выдается таблица значений функции и делается запрос на ввод начального приближения (это может быть α , β или x_0) к тому корню, который надо получить с заданной точностью. После того как введены требуемые данные, идет обращение к подпрограмме и печать результатов.

13. Решение систем линейных алгебраических уравнений

Система линейных алгебраических уравнений (СЛАУ) является одной из математических моделей линейной алгебры. На ее базе ставятся такие важные практические математические задачи, как:

- непосредственное решение линейных систем;
- вычисление определителей матриц;
- вычисление элементов обратных матриц;
- определение собственных значений и собственных векторов матриц.

Решение линейных систем является одной из самых распространенных и важных задач вычислительной математики. К их решению сводятся многочисленные практические задачи нелинейного характера, решения дифференциальных уравнений и др.

Вторая и третья задачи являются также и компонентами технологии решения самих линейных систем.

Обычно СЛАУ n -го порядка записывается в виде

$$\sum_{j=1}^n a_{ij}x_j = b_i; \quad i=\overline{1,n}$$

или в развернутой форме

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{cases} \quad (13.1)$$

или в векторной форме

$$A\bar{x} = \bar{b}, \quad (13.2)$$

где

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}; \quad \bar{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}; \quad \bar{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

В приведенных формах A называется основной матрицей системы с n^2 элементами;

$\bar{x} = (x_1, x_2, \dots, x_n)^T$ – вектор-столбец неизвестных;

$\bar{b} = (b_1, b_2, \dots, b_n)^T$ – вектор-столбец свободных членов.

Определителем (детерминантом – \det) матрицы A n -го порядка называется число $D (\det A)$, равное

$$|A| = D = \det A = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} = \sum (-1)^k a_{1\alpha} a_{2\beta} \cdots a_{n\omega} . \quad (13.3)$$

Здесь индексы $\alpha, \beta, \dots, \omega$ пробегают все возможные $n!$ перестановок номеров $1, 2, \dots, n$; k – число инверсий в данной перестановке.

Первоначальным при решении СЛАУ (13.1) является анализ вида исходной матрицы A и вектора-столбца свободных членов \bar{b} в (13.2).

Если все свободные члены равны нулю, т.е. $\bar{b} = 0$, то система $A\bar{x} = 0$ называется **однородной**. Если же $\bar{b} \neq 0$, или хотя бы одно $b_i \neq 0$ ($i = \overline{1, n}$), то система (13.2) называется **неоднородной**.

Квадратная матрица A называется **невырожденной**, или **неособенной**, если ее определитель $|A| \neq 0$. При этом система (13.1) имеет единственное решение.

При $|A| = 0$ матрица A называется **вырожденной**, или **особенной**, а система (13.1) не имеет решения, либо имеет бесконечное множество решений.

Если $|A| \approx 0$ система (13.1) называется **плохо обусловленной**, т.е. решение очень чувствительно к изменению коэффициентов системы.

В ряде случаев получаются системы уравнений с матрицами специальных видов: диагональные, трехдиагональные (частный случай ленточных), симметричные ($a_{ij} = a_{ji}$), единичные (частный случай диагональной), треугольные и др.

Решение системы (13.2) заключается в отыскании вектора-столбца $\bar{x} = (x_1, x_2, \dots, x_n)^T$, который обращает каждое уравнение системы в тождество.

Существуют две величины, характеризующие степень отклонения полученного решения от точного, обусловленные округлением и ограниченностью разрядной сетки ЭВМ – погрешность ε и «невязка» r :

$$\begin{cases} \varepsilon = \bar{x} - \bar{x}^* \\ r = \bar{B} - A\bar{x}^* \end{cases} \quad (13.4)$$

где \bar{x}^* – вектор решения.

Доказано, что если $\varepsilon \approx 0$, то и $r = 0$. Обратное утверждение не всегда верно. Однако если система не плохо обусловлена, для оценки точности решения используют невязку r .

Методы решения СЛАУ делятся на две группы:

- прямые (точные);
- итерационные (приближенные).

13.1. Прямые методы решения СЛАУ

К **прямым** относятся такие методы, которые, в предположении, что вычисления ведутся без округлений, позволяют получить точные значения неизвестных. Они просты, универсальны и используются для широкого класса систем. Однако они не применимы к системам больших порядков ($n < 200$) и к плохо обусловленным системам из-за возникновения больших погрешностей. К ним можно отне-

сти: *правило Крамера, методы обратных матриц, Гаусса, прогонки, квадратного корня* и др.

13.2. Метод Гаусса

Этот метод является наиболее распространенным методом решения СЛАУ. В его основе лежит идея последовательного исключения неизвестных. (В основном, приводящее исходную систему к треугольному виду, в котором все коэффициенты ниже главной диагонали равны нулю). Существуют различные вычислительные схемы, реализующие этот метод. Наибольшее распространение имеют схемы с выбором главного элемента либо по строке, либо по столбцу, либо по всей матрице. С точки зрения простоты реализации, хотя и с потерей точности, перед этими схемами целесообразней применять так называемую схему единственного деления. Рассмотрим ее суть.

Используя диагональный коэффициент первого уравнения исключается x_1 из последующих уравнений. Далее посредством второго уравнения исключается x_2 из последующих уравнений и т.д. Этот процесс называется **прямым ходом Гаусса**. Исключение неизвестных повторяется до тех пор, пока в левой части последнего n -го уравнения не останется одно неизвестное x_n

$$a'_{nn}x_n = b', \quad (13.6)$$

где a'_{nm} и b' – коэффициенты, полученные в результате линейных (эквивалентных) преобразований.

Прямой ход реализуется по формулам

$$\left. \begin{aligned} a^*_{mi} &= a_{mi} - a_{ki} \frac{a_{mk}}{a_{kk}}, \quad k = \overline{1, n-1}; \quad i = \overline{k, n}; \\ b^*_m &= b_m - b_k \frac{a_{mk}}{a_{kk}}, \quad m = \overline{k+1, n} \end{aligned} \right\} \quad (13.7)$$

где m – номер уравнения, из которого исключается x^k ;

k – номер неизвестного, которое исключается из оставшихся $(n-k)$ уравнений, а также обозначает номер уравнения, с помощью которого исключается x^k ;

i – номер столбца исходной матрицы;

a_{kk} – главный (ведущий) элемент матрицы.

Во время счета необходимо следить, чтобы $a_{kk} \neq 0$. В противном случае прибегают к перестановке строк матрицы.

Обратный ход метода Гаусса состоит в последовательном вычислении x_n, x_{n-1}, \dots, x_1 , начиная с (6) по алгоритму

$$x_n = b' / a'_{nn}; \quad x_k = \frac{1}{a'_{kk}} \left[b'_k - \sum_{i=k+1}^n a'_{ki} x_i \right], \quad k = \overline{n-1, 1}. \quad (13.8)$$

Точность полученного решения оценивается посредством «невязки» (13.4). В векторе невязки $(r_1, r_2, \dots, r_n)^T$ отыскивается максимальный элемент и сравнива-

ется с заданной точностью ε . Приемлемое решение будет, если $r_{\max} < \varepsilon$. В противном случае следует применить схему уточнения решения.

Уточнение корней

Полученные методом Гаусса приближенные значения корней можно уточнить.

Пусть для системы $A\bar{x} = \bar{b}$ найдено приближенное решение x_0 . Положим $\bar{x} = \bar{x}_0 + \bar{\delta}$. Для получения поправки $\delta = (\delta_1, \delta_2, \dots, \delta_n)^T$ корня \bar{x}_0 следует рассмотреть новую систему

$$A(\bar{x}_0 + \bar{\delta}) = \bar{b} \quad \text{или} \quad A\bar{\delta} = \bar{\varepsilon},$$

где $\bar{\varepsilon} = \bar{b} - A\bar{x}_0$ – невязка для исходной системы.

Таким образом, решая линейную систему с прежней матрицей A и новым свободным членом $\bar{\varepsilon} = (\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n)^T$, получим поправки $(\delta_1, \delta_2, \dots, \delta_n)$.

Модифицированный метод Гаусса

В данном случае помимо соблюдения требования $a_{kk} \neq 0$ при реализации формул (13.7) накладываются дополнительные требования, чтобы ведущий (главный) элемент в текущем столбце в процессе преобразований исходной матрицы имел максимальное по модулю значение. Это также достигается перестановкой строк матрицы.

Блок-схема модифицированного метода Гаусса, представленна на рис. 8.

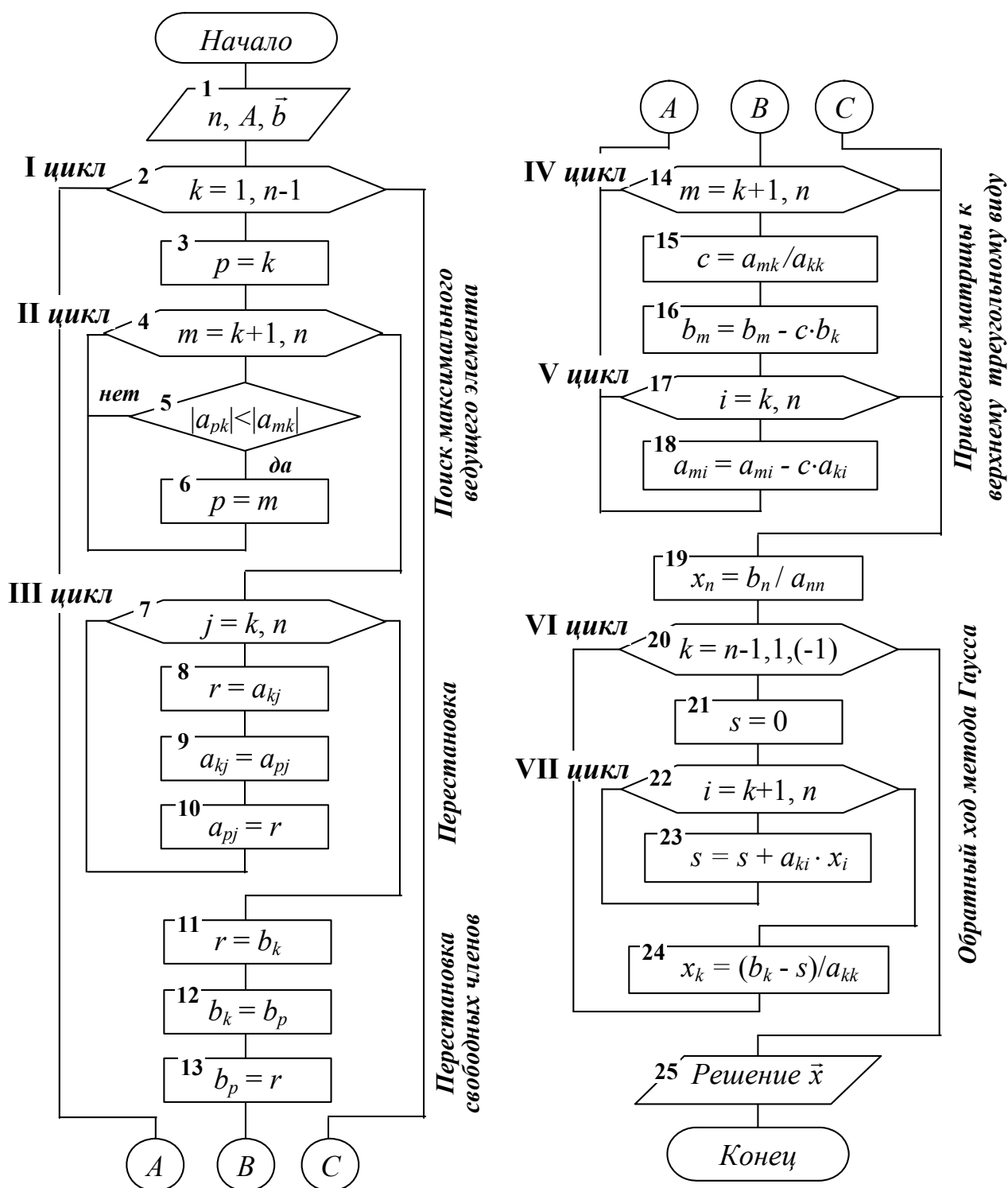


Рис. 8. Блок-схема модифицированного метода Гаусса

13.3. Метод прогонки

Данный метод является модификацией метода Гаусса для частного случая разреженных систем – систем с матрицей трехдиагонального типа (краевая задача ДУ). Каноническая форма их записи

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i; \quad i=\overline{1, n}; \quad a_1 = c_n = 0, \quad (13.9)$$

или в развернутом виде

$$\left\{ \begin{array}{ll} b_1 x_1 + c_1 x_2 & = d_1; \\ a_2 x_1 + b_2 x_2 + c_2 x_3 & = d_2; \\ a_3 x_2 + b_3 x_3 + c_3 x_4 & = d_3; \\ & \dots \\ a_{n-1} x_{n-2} + b_{n-1} x_{n-1} + c_{n-1} x_n & = d_{n-1}; \\ a_n x_{n-1} + b_n x_n & = d_n. \end{array} \right. \quad (13.10)$$

При этом, как правило, все коэффициенты $b_i \neq 0$.

