С. В. АРТЕМОВА

• ИЗДАТЕЛЬСТВО ТГТУ •

Министерство образования Российской Федерации
Тамбовский государственный технический университет

С. В. АРТЕМОВА

ИНФОРМАТИКА

Часть I

Одобрено УМО в области
автоматики, электроники в качестве
учебного пособия для студентов,
обучающихся по направлению 5511 и
специальностям 2008 и 2005

Тамбов
Издательство ТГТУ
2001

УДК 681.3 (075)
ББК з 81я73-1
А86

Рецензенты:
Доктор педагогических наук,
заведующий отделом теории и методики обучения информатике
института общего и среднего образования РАО,
профессор *С. А. Бешенков*,
Кандидат технических наук, доцент ТГУ им. Державина
Н. В. Кузьмина

Артемова С. В.
А86 Информатика: Учебное пособие. Ч. 1. Тамбов: Изд-во Тамб. гос. техн. ун-та, 2001 г. 160 с
ISBN 5-8265-0063-8

ВВЕДЕНИЕ

Информатика: наука, изучающая законы и методы накопления, передачи и обработки информации с помощью ЭВМ; родовое понятие, охватывающее все виды человеческой деятельности, связанные с применением ЭВМ.

Информатика как фундаментальная наука связана с философией через учение об информации как общенаучной категории и теорией познания; математикой, через понятие математической модели, математической логики и теорией алгоритмов; с лингвистикой; системотехникой; кибернетикой – наукой об управлении, связи и переработке информации.

В пособии рассматриваются следующие разделы, изучаемые в информатике:

1) математическое моделирование;

2) выполнение вычислительного эксперимента;

3) алгоритмизация;

4) языки программирования высокого уровня;

5) программирование на примере двух алгоритмических языков СИ и Паскаль.

Инженер в своей профессиональной деятельности для успешного использования ЭВМ должен знать возможности технических средств и технического обеспечения ЭВМ, уметь общаться с машиной, но главное – он должен уметь формулировать задачи, разрабатывать алгоритмы, их решения, записывать алгоритмы на языке, понятном ЭВМ и доводить их до получения результатов.

ИСТОРИЯ РАЗВИТИЯ И ЭТАПЫ ЭВОЛЮЦИИ КОМПЬЮТЕРОВ

Еще в прошлом веке были выявлены и описаны физические явления, которые определяются двумя состояниями: 0 и 1. Также появился математический аппарат для оперирования двоичной системой счисления – булева алгебра.

Изобретателем первой вычислительной машины, предложившим в 1823 г. структуру автоматического вычислителя, считается английский математик Чарльз Бэббидж, а первым программистом – Ада Лавлейс. Структуру вычислителя составляли те же основные устройства, что и структуру современных компьютеров.

Для реализации компьютера не хватало только элементной базы.

Появление электронно-вакуумной лампы позволило претворить в реальность идею создания вычислительной машины, которая появилась в 1946 г. в США и получила название ENIAC – Electronic Numerical Integrator and Calculator (электронный численный интегратор и калькулятор). Первая отечественная ЭВМ – МЭСМ (малая электронная счетная машина) была создана в 1951 г. под руководством академика С. А. Лебедева.

В дальнейшем эволюция ЭВМ определялась развитием электроники, появлением новых элементов и принципов действия, т.е. развитием элементной базы.

В настоящее время насчитывается четыре поколения ЭВМ. Под поколением понимают все типы и модели ЭВМ, разработанные различными конструкторскими коллективами, но построенные на одних и тех же научных и технических принципах. Приведем краткую характеристику каждого поколения.

Первое поколение: (1946 – середина 50-х годов).

Элементная база: электронно-вакуумные лампы. В качестве пассивных элементов используются резисторы и конденсаторы.

Габариты: громадные шкафы, занимающие специальный машинный зал.

Быстродействие: 10 – 20 тыс. операций в секунду.

Эксплуатация: сложна. Лампы часто выходят из строя. Требуются сложные вентиляционные системы.

Программирование: Требуется непосредственное знание всех команд ЭВМ.

Структура: каждой команде соответствует своя логическая схема, выполненная на электролампах.

Второе поколение: (середина 50-х – середина 60-х годов).

Элементная база: полупроводниковые приборы (транзисторы, диоды и т.д.). В качестве пассивных элементов используются резисторы и конденсаторы.

Габариты: Стойки выше человеческого роста.

Быстродействие: 100 тыс.- 600 тыс. операций в секунду.

Эксплуатация: нет перегрева. При неисправности заменяют целиком плату.

Программирование: появились алгоритмические языки.

Структура: появился микропрограммный способ управления. Совмещение во времени работы разных устройств.

Третье поколение: (середина 60-х – середина 70-х годов).

Элементная база: интегральные схемы (ИС).

Габариты: существенно уменьшились. Две стойки человеческого роста.

Быстродействие: до 1 млн. операций в секунду.

Эксплуатация: более оперативный ремонт.

Программирование: похоже на 2-е поколение.

Структура: используются принципы модульности и магистрализации. Увеличен объем памяти.

Четвертое поколение: (середина 70-х – наши дни).

Элементная база: Большие интегральные схемы. Микропроцессоры.

Габариты: Персональные ЭВМ.

Быстродействие: от 1 млн. операций в секунду до сотней миллиардов.

История персональных компьютеров (ПК), получивших такое широкое распространение в настоящее время, более коротка. Важнейшей предпосылкой появления ПК явился выпуск в 1971 г. фирмой Intel МС i4004. Это был первый микропроцессор, функционально аналогичный центральному процессору ЭВМ, но значительно уступавший ему по мощности.

Первый ПК Altair 8800 был выпущен фирмой MITS в 1975 г и был основан на 8-битовом микропроцессоре (МП) i8080. За ним последовали ПК других фирм.

Но наибольшим успехам пользовался ПК Apple II фирмы Apple Computer. Эта разработка была в значительной степени обязана своей удачей тому, что компьютер Apple II обладал модульной конструкцией, что позволяло легко модифицировать его в дальнейшем.

Успех ПК фирмы Apple стал поводом для беспокойства корпорации IBM. В 1981 г IBM выпускает модель IBM PC (Personal Computer), разработанную небольшой группой специалистов под руководством Дона Эстриджа. Для минимизации затрат на разработку IBM PC был построен на основе открытой архитектуры. IBM PC собирался из модулей, производимых разными