Метод реализуется в два этапа – прямой и обратный ходы.

Прямой ход. Каждое неизвестное x_i выражается через x_{i+1}

$$x_i = A_i \cdot x_{i+1} + B_i \quad \text{для } i = 1, 2, \dots, n-1, \quad (13.11)$$

посредством прогоночных коэффициентов A_i и B_i . Определим алгоритм их вычисления.

Из первого уравнения системы (13.10) находим x_1

$$x_1 = -\frac{c_1}{b_1} x_2 + \frac{d_1}{b_1}.$$

Из уравнения (13.11) при $i=1$: $x_1 = A_1 \cdot x_2 + B_1$. Следовательно

$$A_1 = -\frac{c_1}{b_1}; \quad B_1 = \frac{d_1}{b_1}. \quad (13.12)$$

Из второго уравнения системы (10) определяем x_2 через x_3 , подставляя найденное значение x_1

$$a_2 (A_1 x_2 + B_1) + b_2 x_2 + c_2 x_3 = d_2,$$

откуда

$$x_2 = \frac{-c_2 x_3 + d_2 - a_2 B_1}{a_2 A_1 + b_2}; \quad (13.13)$$

и согласно (13.11) при $i=2$: $x_2 = A_2 \cdot x_3 + B_2$, следовательно

$$A_2 = -\frac{c_2}{e_2}; \quad B_2 = \frac{d_2 - a_2 B_1}{e_2}, \quad \text{где } e_2 = a_2 \cdot A_1 + b_2.$$

Ориентируясь на соотношения индексов при коэффициентах (13.12) и (13.13) можно получить эти соотношения для общего случая

$$A_i = -\frac{c_i}{e_i}; \quad B_i = \frac{d_i - a_i B_{i-1}}{e_i}, \quad \text{где } e_i = a_i \cdot A_{i-1} + b_i \quad (i=2, 3, \dots, n-1). \quad (13.14)$$

Обратный ход. Из последнего уравнения системы (13.10) с использованием (13.11) при $i=n-1$

$$x_n = \frac{d_n - a_n B_{n-1}}{b_n + a_n A_{n-1}}.$$

Далее посредством (13.11) и прогоночных коэффициентов (13.12), (13.13) последовательно вычисляем $x_{n-1}, x_{n-2}, \dots, x_1$.

При реализации метода прогонки нужно учитывать, что при условии

$$|b_i| \geq |a_i| + |c_i|, \quad (13.15)$$

или хотя бы для одного b_i имеет место строгое неравенство (13.15), деление на «0» исключается и система имеет единственное решение.

Заметим, что условие (13.15) является достаточным, но не необходимым.

Схема алгоритма метода прогонки может иметь вид, представленный на рис. 9.

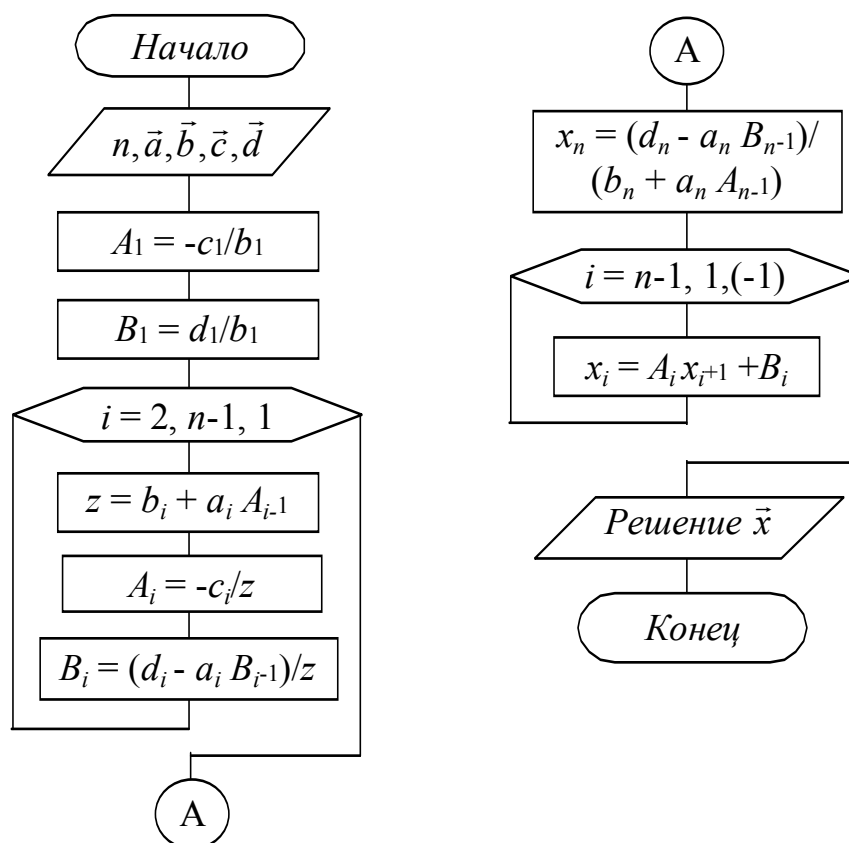


Рис. 9. Блок-схема метода прогонки

13.4. Метод квадратного корня

Данный метод используется для решения линейной системы

$$A\bar{x} = \bar{b}, \quad (13.16)$$

у которой матрица A симметрическая, т.е. $A^T = A$, $a_{ij} = a_{ji}$ ($i, j = 1, \dots, n$).

Решение системы (13.16) осуществляется в два этапа.

Прямой ход. Преобразование матрицы A и представление ее в виде произведения двух взаимно транспонированных треугольных матриц:

$$A = S^T \cdot S, \quad (13.17)$$

где

$$S = \begin{vmatrix} s_{11} & s_{12} & \cdots & s_{1n} \\ 0 & s_{22} & \cdots & s_{2n} \\ & \cdots & & \\ 0 & 0 & \cdots & s_{nn} \end{vmatrix}; \quad S^T = \begin{vmatrix} s_{11} & 0 & \cdots & 0 \\ s_{21} & s_{22} & \cdots & 0 \\ & \cdots & & \\ s_{n1} & s_{n2} & \cdots & s_{nn} \end{vmatrix}.$$

Перемножая S^T и S , и приравнявая матрице A , получим следующие формулы для определения s_{ij} :

$$\begin{cases} s_{11} = \sqrt{a_{11}}, & s_{1j} = a_{1j} / s_{11}, & (j > 1); \\ s_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} s_{ki}^2}, & & (1 \leq i \leq n); \\ s_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} s_{ki} s_{kj}}{s_{ii}}, & & (i < j); \\ s_{ij} = 0, & & (i > j). \end{cases} \quad (13.18)$$

После нахождения матрицы S системы (13.16) заменяем двумя ей эквивалентными системами с треугольными матрицами (13.17)

$$S^T \bar{y} = \bar{b}, \quad S \bar{x} = \bar{y}. \quad (13.19)$$

Обратный ход. Записываем системы (13.19) в развернутом виде:

$$\begin{cases} s_{11}y_1 = b_1; \\ s_{12}y_1 + s_{22}y_2 = b_2; \\ \cdots \\ s_{1n}y_1 + s_{2n}y_2 + \cdots + s_{nn}y_n = b_n; \end{cases} \quad (13.20)$$

$$\begin{cases} s_{11}x_1 + s_{12}x_2 + \cdots + s_{1n}x_n = y_1; \\ s_{22}x_2 + \cdots + s_{2n}x_n = y_2; \\ \cdots \\ s_{nn}x_n = y_n. \end{cases} \quad (13.21)$$

Используя (13.20) и (13.21) последовательно находим

$$y_1 = \frac{b_1}{s_{11}}, \quad y_i = (b_i - \sum_{k=1}^{i-1} s_{ki} y_k) / s_{ii}; \quad (i > 1); \quad (13.22)$$

$$x_n = \frac{y_n}{s_{nn}}, \quad x_i = (y_i - \sum_{k=i+1}^n s_{ik} x_k) / s_{ii}; \quad (i < n).$$

Метод квадратных корней дает большой выигрыш во времени по сравнению с рассмотренными ранее прямыми методами, так как, во-первых, существенно уменьшает число операций умножения и деления, во-вторых, позволяет накапливать сумму произведений без записи промежуточных результатов.

Машинная реализация метода предусматривает его следующую трактовку. Исходная матрица A системы (13.16) представляется в виде произведения трех матриц

$$A = S^T \cdot D \cdot S,$$

где D – диагональная матрица с элементами $d_{ii} = \pm 1$; S – верхняя треугольная ($s_{ik} = 0$, если $i > k$, причем $s_{ii} > 0$); S^T – транспонированная нижняя треугольная.

Требование выполнения условия $s_{ii} > 0$ необходимо для полной определенности разложения, определяющее введение диагональной матрицы D с элементами $d_{ii} = \pm 1$.

Матрицу S можно по аналогии с числами трактовать как корень квадратный из матрицы A , отсюда и название метода.

Итак, если S и D известны, то решение исходной системы $A \cdot \vec{x} = S^T \cdot D \cdot S \cdot \vec{x} = \vec{b}$ сводится к последовательному решению трех систем – двух треугольных и одной диагональной:

$$S^T \cdot \vec{z} = \vec{b}; \quad D \cdot \vec{y} = \vec{z}; \quad S \cdot \vec{x} = \vec{y}. \quad (13.23)$$

Здесь $\vec{z} = DS\vec{x}$; $\vec{y} = S\vec{x}$.

Нахождение элементов матрицы S (извлечение корня из A) осуществляется по рекуррентным формулам:

$$\begin{aligned} d_k &= \text{sign} \left(a_{kk} - \sum_{i=1}^{k-1} d_i s_{ik}^2 \right); \\ s_{kk} &= \sqrt{\left| a_{kk} - \sum_{i=1}^{k-1} d_i s_{ik}^2 \right|}; \\ s_{kj} &= \left(a_{kj} - \sum_{i=1}^{k-1} d_i s_{ik} s_{ij} \right) / (s_{kk} d_k); \end{aligned} \quad (13.24)$$

$k = 1, 2, \dots, n; \quad j = k+1, k+2, \dots, n.$

В этих формулах сначала полагаем $k = 1$ и последовательно вычисляем

$$d_1 = \text{sign}(a_{11}); \quad s_{11} = \sqrt{|a_{11}|}$$

и все элементы первой строки матрицы S ($s_{1j}, j > 1$), затем полагаем $k = 2$, вычисляем s_{22} и вторую строку матрицы s_{2j} для $j > 2$ и т.д.

Решение систем (13.23) ввиду треугольности матрицы S осуществляется по формулам, аналогичным обратному ходу метода Гаусса:

$$\begin{aligned} y_1 &= \frac{b_1}{s_{11} d_1}, & y_i &= (b_i - \sum_{k=1}^{i-1} d_k s_{ki} y_k) / (s_{ii} d_i); \quad i = 2, 3, \dots, n; \\ x_n &= \frac{y_n}{s_{nn}}, & x_i &= (y_i - \sum_{k=i+1}^n s_{ik} x_k) / s_{ii}; \quad i = n-1, n-2, \dots, 1. \end{aligned}$$

Метод квадратного корня почти вдвое эффективнее метода Гаусса, т.к. полезно использует симметричность матрицы.

Схема алгоритма метода квадратного корня представлена на рис. 10 [4].

Проиллюстрируем метод квадратного корня, решая систему трех уравнений:

$$\begin{cases} x_1 + x_2 + x_3 = 3; \\ x_1 + 2x_2 + 2x_3 = 5; \\ x_1 + 2x_2 + 3x_3 = 6; \end{cases} \quad A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 3 \\ 5 \\ 6 \end{bmatrix}.$$

Нетрудно проверить, что матрица A есть произведение двух треугольных матриц (здесь $d_{ii} = 1$):

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = S^T \cdot S.$$

Исходную систему запишем в виде

$$S^T \cdot S \cdot \vec{x} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 6 \end{bmatrix}.$$

Обозначим

$$S \cdot \vec{x} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

Тогда для вектора \vec{y} получим систему $S^T \vec{y} = \vec{b}$:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 6 \end{bmatrix}, \quad \text{откуда} \quad y_1 = 3; \quad y_2 = 2; \quad y_3 = 1.$$

Зная \vec{y} , решаем систему $S \vec{x} = \vec{y}$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}, \quad \text{откуда} \quad x_1 = 1; \quad x_2 = 1; \quad x_3 = 1.$$

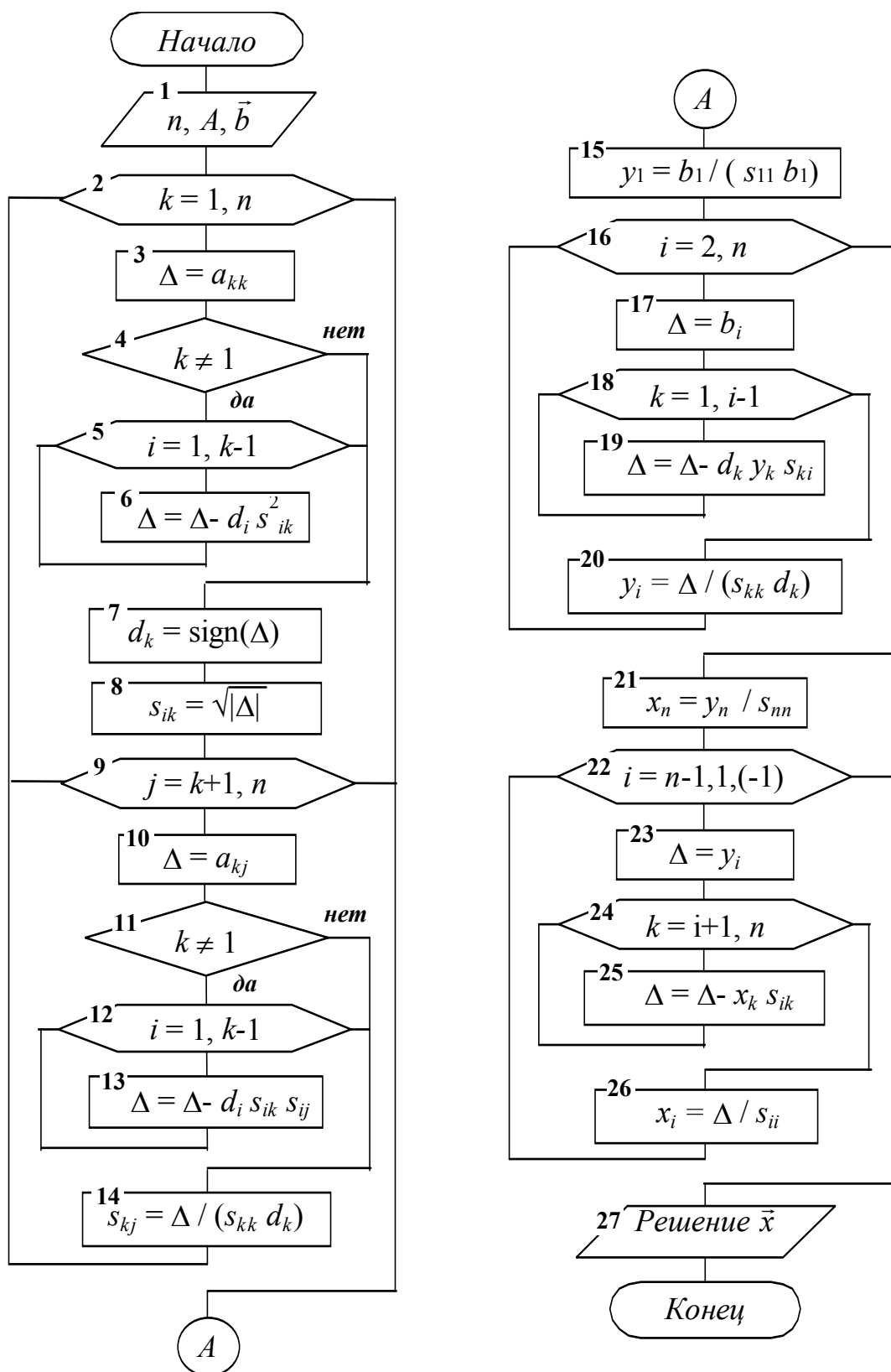


Рис. 10. Блок-схема метода квадратного корня

13.5. Итерационные методы решения СЛАУ

К **приближенным** относятся методы, которые даже в предположении, что вычисления ведутся без округлений, позволяют получить решение системы лишь с заданной точностью. Это итерационные методы, т.е. методы последовательных приближений. К ним относятся методы *простой итерации*, *Зейделя*.

Напомним, что достоинством итерационных методов является их применимость к плохо обусловленным системам и системам высоких порядков, их «самоисправляемость» и простота реализации на ЭВМ. Итерационные методы для начала вычисления требуют задания какого-либо начального приближения к искомому решению.

Следует заметить, что условия и скорость сходимости итерационного процесса существенно зависят от свойств матрицы A системы и от выбора начальных приближений.

Для применения метода итераций исходную систему (13.1) или (13.2) необходимо привести к виду

$$\bar{x} = G\bar{x} + \bar{f} \quad (13.25)$$

и затем итерационный процесс строится по рекуррентным формулам

$$\bar{x}^{(k+1)} = G\bar{x}^{(k)} + \bar{f}, \quad k = 0, 1, 2, \dots \quad (13.26)$$

Для сходимости (13.26) необходимо и достаточно, чтобы $|\lambda_i(G)| < 1$, где $\lambda_i(G)$ – все собственные значения матрицы G .

Чаще всего останавливаются на проверке условий ($\| \dots \|$ – норма матрицы):

$$\|G\| < 1, \quad \|G\| = \max_{1 \leq i \leq n} \sum_{j=1}^n |g_{ij}|, \quad \text{или} \quad \|G\| = \max_{1 \leq j \leq n} \sum_{i=1}^n |g_{ij}|, \quad (13.27)$$

или, если исходная матрица A имеет диагональное представление, т.е.

$$|a_{ii}| > \sum_{i,j=1; i \neq j}^n |a_{ij}|, \quad A = \{a_{ij}\}_1^n. \quad (13.28)$$

Если (13.27) или (13.28) выполняются, метод итерации сходится при любом начальном приближении $\bar{x}^{(0)}$. Чаще всего вектор $\bar{x}^{(0)}$ берут или нулевым, или единичным, или сам вектор \bar{f} из (13.25).

Имеется много подходов к преобразованию исходной системы (13.2) с матрицей A для обеспечения вида (13.25) или условий сходимости (13.27) и (13.28).

Например, (13.25) можно получить следующим образом.

Пусть $A = B + C$, $\det B \neq 0$;
тогда $(B+C)\bar{x} = \bar{b} \Rightarrow B\bar{x} = -C\bar{x} + \bar{b} \Rightarrow B^{-1}B\bar{x} = -B^{-1}C\bar{x} + B^{-1}\bar{b}$,
откуда $\bar{x} = -B^{-1}C\bar{x} + B^{-1}\bar{b}$.

Положив $-B^{-1}C = G$, $B^{-1}\bar{b} = \bar{f}$ и получим (13.25).

Из условий сходимости (13.27) и (13.28) видно, что представление $A = B + C$ не может быть произвольным.

Если матрица A удовлетворяет требованиям (13.28), то в качестве матрицы B можно выбрать нижнюю треугольную

$$B = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ & \cdots & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad a_{ii} \neq 0.$$

Или

$$A\bar{x} = \bar{b}; \quad \Rightarrow \quad A\bar{x} - \bar{b} = 0; \quad \Rightarrow \quad \bar{x} + (A\bar{x} - \bar{b}) = \bar{x}; \quad \Rightarrow$$

$$\bar{x} = \bar{x} + \alpha(A\bar{x} - \bar{b}) = \bar{x} + \alpha A\bar{x} - \alpha \bar{b} = (E + \alpha A)\bar{x} - \alpha \bar{b} = G\bar{x} + \bar{f}.$$

Подбирая параметр α можно добиться, чтобы $\|G\| = \|E + \alpha A\| < 1$.

Если имеет место преобладание (13.28), тогда преобразование к (13.25) можно осуществить просто, решая каждое i -ое уравнение системы (13.1) относительно x_i по следующим рекуррентным формулам:

$$x_i^k = -\frac{1}{a_{ii}} \left[\sum_{j=1, j \neq i}^n a_{ij} x_j^{k-1} \right] = \sum_{j=1}^n g_{ij} x_j^{k-1} + f_i;$$

$$g_{ij} = -a_{ij} / a_{ii}; \quad g_{ij} = 0; \quad f_i = b_i / a_{ii}; \quad (13.29)$$

$$G = \{g_{ij}\}_1^n.$$

Если же в матрице A нет диагонального преобладания, его нужно добиться посредством каких-либо ее линейных преобразований, не нарушающих их равносильности.

Для иллюстрации рассмотрим систему

$$\begin{cases} 2x_1 - 1,8x_2 + 0,4x_3 = 1; & (I) \\ 3x_1 + 2x_2 - 1,1x_3 = 0; & (II) \\ x_1 - x_2 + 7,3x_3 = 0; & (III) \end{cases}$$

Как видно в уравнениях (I) и (II) нет диагонального преобладания, а в (III) есть, поэтому его оставляем неизменным.

Добьемся диагонального преобладания в уравнении (I). Умножим (I) на α , (II) на β , сложим оба уравнения и в полученном уравнении выберем α и β так, чтобы имело место диагональное преобладание:

$$(2\alpha + 3\beta)x_1 + (-1,8\alpha + 2\beta)x_2 + (0,4\alpha - 1,1\beta)x_3 = \alpha.$$

Взяв $\alpha = \beta = 5$, получим $25x_1 + x_2 - 3,5x_3 = 5$.

Для преобразования второго уравнения (II) с преобладанием, (I) умножим на γ , (II) умножим на δ , и из (II) вычтем (I). Получим

$$(3\delta - 2\gamma)x_1 + (2\delta + 1,8\gamma)x_2 + (-1,1\delta - 0,4\gamma)x_3 = -\gamma.$$

Положим $\delta = 2$, $\gamma = 3$, получим $0x_1 + 9,4x_2 - 3,4x_3 = -3$.

В результате получим систему:

$$\begin{cases} 25x_1 + x_2 - 3,5x_3 = 5; \\ 9,4x_2 - 3,4x_3 = -3; \\ x_1 - x_2 + 7,3x_3 = 0. \end{cases}$$

В полученной системе разделим каждое уравнение на диагональный элемент:

$$\begin{cases} x_1 + 0,04x_2 - 0,14x_3 = 0,2; \\ x_2 - 0,36x_3 = -0,32; \\ 0,14x_1 - 0,14x_2 + x_3 = 0. \end{cases} \quad \text{или} \quad \begin{cases} x_1 = -0,04x_2 + 0,14x_3 + 0,2; \\ x_2 = 0,36x_3 - 0,32; \\ x_3 = -0,14x_1 + 0,14x_2. \end{cases}$$

Взяв в качестве начального приближения, например, вектор $\bar{x}^{(0)} = (0,2; -0,32; 0)^T$. Будем решать эту систему по технологии (13.26):

$$\begin{aligned} x_1^{(k+1)} &= -0,04x_2^{(k)} + 0,14x_3^{(k)} + 0,2; \\ x_2^{(k+1)} &= 0,36x_3^{(k)} - 0,32; \\ x_3^{(k+1)} &= -0,14x_1^{(k)} + 0,14x_2^{(k)}. \end{aligned}$$

Процесс вычисления прекращается, когда два соседних приближения вектора решения совпадают по точности, т.е.

$$\left| \bar{x}^{(k+1)} - \bar{x}^{(k)} \right| < \varepsilon.$$

13.6. Метод простой итерации

Технология решения системы вида (13.25) названа методом **простой итерации**.

Оценка абсолютной погрешности для метода простой итерации

$$\left\| \bar{x}^* - \bar{x}^{(k+1)} \right\| \leq \|G\|^{k+1} \cdot \left\| \bar{x}^{(0)} \right\| + \frac{\|G\|^{k+1}}{1 - \|G\|} \cdot \|f\|.$$

Пример. Для иллюстрации алгоритма рассмотрим решение системы (две итерации):

$$\begin{cases} 4x_1 - x_2 - x_3 = 2; \\ x_1 + 5x_2 - 2x_3 = 4; \\ x_1 + x_2 + 4x_3 = 6; \end{cases} \quad A = \begin{bmatrix} 4 & -1 & -1 \\ 1 & 5 & -2 \\ 1 & 1 & 4 \end{bmatrix}; \quad \bar{b} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}. \quad (13.30)$$

Преобразуем систему к виду (13.25) согласно (13.29):

$$\begin{cases} x_1 = (2 + x_2 + x_3)/4; \\ x_2 = (4 - x_1 + 2x_3)/5; \\ x_3 = (6 - x_1 - x_2)/4; \end{cases} \Rightarrow \begin{cases} x_1^{(k+1)} = (2 + x_2^{(k)} + x_3^{(k)})/4; \\ x_2^{(k+1)} = (4 - x_1^{(k)} + 2x_3^{(k)})/5; \\ x_3^{(k+1)} = (6 - x_1^{(k)} - x_2^{(k)})/4. \end{cases} \quad (13.31)$$

Возьмем начальное приближение $\bar{x}^{(0)} = (0; 0; 0)^T$. Тогда для $k=1$ очевидно, что значение $\bar{x}^{(1)} = (0,5; 0,8; 1,5)^T$. Подставим эти значения в (31), т.е. при $k=2$, получим $\bar{x}^{(2)} = (1,075; 1,3; 1,175)^T$.

$$\text{Ошибка } \varepsilon_2 = \max_{1 \leq i \leq 2} |x_i^{(2)} - x_i^{(1)}| = \max(0,575; 0,5; 0,325) = 0,575.$$

Блок-схема алгоритма нахождения решения СЛАУ по методу простых итераций согласно рабочим формулам (13.29) представлена на рис. 2.5 [4].

14. Поиск и сортировка

14.1. Последовательный поиск в массиве

Человеку постоянно приходится сталкиваться с задачами поиска требуемой информации. Алгоритмы поиска, таким образом, являются основными алгоритмами обработки данных при решении как системных, так и прикладных задач.

Типичными примерами может служить работа с тем или иным справочником, телефонной книгой, картотекой в библиотеке и т.д.

Пусть множество из n элементов задано в виде массива $a[n]$.

К наиболее простым задачам поиска можно отнести следующие операции.

1. Найти хотя бы один элемент, равный заданному значению X . В результате необходимо получить i – индекс (порядковый номер) элемента массива, для которого выполняется равенство $a[i]=X$.

2. Найти все элементы, равные заданному значению X . В результате необходимо получить количество таких элементов и их порядковые номера.

Иногда поиск организуется не по совпадению со значением элемента X , а по выполнению некоторых заданных условий. Например, организуется поиск для интервала значений, т.е. когда отыскиваются элементы, удовлетворяющие неравенствам $X1 \leq a[i] \leq X2$, где значения для $X1$ и $X2$ заданы.

Отметим, что операции поиска могут осуществляться с данными, которые находятся в разных состояниях. Эти состояния определяются степенью упорядоченности данных.

Для массива не упорядоченных данных единственный путь для поиска заданного элемента состоит в сравнении каждого элемента массива с заданным и при совпадении некоторого элемента массива с заданным его позиция в массиве фиксируется.

Таким образом, если какая либо дополнительная информация о разыскиваемых данных не задана, то очевиден следующий подход – простой последовательный просмотр массива данных. Такой метод называют линейный (последовательный) поиск.

Рассмотрим реализацию алгоритма последовательного поиска для первой задачи.

Условием окончания поиска является одна из следующих двух ситуаций:

1. Требуемый элемент найден, т. е. условие $a[i] = X$ выполняется.

2. Весь массив просмотрен, но совпадения обнаружено не было, т.е. требуемый элемент отсутствует.

Одним из возможных алгоритмов решения данной задачи может быть:

1. Вводим переменную *flag* логического типа, которая имеет значение «истина» (1), если требуемый элемент в массиве найден и «ложь» (0) в противном случае.

2. *ix* – переменная для номера искомого элемента.

```
flag = «ложь» ;  
ix=0;  
Начало цикла (выполнять для i от 1 до N)  
если a[i] == X  
тогда  
  flag = «истина»;  
  ix=i;  
все  
конец цикла  
если flag==1 тогда:  
  вывод: «элемент найден, его номер=ix»;  
иначе: вывод: «элемент не найден».
```

т.е. если после выполнения алгоритма *flag*=«истина», то элемент нашли, и если в массиве несколько таких элементов, то в переменной *ix* будет храниться

Если же переменная *flag* не изменила своего значения, значит элемента нет, и переменная осталась в нуле.

На практике последовательный поиск требует наибольших временных затрат. Поэтому скорость работы программ, реализующих такой поиск, находится в прямой зависимости от размеров совокупностей данных.

Рассмотренный алгоритм требует просмотра всего массива данных, даже в том случае, если искомый элемент находится в массиве на первом месте.

Для сокращения времени работы программы можно остановить ее работу сразу после того, как элемент будет найден. В этом случае весь массив будет просмотрен только тогда, когда искомый элемент или последний или его вообще нет.

И в этом случае получив следующий алгоритм:

```
flag = «ложь» ;  
ix=0; i=1;  
Начало цикла: выполнять пока (i≤N и a[i]!=X)  
если a[i] == X  
тогда  
  flag = «истина»;  
  ix=i;  
все  
i++;  
конец цикла
```


если $flag == 1$ тогда:

вывод: «элемент найден, его номер= ix »;

иначе: вывод: «элемент не найден».

Окончание цикла гарантировано, так как на каждом его шаге значение параметра цикла увеличивается, и он за конечное число шагов достигнет числа N , или же как только будет найден искомый элемент цикл досрочно закончится.

Из условия цикла следует, что если элемент найден, то это первый из таких элементов.

В этом алгоритме, таким образом на каждом шаге требуется увеличивать индекс и вычислять логическое выражение.

14.2. Барьерный последовательный поиск

Единственная возможность упростить работу и таким образом ускорить поиск – попытаться упростить само логическое выражение, которое состоит из двух частей. И в данном случае единственный путь к более простому решению – сформулировать простое условие, эквивалентное сложному. Но это можно сделать только в том случае, если совпадение гарантировано.

Для этого достаточно в конец массива поместить дополнительный элемент со значением X . Этот вспомогательный элемент называют «барьером», или «часовым» так как он препятствует выходу за пределы массива. Размер исходного массива теперь $N+1$ элемент.

Алгоритм линейного поиска с «барьером» выглядит следующим образом:

$A[N+1] = X;$

$ix = 0; i = 1;$

Начало цикла: выполнять пока $(a[i] \neq X)$

если $a[i] == X$

тогда

$ix = i;$

все

$i++;$

конец цикла

если $i \neq N+1$ тогда:

вывод: «элемент найден, его номер= ix »;

иначе: вывод: «элемент не найден».

Равенство $i = N+1$ свидетельствует о том, что совпадения не произошло (если считать совпадение с «барьером» контрольной точкой для завершения полного цикла).

Для реализации алгоритмов поиска в задачах типа 2 (напомним: нахождение количества элементов, равных значению X). Здесь в любом случае нужно просмотреть весь массив.

Вводим переменную, значение которой будет увеличиваться каждый раз на 1, когда будет найден нужный элемент. Такую переменную называют «счетчиком».

А, для получения индексов искомых элементов, введем массив $b[N]$. Как только будет найден необходимый элемент, его индекс будет заноситься в массив b . В переменной k будет храниться индекс первого свободного места в массиве b .

Изначально $k=1$.

$ix=0$;

Начало цикла (выполнять для i от 1 до N)

если $a[i] == X$

тогда

$b[k]=i$

$k++$;

все

Конец цикла

если $k \neq 1$ тогда:

вывод: «элементы найдены, их номера=»

Начало цикла (выполнять для i от 1 до $k-1$)

Вывод $b[i]$;

Конец цикла.

иначе: вывод: «элементы не найдены»;

В качестве примера рассмотрим задачу нахождения в целочисленной совокупности максимального элемента и определения его местоположения. Для этого необходимо сравнить текущий элемент массива с уже найденным на предыдущем шаге максимальным значением и в случае необходимости изменить текущее максимальное значение и его текущий порядковый номер.

Итак, для хранения текущего максимума введем переменную max а для его индекса переменную k . Алгоритм поиска будет таким.

Предполагаем, что максимальный элемент стоит первым.

Поэтому изначально:

$max=a[1]$;

$k=1$;

Начало цикла (выполнять для i от 2 до N)

Если

Тогда

$max=a[i]$;

$k=i$;

все

Конец цикла.

После выполнения алгоритма переменная \max будет содержать значение максимального элемента, а переменная k – его порядковый номер. Если их несколько, то запомнится первый из них. Если же использовать условие $a[i] \geq \max$, то запомнится последний максимум.

14.3. Сортировка вставками (включениями)

Ранее была сформулирована общая постановка задачи сортировки и рассмотрены методы сортировки выбором. Теперь рассмотрим алгоритмы сортировки при помощи вставки.

Метод 1. Линейная (простая) вставка

Данный метод основан на последовательной вставке элементов в упорядоченный рабочий массив. Линейная вставка чаще всего используется тогда, динамически вносят изменения в массив, все элементы которого известны и который все время должен быть упорядоченным.

Алгоритм линейной вставки следующий. Первый элемент исходного массива помещается в первую позицию рабочего массива. Следующий элемент исходного массива сравнивается с первым. Если этот элемент больше, он помещается во вторую позицию рабочего массива. Если этот элемент меньше, то первый элемент сдвигается на вторую позицию, а новый элемент помещается на первую. Далее все выбираемые из исходного массива элементы последовательно сравниваются с элементами рабочего массива, начиная с первого, до тех пор, пока не встретится элемент больше добавляемого. Тогда этот элемент и все последующие за ним элементы рабочего массива смещаются на одну позицию, освобождая место для нового элемента.

Таким образом, каждый новый элемент $a[i]$ будем вставлять на подходящее место в уже упорядоченную совокупность $a[1], a[2], \dots, a[i-1]$. И это место определяется последовательными сравнениями элемента $a[i]$ с упорядоченными элементами $a[1], \dots, a[i-1]$.

Такой метод сортировки и называли сортировкой простыми вставками. Он, как и выбор требует порядка $(N^2 - N)/2$ операций сравнения.

Метод 2. Центрированная и двоичная вставки

Центральный элемент этого массива часто называют медианой, которая разбивает массив на две ветви – нисходящую (левую) и восходящую (правую).

Алгоритм центрированной вставки можно сформулировать так.

В позицию, расположенную в середине рабочего массива, помещается первый элемент (он и будет медианой). Нисходящая и восходящая ветви имеют указатели, которые показывают на ближайшие к началу и концу занятые позиции. После загрузки первого элемента в центральную позицию оба указателя совпадают и показывают на него. Следующий элемент исходного массива сравнивается с медианой. Если новый элемент меньше, то он размещается на нисходящей ветви, в противном случае – на восходящей ветви. Кроме того, соответствующий конце-

вой указатель продвигается на единицу вниз (нисходящая ветвь) или на единицу вверх (восходящая ветвь).

Каждый последующий элемент исходного массива сравнивается вначале с медианой, а затем с элементами на соответствующей ветви до тех пор, пока не займет нужную позицию.

Метод 3. Двоичная (бинарная) вставка

Здесь для поиска места для вставки элемента $a[i]$ используется алгоритм двоичного (бинарного) поиска. Он сравнивается, вначале с элементом $[i/2]$. Затем, если он меньше сравнивается с элементом $[i/4]$, а если больше: – с $[i/2] + [i/4]$ и т.д. до тех пор, пока для него не найдется место. Все элементы рабочего массива, начиная с позиции вставки и ниже, сдвигаются на одну позицию, освобождая место для i -го элемента.

Количество операций сравнения будет порядка $N \cdot \log_2(N)$.

Вариант алгоритма двоичной вставки

Начало цикла (выполнять для i от 2 до N)

$R=i;$

$L=1$

Начало цикла: выполнять пока $L < R$;

$k=(L+R)/2;$

если $a[k] > a[i]$

тогда

$L=k+1;$

иначе

$R=k;$

все

конец цикла

$k=R;$

$x=a[i];$

Начало цикла (выполнять для t от i до $k+1$ шаг -1)

$a[t] = a[t-1];$

конец цикла

$a[k]=x;$

все

конец цикла

Цикл «**пока $L < R$** » – цикл поиска места вставки. Он основан на методе двоичного поиска.

Цикл «**для t от i до $k+1$ шаг -1** » сдвигает элементы для освобождения места вставки.

Пример. Пусть есть последовательность чисел $\{0, 1, 9, 2, 4, 3, 6, 5\}$.

Сортировать будем по убыванию

Изначально рабочий массив пуст. Размещаем в нем 1. Для простоты будем записывать всю совокупность, а элемент, который вставляем, будет подчеркнут.

Существующий массив отделен вертикальной линией.

Для вставки элемента в нужное место все элементы, стоящие за ним, сдвигаем до позиции, которую занимал вставляемый на данном шаге элемент.

1) 1, | 9, 2, 4, 3, 6, 5, 0

2) 9, 1, | 2, 4, 3, 6, 5, 0

3) 9, 2, 1, | 4, 3, 6, 5, 0

4) 9, 4, 2, 1, | 3, 6, 5, 0

5) 9, 4, 3, 2, 1, | 6, 5, 0

6) 9, 6, 4, 3, 2, 1, | 5, 0

7) 9, 6, 5, 4, 3, 2, 1, | 0

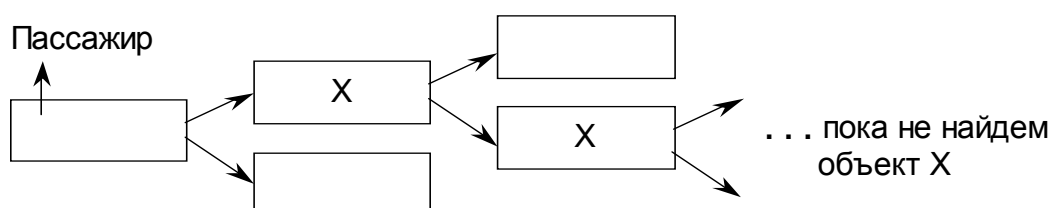
Получим:

8) 9, 6, 5, 4, 3, 2, 1, 0 |

14.4. Двоичный поиск

Приведем пример из жизни [1]. Возникла следующая ситуация. В отделение милиции аэропорта поступило сообщение о том, что некто пытается пронести в самолет бомбу. Известно, что он пока находится в здании аэропорта. Поставлена задача: как можно быстрее определить, у кого из пассажиров находится бомба. Самое простое, но и самое продолжительное по времени решение – это последовательная проверка всех пассажиров, которые находятся в здании.

В такой ситуации лучше поступить так. Размещаем всех пассажиров в две комнаты. Например, служебная собака, реагирующая на взрывчатку сможет указать, в какой из комнат находится бомба. В результате такого действия количество «подозреваемых» уменьшится вдвое. С «подозреваемыми» поступим аналогичным образом: разведем их по двум комнатам и опять количество «подозреваемых» сократится вдвое. Продолжаем поступать так, пока не найдем бомбу. Графически это можно отобразить так:



Такой метод поиска называется двоичным поиском или: бинарный поиск, логарифмический поиск, метод деления пополам, дихотомия.

В программировании такой поиск применяют для нахождения элемента X в отсортированном массиве $a[1..N]$.

Основная идея заключается в следующем.

Пусть массив отсортирован в порядке убывания:

- 1) определяем средний элемент массива $a[m]$, где $m = (n+1)/2$;
- 2) сравниваем его с X , и если $X=a[m]$, то поиск на этом заканчивается;
- 3) если $a[m] < X$, то все элементы от m до n отбрасывают;
- 4) если $a[m] > X$, то отбрасывают элементы от 1 до m ;
- 5) таким образом, каждый раз необходимо пересчитывать левую (L) или правую (R) границы, т.е. изначально $L=1$, $R=n$ (первый шаг); на втором шаге или $L=m$ или $R=m$ т.е. на втором шаге либо левая, либо правая граница меняет свое значение и т.д.;

б) поиск продолжается до тех пор, пока элемент не будет найден: $a[m]=X$, либо пока левая и правая границы поиска не совпадут: ($L=R$), т.е. объект X в массиве отсутствует.

Пусть K – количество операций сравнения, которые необходимы для нахождения элемента в упорядоченном массиве при помощи алгоритма бинарного поиска. Число K определяется из следующего неравенства: $N \leq 2^K$, причем K – минимальное число из всех возможных.

Обычно в математике для определения числа K пользуются функцией \log . Тогда число K будет вычислено по следующей формуле $K = \lceil \log_2 N \rceil + 1$, где квадратные скобки обозначают целую часть числа, находящегося в скобках.

Примеры

1. В массиве a найти элемент $x=1$,
 $a = 9, 6, 5, 4, 3, 2, 1, 0$ ($N=8$)
 $\text{flag} = \text{«ложь»}$
 $L=1; R=8; m_1 = 9 \% 2 = 4$ (в языке Си $\%$ – операция целочисленного деления) $m=4; a[4]=4; 4>1$ $L=5; R=8; \text{flag} = \text{«ложь»}$ Имеем:

3, 2, 1, 0

5 6 7 8

$m_2 = 5 \% 2 = 2$ $m=6; a[6]=2; 2>1$ $\text{flag} = \text{«ложь»}$

Имеем:

$L=7; R=8; m=7; a[7]=1; 1=1$ $1=1$ $\text{flag} = \text{«истина»}$

2. Теперь массив $a=9, 8, 7, 6, 5, 4, 3, 2, 0$

В массиве a опять найти элемент $x=1$,

$\text{flag} = \text{«ложь»}$

$L=1; R=9; m=5; a[5]=5; \text{flag} = \text{«ложь»}$

$L=6; R=9; m=7; a[7]=3; \text{flag} = \text{«ложь»}$

$L=8; R=9; m=8; a[8]=2; \quad \text{flag} = \text{«ложь»}$
 $L=9; R=9; m=9; a[9]=0; \quad \text{flag} = \text{«ложь»}$
 $L=9; R=8; L>R; \text{Все.} \quad \text{flag} = \text{«ложь»}$

Элемент в массиве отсутствует.

Алгоритм имеет следующий вид:

$\text{flag} = \text{«ложь»};$

$L=1;$

$R=N$

Начало цикла: выполнять пока $(L \leq R)$ и $(\text{flag} \neq \text{«ложь»});$

$m=(R+L)/2;$

если $a[m] = X$

тогда

$\text{flag} = \text{«истина»};$

иначе

если $a[m] > X$

тогда

$L=m+1;$

иначе

$R=m+1;$

все

все

конец цикла

Если $(\text{flag} = \text{true})$ то

печатать сообщения: «Элемент X найден, его номер – m »,

иначе

печатать: «Элемента X нет!»,

Можно несколько упростить алгоритм, если, как и в случае линейного поиска, добавить элемент X в исходный массив.

$L=1;$

$R=N+1$

$A[N+1]=X$

Начало цикла: выполнять пока $(L \leq R);$

$m=(R+L)/2;$

если $a[m] = X$

тогда

$\text{flag} = \text{«истина»};$

иначе

если $a[m] > X$

тогда

$L=m+1;$

иначе

$R=m+1;$

все
все

конец цикла

После выполнения цикла номер элемента, равного X хранится в переменной R . Если $R=N-1$, значит совпадения нет.

14.5. Сортировка выбором

Концепция упорядоченного множества элементов имеет существенное значение в разнообразных областях нашей жизни.

В общем, *сортировку* (упорядочение) следует понимать как процесс перегруппировки (перераспределения) заданного множества объектов в некотором определенном порядке. Основная цель сортировки – облегчить поиск элементов в таком упорядоченном множестве.

Мы встречаемся с упорядоченными объектами на складах, в телефонных справочниках, в библиотечных каталогах, в словарях, т.е. почти везде, где нужно осуществлять поиск в массивах объектов.

Задачу сортировки формулирую так.

Пусть надо упорядочить N элементов

$$a_1, a_2, \dots, a_n,$$

которые назовем объектами. Каждый объект a_i имеет свой ключ k_i , который и управляет процессом сортировки

Задача сортировки – найти такую перестановку объектов $p(1)p(2)...p(n)$, после которой ключи объектов расположились бы например так:

$$K_{p(1)} \leq K_{p(2)} \leq \dots \leq K_{p(n)}.$$

Порядок, при котором в массиве объектов первым элементом является объект с наименьшим ключом, а каждое следующее значение больше предыдущего, и наконец последнее – самое большое из данного массива называют *возрастающим*.

Примером задачи сортировки может служить следующая формулировка.

В массиве из N различных целых чисел осуществить их перестановку так, чтобы после перестановки в массиве первым элементом было самое большое число. Каждое следующее должно быть меньше предыдущего, и последнее – наименьшее из чисел массива. Такой порядок расположения чисел называют *убывающим*.

Сортировка – хороший пример комплекса задач, которые решают при помощи большой совокупности разнообразных алгоритмов. Каждый из алгоритмов имеет и свои достоинства, и свои недостатки. Выбирать конкретный алгоритм рекомендуется исходя из конкретной постановки задачи.

Выбранный метод сортировки должен экономно использовать оперативную память компьютера. А это значит, что перестановки, приводящие элементы в требуемый порядок, должны выполняться по возможности на том же участке

памяти. Поэтому методы, в которых элементы из массива *a* пересылаются в результирующий массив *b* используют только в исключительных случаях.

Существенным критерием выбора конкретного метода сортировки из существующих является и его экономичность, т. е. время работы. Поэтому одним из критериев эффективности алгоритмов сортировки является число необходимых сравнений элементов.

Таким образом, основное требование к методам сортировки – экономное использование времени процессора и памяти. Например, хорошими алгоритмами считаются те, которые затрачивают на сортировку *N* объектов время порядка $N \cdot \log_2(N)$.

При сортировке либо перемещаются сами объекты, либо создается вспомогательная таблица, которая описывает перестановку и обеспечивает доступ к объектам в соответствии с порядком их ключей.

Обычно сортировку подразделяют на два класса: *внутреннюю*, когда все объекты хранятся в оперативной памяти, и *внешнюю*, когда они там все не помещаются. Мы рассмотрим только внутреннюю сортировку.

Существующие методы сортировки обычно разбивают на три класса, в зависимости от лежащего в их основе приема:

- а) сортировка выбором;
- б) сортировка обменами;
- в) сортировка включения (вставками).

Рассмотрим основные алгоритмы сортировки на примере целочисленного массива.

14.6. Алгоритмы сортировки выбором

Метод 1. Простой линейный выбор

Метод предполагает использование рабочего (дополнительного) массива, в который в конечном итоге помещается отсортированный массив. Количество просмотров определяется количеством элементов массива. Сортировка посредством простого линейного выбора сводится к следующему:

1. Найти наименьший (наибольший) элемент, переслать его в рабочий массив и заменить его в исходном массиве величиной, которая больше (меньше) любого реального элемента.

2. Повторить шаг 1. На этот раз будет выбран наименьший (наибольший) из оставшихся элементов.

3. Повторять шаг 1 до тех пор, пока не будут выбраны все *N* элементов.

Программа, реализующая алгоритм простого линейного выбора может быть следующей:

```
#include <iostream.h>
#include <conio.h>
#include <limits.h>
// Файл с декларацией max(min) целого: INT_MAX (INT_MIN)
```

```

void Sort_Lin(int *, int);
void main(void)
{
    int *a, n, i;
    cout << "\n Input N: ";
    cin >> n;
    a = new int [n];    // Захватываем память для основного массива
    cout << "\n Array ? "<<endl;
    // Формируем исходный массив
    for ( i=0; i<n; i++)
    {
        cout<<"a["<<i<<"]=""; cin >> a[i];
    }
    Sort_Lin ( a, n ); // Вызвали сортировку
    // и распечатали отсортированный массив
    cout << "\n Sort Array : ";
    for ( i=0; i<n; i++) cout << a[i] << " ";
    cout << endl;
    delete [ ]a;
    getch();
}

```

//----Функция, реализующая простой линейный выбор -----

```

void Sort_Lin(int *a, int n)
{
    int i, j, imn, amin, *p;
    p = new int [n];    // Захватываем память для рабочего массива
    for ( j=0; j<n; j++) // Внешний цикл по j
    {
        for ( amin=INT_MAX, i=0; i<n; i++) // Внутренний цикл по i
            if ( a[i] < amin )
            {
                imin = i;
                amin = a[i];
            }
        p[j] = amin; // Найденный текущий min в рабочий массив
        a[imin] = INT_MAX; // а на его место max_допустимое целое
    }
    for ( j=0; j<n; j++)
        a[j] = p[j]; // Отсортированный массив на место исходного
    delete [ ]p;
}

```

В результате получим:

Input N: 5

Array ?

```
a[0]=4
a[1]=1
a[2]=8
a[3]=5
a[4]=3
Sort Array : 1 3 4 5 8
Press any key to continue
```

Здесь, внешний **цикл по j** осуществляет формирование рабочего отсортированного массива. последовательно укорачивая исходный массив записывая на место текущего максимально допустимое целочисленное значение INT_MAX., А внутренний **цикл по i** осуществляет поиск min (и определяет его порядковый номер imin) в текущей рабочей части массива, который будет максимальным по отношению к уже сформированной по возрастанию части рабочего массива.

После выхода из внешнего цикла все элементы исходного массива a имеют значение равное INT_MAX. Поэтому последний цикл переписывает отсортированные значения рабочего массива в исходный, после чего рабочий массив уничтожаем.

Метод 2. Линейный выбор с обменом (сортировка выбором)

Рабочий массив здесь не используется.

Рассмотрим принципы, на которых основан этот метод сортировки.

Например, нужно отсортировать массив по убыванию.

1. Выбирается максимальный элемент.
2. Он меняется местами с первым элементом a[1]. На первом месте оказывается максимальный элемент.
3. Далее рассматривается только не отсортированная часть массива, и этот процесс повторяется с оставшимися N-1, N-2 элементами и т. д. до тех пор, пока не останется один, наименьший элемент.

Рассмотрим работу данного метода на примере целочисленного массива a={0, 1, 9, 2, 4, 3, 6, 5}.

Максимальный элемент на текущем шаге подчеркиваем, а рабочую часть отделять вертикальной линией. Итак:

- 1) | 0, 1, 9, 2, 4, 3, 6, 5 — выбрали, поменяли с первым (с 1-м)
- 2) 9, | 1, 0, 2, 4, 3, 6, 5 — выбрали, поменяли с первым оставшейся части массива
- 3) 9, 6, | 0, 2, 4, 3, 1, 5 — и т.д
- 4) 9, 6, 5, | 2, 4, 3, 1, 0
- 5) 9, 6, 5, 4, | 2, 3, 1, 0
- 6) 9, 6, 5, 4, 3, 2, | 1, 0
- 7) 9, 6, 5, 4, 3, 2, 1, | 0
- 8) 9, 6, 5, 4, 3, 2, 1, 0

Подсчитаем количество сравнений, которые пришлось сделать для упорядочения массива.

На первом шаге для нахождения максимального элемента необходимо $N-1$ сравнение, на втором – $N-2$, на третьем – $N-3, \dots$, на последнем шаге – одно сравнение. Найдем сумму:

$$N-1+N-2+N-3+\dots+1=(N^2-N)/2;$$

Сумму можно посчитать и исходя из следующих соображений. Количество слагаемых равно $N-1$, сумма первого и последнего, второго и предпоследнего и т. д. равна N . Произведение $N(N-1)$ даст удвоенную сумму, т. к. каждое слагаемое входит дважды, поэтому его и нужно разделить на 2. (Для нашего примера имеем: $(8^2-8)/2=28$ сравнений).

Алгоритм метода будет следующим:

Начало цикла 1 (выполнять для i от 1 до $N-1$)

$k=i;$

$max=a[i];$

Начало цикла 2 (выполнять для j от $i+1$ до N)

если $a[j] > max$, тогда

$max=a[j];$

$k=j;$

все

конец цикла 2;

$a[k]=a[i];$

$a[i]=max;$

конец цикла 1.

Внутренний цикл является ничем иным, как стандартным алгоритмом поиска максимального элемента в массиве.

14.7. Сортировка обменом

Метод 1. Стандартный обмен (метод "пузырька")

Рассмотрим еще один метод сортировки, который основан на следующих принципах.

Пусть необходимо отсортировать массив по убыванию.

1. Сравниваем первые два элемента. Если первый меньше второго, то меняем их местами.

2. Сравниваем второй и третий, третий и четвертый, ..., предпоследний и последний, при необходимости меняем их местами. В конечном итоге наименьший окажется на последнем месте.

3. Снова просматриваем массив с его начала, уменьшив на единицу количество просматриваемых элементов (т.е. как раньше упорядоченную часть массива больше не рассматриваем).

4. Массив будет отсортирован после просмотра, в котором приняли участие только первый и второй элементы.

Если:

- В этом случае при каждом просмотре один пузырек как бы поднимается до уровня, соответствующего его массе. Такой метод широко известен под названием «пузырьковая сортировка».

Обозначим:

k – номер просмотра

N-k – номер последней пары

Дано: а= 5 4 8 2 9, N=5 Сортируем по возрастанию

(k=1, L=5)

> меняем

< не меняем

> меняем

< не меняем

Второй просмотр: рассматриваем часть массива с первого до четвертого элемента.

< не меняем

> меняем

< не меняем

Третий просмотр: рассматриваемая часть массива содержит три первых элемента.

> меняем

205

< не меняем

5 стоит на своем месте.

Четвертый просмотр: рассматриваем последнюю пару.

(k=4, L=2)

i=1 2 4 | 5 8 9

< не меняем

4 стоит на своем месте.

Для самого маленького элемента (2) остается только одно место – первое.

А так будет выглядеть сортировка по убыванию массива, отсортированного выбором в [1].

Пример 2.

a={0, 1, 9, 2, 4, 3, 6, 5}.

1) 1, 9, 2, 4, 3, 6, 5, | 0

2) 9, 2, 4, 3, 6, 5, | 1, 0

3) 9, 4, 3, 6, 5, | 2, 1, 0

4) 9, 4, 6, 5, | 3, 2, 1, 0

5) 9, 6, 5, | 4, 3, 2, 1, 0

6) 9, 6, | 5, 4, 3, 2, 1, 0

7) 9, | 6, 5, 4, 3, 2, 1, 0

Число сравнений в данном алгоритме равно $(N^2-N)/2$.

Алгоритм метода имеет вид:

Начало цикла 1 (выполнять для i от 1 до N-1)

Начало цикла 2 (выполнять для j от 1 до N-1)

если $a[j] > a[j+1]$, тогда

$x = a[j];$

$a[j] = a[j+1];$

$a[j+1] = x;$

все

конец цикла 2;

конец цикла 1.

В последнем примере последних два прохода не влияют на порядок расположения элементов из-за того, что они уже отсортированы. Оптимизируем алгоритм – можем запомнить, были или не были перестановки в процессе некоторого прохода. И если перестановок не было, то работу можно заканчивать.

Можно сделать еще одно улучшение, если запомнить не только сам факт, что обмена не было, но и положение (индекс i1) последнего обмена. Все пары соседних элементов после этого индекса уже отсортированы (следуют в требуемом порядке). Поэтому просмотр можно завершить.

Как и раньше введем логическую переменную-флажок flag для контроля: был обмен или нет и переменную i1 для запоминания индекса последнего обмена.

Еще нужна одна переменная R – это будет граница, на которой заканчиваем просмотр

```

flag = «истина»;
i1=N-1;
Начало цикла: выполнять пока flag = «истина»;
    flag = «ложь»;
    R=i1;
Начало цикла (выполнять для j от 1 до R)
если a[j] > a[j+1]
тогда
    x= a[j];
    a[j]= a[j+1];
    a[j+1]=x;
    flag = «истина»;
                                i1=j;
                                все
конец цикла
конец цикла

```

Приведем один из вариантов программы:

```

#include<iostream.h>
#include<conio.h>
#include<stdio.h>
void main()
{
    int *a, i, j, n, r;
    cout << "\n n-> "; cin >> n;
    a=new int[n];
    cout << "\n Input array\n";
    for (i=0; i<n; i++)
    {
        printf("\n a[%d] = ", i+1);
        cin >> a[i];
    }
    for (i=0 ; i<n; i++)
        for (j=0; j<(n-1)-i; j++)
            if(a[j]<a[j+1])
            {
                r=a[j];
                a[j]=a[j+1];
                a[j+1]=r;
            }
    cout << "\n Sort Array \n";
    for ( i=0; i<n; i++)

```

```

        cout << " " << a[i];
    cout<<endl;
    delete [ ]a;
    getch();
}

```

Метод 2. Шейкерная сортировка

Если внимательно проанализировать «пузырьковую сортировку», можно заменить некоторую своеобразную асимметрию. Самый легкий пузырек занимает свое место за один проход, а самый тяжелый будет просачиваться на свое место на один шаг при каждом просмотре. Иными словами: всплывает пузырек сразу, за один проход, а тонет очень медленно, за один проход на одну позицию. Это наводит на мысль чередовать направление последовательных просмотров. Такая сортировка называется «шейкерной». Рассмотрим ее работу на том же массиве $A=\{0, 1, 9, 2, 4, 3, 6, 5\}$ (L – левая граница просмотра, R – правая).

- | | |
|---------------------------|----------------|
| 1) 1, 9, 2, 4, 3, 6, 5, 0 | ($L=1, R=7$) |
| 2) 9, 1, 6, 2, 4, 3, 5, 0 | ($L=2, R=7$) |
| 3) 9, 6, 2, 4, 3, 5, 1, 0 | ($L=2, R=6$) |
| 4) 9, 6, 5, 2, 4, 3, 1, 0 | ($L=3, R=6$) |
| 5) 9, 6, 5, 4, 3, 2, 1, 0 | ($L=3, R=5$) |
| 6) 9, 6, 5, 4, 3, 2, 1, 0 | ($L=4, R=5$) |

Алгоритм данной сортировки предлагаем написать самостоятельно.

Все эти улучшения сокращают количество операций сравнения для частых случаев, однако при неблагоприятной начальной расстановке элементов массива (подумайте какой) придется проделать все $(N^2-N)/2$ операций сравнения.

Метод 3. Челночная сортировка

Челночная сортировка, называемая также «сортировкой просеиванием» или «линейной вставкой с обменом» является наиболее эффективной из всех рассмотренных выше методов и отличается от сортировки обменом тем, что не сохраняет фиксированной последовательности сравнений.

Алгоритм челночной сортировки действует точно также как стандартный обмен до тех пор, пока не возникает необходимость перестановки элементов исходного массива. Сравнимый меньший элемент поднимается, насколько это возможно, вверх. Этот элемент сравнивается в обратном порядке со своими предшественниками, по направлению к первой позиции. Если он меньше предшественника, то выполняется обмен и начинается очередное сравнение. Когда элемент, передвигаемый вверх, встречает меньший элемент, этот процесс прекращается и нисходящее сравнение возобновляется с той позиции, в которой выполнялся первый обмен.

Сортировка заканчивается, когда нисходящее сравнение выходит на границу массива.

Процесс челночной сортировки можно проследить на следующем примере:

Исходный массив: 2 7 9 5 4

Сортируем по возрастанию

Нисходящие сравнения: 2 7; 7 9; **9 5**;

После перестановки: 2 7 **5 9** 4

Восходящие сравнения и обмен: 7 5 -> 5 7; 2 5 (встретил меньший элемент) – конец восходящего сравнения; получен массив 2 5 7 9 4

Нисходящие сравнения: **9 4**

После перестановки: 2 5 7 4 **9**

Восходящие сравнения и обмен: 7 4 -> 4 7; 5 4 -> 4 5; 2 4 – конец восходящего сравнения;
получен массив 2 4 5 7 9.

Сортировка выбором наилучшего (наименьшего, наибольшего) элемента

Идея метода – проста. Просматривается последовательно весь массив и *i*-тый элемент сравнивается со всеми, следующими за ним. Найдя наилучший, переставляем местами его и *i*-ый. После перебора всех элементов, массив оказывается отсортирован.

Внешний цикл изменяется от $i=0$ до $i<(n-1)$, внутренний цикл изменяется от $j=i$ до $j<n$.

Программа сортировки целых чисел по убыванию может иметь вид:

```
...
void main()
{
    int *a, r, i, j, n, max, i_max;
    cout << "\n Size : "; cin >> n;
    a=new int[n];
    for ( i=0; i<n; i++)
        cin >> a[i];
    for ( i=0; i<(n-1); i++)
    {
        max=a[i]; i_max=i;
        for ( j=i+1; j<n; j++)
            if( max<a[j] )
            {
                max=a[j]; i_max=j;
            }
        r=a[i]; a[i]=max; a[i_max]=r;
    }
    for(i=0;i<n;i++)
        printf("\n %d",a[i]);
    delete [ ]a;
    getch();
}
```

14.8. Быстрая сортировка

Все рассмотренные ранее методы обмена не столь эффективны при реализации на ЭВМ из-за сравнений и возможных обменов только соседних элементов. Лучшие результаты дают методы, в которых пересылки выполняются на большие расстояния. Рассмотрим два наиболее быстрых алгоритмов.

Метод Хоара

Метод Хоара (Charles Antony Richard Hoare, 1962 г.), называемый также методом быстрой сортировки (Quicksort) основывается на следующем: находится такой элемент, который разбивает множество на два подмножества так, что в одном все элементы больше, а в другом – меньше делящего. Каждое из подмножеств также разделяется на два, по такому же признаку. Конечным итогом такого разделения станет рассортированное множество. Рассмотрим один из вариантов реализации сортировки Хоора.

В самой процедуре сортировки сначала выберем средний элемент. Потом, используя переменные i (цикл $i=0, i++$) и j (цикл $j=n-1, j--$), пройдемся по массиву, отыскивая в левой части элементы больше среднего, а в правой – меньше среднего. Два найденных элемента переставим местами. Будем действовать так, пока i не станет больше j . Тогда мы получим два подмножества, ограниченные с краев индексами l и r , а в середине – j и i . Если эти подмножества существуют (т.е. $i < r$ и $j > l$), то выполняем их сортировку.

Например, необходимо рассортировать массив: 13, 3, 23, 19, 7, 53, 29, 17. Переставляемые элементы будем подчеркивать, средний – выделим жирным шрифтом (19), а элемент, попавший на свое место и не участвующий в последующих сравнениях – наклонным шрифтом. Индекс i будет задавать номер элемента слева от среднего, а j справа от среднего.

1) 13 3 23 **19** 7 53 29 17
 $i \rightarrow \dots \quad \leftarrow j$

ищем в левой части элемент больший среднего, а в правой – меньший среднего, это 23 и 17, переставляем их;

2) 13 3 17 **19** 7 53 29 23 – переставили (19 и 7);

3) 13 3 17 7 | 19 53 29 23

справа все большие среднего (19), слева – меньшие, теперь делим совокупность на два подмножества:

----- A -----	----- B -----
<u>13</u> 3 17 7	19 <u>53</u> 29 <u>23</u>
---A ₁ --- -----A ₂ -----	19 23 29 53 – здесь
Выделяем подмножество	3 13 17 7
13 <u>17</u> <u>7</u>	
Выделяем подмножество	<u>13</u> <u>7</u> 17
Результат:	3 7 <u>13</u> 17 19 23 29 53
Как искать точку деления:	
19 53 24 <u>23</u>	
i=left	j=right

19 < 23 – обмен не требуется: $j = j-1$
 19 53 24 23
 19 < 24 – обмен не требуется: $j = j-1$
 19 53 24 23

Метод Шелла

Сортировка Шелла (Donald Lewis Shell), предложенная в 1959 г., называется также «слиянием с обменом» и является расширением челночной сортировки.

Основная идея алгоритма состоит в том, что на начальном этапе реализуется сравнение и перемещение далеко отстоящих друг от друга элементов. Интервал между сравниваемыми элементами (h) постепенно уменьшается до единицы, что приводит к перестановке соседних элементов на последних стадиях сортировки (если это необходимо).

Реализуем метод Шелла следующим образом. Начальный шаг сортировки примем равным $h=n/2$, т.е. 0,5 от общей длины массива, и после каждого прохода будем уменьшать его в два раза. Каждый этап сортировки включает в себя проход всего массива и сравнение отстоящих на h элементов ($i=0$; $j=n/2$; $i++$; $j++$; до тех пор, пока $j<n$). Проход с тем же шагом повторяется, если элементы переставлялись. Заметим, что после каждого этапа отстоящие на h элементы отсортированы.

Таким образом, вначале сравниваются (и, если нужно, переставляются) далеко стоящие элементы, а на последнем проходе – соседние элементы

Рассмотрим пример. Рассортировать массив: 41, 53, 11, 37, 79, 19, 7, 61.

В строке после массива в круглых скобках указаны индексы сравниваемых элементов и указан номер внешнего цикла.

Цикл	0	1	2	3	4	5	6	7	– индексы элементов
	41	53	11	37	79	19	7	61	– исходный массив
1-ый	41	19	11	37	79	53	7	61	– (0,4),(1,5)
	41	19	7	37	79	53	11	61	– (2,6),(3,7)
2-ой	7	19	41	37	11	53	79	61	– (0,2),(1,3),(2,4),(3,5),(4,6),(5,7)
	7	19	11	37	41	53	79	61	– (0,2),(1,3),(2,4),(3,5),(4,6),(5,7)
3-ий	7	11	19	37	41	53	61	79	– сравнивались соседние

Алгоритм метода Шелла

Внешний цикл (1): от $h=n/2$ до $h>0$; $h=h/2$

Вложенный внутренний цикл (2): do (выполнять пока $end_sort = 1$):

1) $end_sort = 0$

2) цикл (3): для i от 0 до h ; j от h ; до тех пор, пока $j<n$; $i++$, $j++$
 выполнить сравнение:

если $x[i] > x[j]$ то переставить местами, т.е.

$buf = x[j]$, $x[j] = x[i]$, $x[i] = buf$;

изменить признак: `end_sort = 1;`

3) конец цикла (3)

Конец циклов (2 и 1): выполнять пока `end_sort = 1`, каждый выход будет уменьшать `h` в два раза.

Выигрыш получается за счет того, что на каждом этапе сортировки либо участвует сравнительно мало элементов, либо эти элементы уже довольно хорошо упорядочены, и требуется небольшое количество перестановок. Последний просмотр сортировки Шелла выполняется по тому же алгоритму, что и в челночной сортировке. Предыдущие просмотры подобны просмотрам челночной обработки, но в них сравниваются не соседние элементы, а элементы, отстоящие на заданное расстояние h . Большой шаг на начальных этапах сортировки позволяет уменьшить число вторичных сравнений на более поздних этапах.

При анализе данного алгоритма возникают достаточно сложные математические задачи, многие из которых еще не решены. Например, неизвестно, какая последовательность расстояний дает лучшие результаты, хотя выяснено, что расстояния h_i не должны быть множителями один другого.

Сортировка методом Шелла может быть реализована, например, в следующем виде:

```
#include <iostream.h>
#include <conio.h>
void sort_Sh ( int *x, int n)
{
    int i, j, buf, h, end_sort;
    for( h = n/2; h>0; h/=2)
        do
        {
            end_sort = 0;
            for( i=0, j=h; j<n; i++, j++)
                if( *(x+i)> *(x+j) )           // Сравниваем отстоящие на h элементы
            {
                buf = *(x+j);           // Переставляем элементы
                *(x+j) = *(x+i);
                *(x+i) = buf;
                end_sort = 1;           // Есть еще не рассортированные данные
            }
        } while (end_sort);           // Окончание этапа сортировки
}
void main()
{
    int x[ ] = { 41, 53, 11, 37, 79, 19, 7, 61 }; // задали явно
    sort_Sh(x,8);
    cout<<"Array: ";
    for (int i=0; i<8; i++)
        cout << x[i] << " ";
    cout<<endl;
}
```

```

getch();
}

```

В результате получим:

Array: 7 11 19 37 41 53 61 79

15. Обработка статистических данных

Генеральная совокупность – множество статистических данных, количество которых достаточно велико. Получить необходимые сведения о генеральной совокупности, не обрабатывая всех ее данных можно, используя **выборочный метод** [5].

Предположим, что генеральная совокупность – случайная величина X , имеющая:

- плотность вероятности $f(x)$;
- функцию распределения $F(x)$;
- математическое ожидание $M[X]$;
- дисперсию $D[X]$.

Но ничего из приведенного выше мы не знаем, а знаем только значения элементов, составляющих эту генеральную совокупность.

Выборочный метод заключается в том, что из генеральной совокупности, объем которой стремится к бесконечности (в общем случае), производится случайный выбор некоторого небольшого количества данных, называемый **выборка** – $\{x_i\}$ ($i=1, \dots, N$). **Объем выборки** – значение N (количество данных в выборке).

Выборка должна быть «представительской», т.е. хорошо описывать поведение исследуемой случайной величины X .

Далее строится вариационный ряд, т.е. данные выборки упорядочивают по возрастанию. Эти данные, называемые **вариантами**, могут быть сведены в таблицу, данные которой понадобятся в дальнейшем:

x_i	x_1	x_2	x_3	...	x_n	$x > x_n$
kol_i	kol_1	kol_2	kol_3		kol_n	–
p^*_i	p^*_1	p^*_2	p^*_3		p^*_n	–
$F^*(x)$	0	p^*_1	$p^*_1 + p^*_2$		$p^*_1 + \dots + p^*_{n-1}$	1

Некоторые данные в выборке могут повторяться, поэтому $n \leq N$.

Частоту повторения варианты x_i в выборке обозначим kol_i – (чаще = 1), сумма всех $kol_i = N$ ($i=1, \dots, N$).

Частость (относительную частоту), т.е. отношение частоты к объему выборки можем принять за приближенное значение вероятности соответствующей варианты, т.е. $p^*_i = kol_i/N$ ($i=1, \dots, N$).

Причем, сумма всех $p^*_i = 1$ ($i = 1, \dots, N$).

Расчет эмпирической функции распределения $F^*(x)$

При $N \rightarrow \infty$ эмпирическая функция распределения по вероятности сходится к теоретической функции распределения $F(x)$. Эмпирическая функция распределения находится по формуле:

$$F^*(x) = \text{сумме } p^*_i \text{ при } X < x_i, \text{ где } p^*_i = \text{kol}_i/N,$$

т.е. равна количеству значений в вариационном ряду, строго меньших x , разделенному на N .

$$F^*(x) = 0 \text{ при } x \leq x_1$$

$$F^*(x) = p^*_1 \text{ при } x < x_2$$

$$F^*(x) = p^*_1 + p^*_2 \text{ при } x < x_3$$

...

$$F^*(x) = p^*_1 + \dots + p^*_{n-1} \text{ при } x < x_{n-1}$$

$$F^*(x) = 1 \text{ при } x > x_n$$

Полученные данные можно добавить в таблицу.

Оценки числовых характеристик

1. Несмещенная состоятельная оценка математического ожидания, которая называется выборочное среднее, вычисляется по формуле (2.8) [5]:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i.$$

2. Несмещенная состоятельная оценка дисперсии при неизвестном математическом ожидании вычисляется по формуле (2.9) [5]:

$$S_0^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2.$$

3. Смещенная состоятельная оценка дисперсии вычисляется по формуле (2.10) [5]:

$$S^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2.$$

4. Очевидно, что состоятельная оценка среднеквадратического отклонения:
 $S_0 = \sqrt{S_0^2}.$

Гистограмма распределения

Гистограммой называется оценка плотности вероятности, т.е. приближенное значение $f(x)$, поэтому ее будем обозначать $f^*(x)$. Наиболее часто используется равноинтервальная гистограмма.

Равноинтервальная гистограмма

При построении гистограммы отрезок $[x_1, x_n]$ разбивают на m интервалов:

$$m = (\text{int})\sqrt{N}, \text{ для } N \leq 100.$$

Находят шаг $h = (x_n - x_1)/m$ и для удобства строят таблицу

Интервалы	$[x^*_1, x^*_2]$	$[x^*_2, x^*_3]$...	$[x^*_m, x^*_{m+1}]$
kol^*_i	kol^*_1	kol^*_2		kol^*_m
f^*_i	f^*_1	f^*_2		f^*_m

Очевидно, что $x^*_i = x_1 + h \cdot (i-1)$, а kol^*_i – количество элементов выборки, попавших в i -тый интервал.

При построении гистограммы на каждом интервале строится прямоугольник с основанием h и высотой

$$f^*_i = kol^*_i / (N \cdot h),$$

где f^*_i – средняя плотность вероятности.

Равновероятностная гистограмма

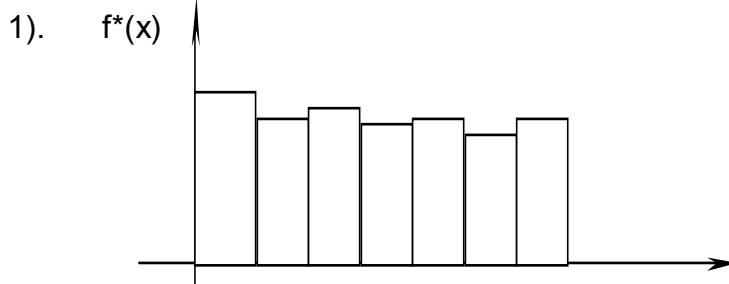
В равновероятностной гистограмме шаги рассчитываются из условия, чтобы в каждый интервал попало одинаковое число значений, равное m .

Для построения равновероятностной гистограммы на каждом интервале строится прямоугольник с основанием h_i и высотой $f^*_i = m / (N \cdot h_i)$.

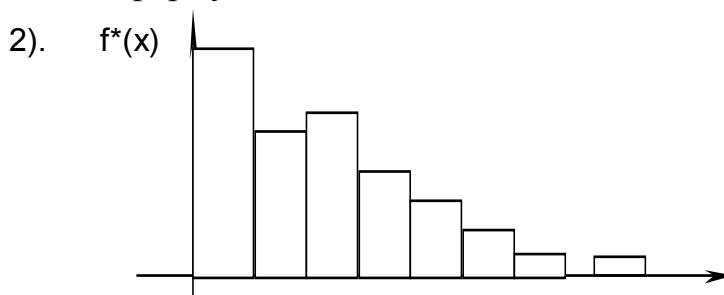
Например, значения 2, 3, 3, 3, 4; распределим 3 равномерно на интервале: шаг = $(4-2)/(кол-1)$;

По виду гистограммы выдвигается гипотеза о законе распределения случайной величины X .

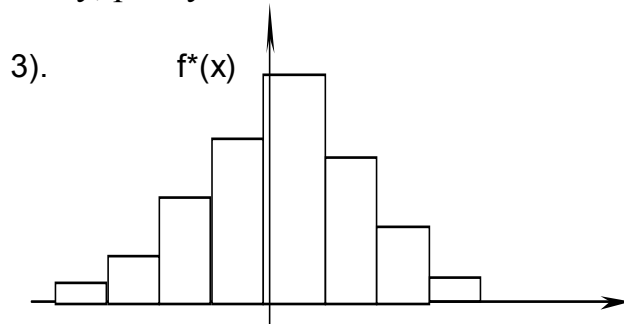
Например



По данной гистограмме выдвигаем гипотезу о равномерном законе распределения $R(\hat{a}, \hat{b})$, имеющем два параметра a и b , в данном случае – их оценки, вычисляемые по формулам $\hat{a} = \bar{x} - \sqrt{3S^2}$, $\hat{b} = \bar{x} + \sqrt{3S^2}$.



По данной гистограмме, если все $x_i > 0$, выдвигаем гипотезу об экспоненциальном законе распределения $E(1/\bar{x})$, имеющем один параметр λ , в данном случае – его оценку, равную $1/\bar{x}$.



По данной гистограмме выдвигаем гипотезу о нормальном законе распределения $N(\bar{x}, S_0)$, имеющем два параметра μ – математическое ожидание, σ – среднеквадратическое отклонение, в нашем случае – их оценки.

Критерий согласия "Хи-квадрат"

Проверяем выдвинутую гипотезу

$$H_0 = f^*(x) = f_0(x);$$

$$H_1 = f^*(x) \neq f_0(x);$$

где $f_0(x)$ – плотность вероятности гипотетического (предполагаемого) закона распределения.

Вычисляем значение критерия по формуле (2.25) [5]:

$$\chi^2 = N \sum_{i=1}^k \frac{(p_i - p_i^*)^2}{p_i} = \sum_{i=1}^k \frac{(kol_i - N p_i)^2}{N p_i}$$

где p_i – теоретические вероятности, которые рассчитываются по следующим формулам [5]:

1) равномерный закон $p_i = (x_{i+1}^* - x_i^*) / (x_n - x_1)$ – формула (2.27);

2) экспоненциальный закон $p_i = \exp(-x_i^* / \bar{x}) - \exp(-x_{i+1}^* / \bar{x})$ x_i^* – формула (2.26); причем $x_1^* = 0$, $x_{m+1}^* = +\infty$.

3) нормальный закон $p_i = 0,5 \left[\Phi\left(\frac{x_{i+1}^* - \bar{x}}{S_0}\right) - \Phi\left(\frac{x_i^* - \bar{x}}{S_0}\right) \right]$ – формула (2.28);

причем $x_1^* = -\infty$, $x_{m+1}^* = +\infty$. Значения функции (интеграла) Лапласа $\Phi(x)$ можно рассчитать, используя методы приближенного интегрирования, или – по таблицам. Напомним, что $\Phi(-x) = -\Phi(x)$, а для $x > 4$ $\Phi(x) = 1$ (или 0,5, в зависимости от множителя перед интегралом – 1(2)/на корень из Pi^* сигма).

Из таблицы функции «Хи-квадрат» для уровня значимости $\alpha = 0,05$ и k – число степеней свободы

$$k = m - 1 - s,$$

где m – число интервалов, s – число параметров распределения, выбираем соответствующие значения:

$$X^2(0,05; 8) = 15,51;$$

$$X^2(0,05; 7) = 14,07.$$

Если расчетное (полученное нами) значение $X^2 < X^2(0,05; k)$, то выдвинутая гипотеза H_0 – принимается, хотя она может быть и не верна. В противном случае, гипотеза отвергается.

Критерий согласия Колмогорова

Проверяем выдвинутую гипотезу:

$$H_0 = F^*(x) = F_0(x);$$

$$H_1 = F^*(x) \neq F_0(x);$$

где $F_0(x)$ – теоретическая функция распределения, значения которой рассчитываются по следующим формулам [5]:

1) равномерный закон формула (2.30):

$$F_0(x) = \begin{cases} 0; & x < x_1; \\ (x - x_1) / (x_n - x_1); & x \in [x_1, x_n]; \\ 1; & x \geq x_n. \end{cases}$$

2) экспоненциальный закон формула (2.31):

$$F_0(x) = 1 - \exp(-x / \bar{x}); x \geq 0$$

3) нормальный закон формула (2.28):

$$F_0(x) = 0,5 + 0,5 \cdot \Phi\left(\frac{x - \bar{x}}{S_0}\right).$$

Рассчитывают 10-20 значений, по ним строят график теоретической функции там же, где построена эмпирическая функция.

Вычисляют значение критерия Колмогорова

$$\lambda = m \cdot \max |F^*(x) - F_0(x)|.$$

Из таблицы функции Колмогорова для уровня значимости $\alpha = 0,05$, т.е. при доверительной вероятности $\gamma = 1 - \alpha = 0,95$ выбирают значение критерия $\lambda_\gamma = 1,36$.

Если расчетное значение меньше табличного $\lambda < \lambda_\gamma$, выдвинутая гипотеза H_0 – ПРИНИМАЕТСЯ, хотя она может быть и НЕ ВЕРНА.

Выборка двумерной случайной величины

Пусть задана выборка (x, y) , объема n .

Несмещенная состоятельная оценка корреляционного момента вычисляется по формуле (2.13) [5]:

$$\hat{K}_{xy} = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x}) \cdot (y_i - \bar{y}),$$

где \bar{x} – выборочное среднее X , \bar{y} – выборочное среднее Y .

Состоятельная оценка коэффициента корреляции (формула 2.14):

$$\hat{r}_{xy} = \frac{\hat{K}_{xy}}{\sqrt{S_0^2(x) \cdot S_0^2(y)}}$$

где $S_0^2(x)$, $S_0^2(y)$ – несмещенные оценки дисперсий X и Y , соответственно.

Простейшим видом регрессии является линейная:

$$\bar{y}(x) = \hat{a}_0 + \hat{a}_1 x ,$$

где оценки коэффициентов линейной регрессии вычисляем по формулам (2.35) и (2.36), соответственно:

$$\hat{a}_1 = \frac{\hat{K}_{xy}}{S_0^2(x)} , \quad \hat{a}_0 = \bar{y} - \hat{a}_1 x .$$

Используемая литература

1. Котов В. и др. Методы алгоритмизации. Мн. Народная асвета. 1996.
2. Крячков А. и др. Программирование на С иС++. Практикум. М: Горячая линия-Телеком. 2000.
3. Демидович Е.М. Основы алгоритмизации и программирования. Язык Си. – Мн.: Бестпринт, 2001.
4. Синицын А.К., Навроцкий А.А. Алгоритмы вычислительной математики: Лабораторный практикум по курсу «Программирование» для студентов 1-2-го курсов всех специальностей БГУИР. – Мн., БГУИР, 2002.
5. Аксенчик А.В., Волковец А.И. и др. Методические указания и контрольные задания по курсу «Теория вероятностей и математическая статистика» для студ. всех спец. БГУИР заочной формы обучения. – Мн.Ж БГУИР, 2002.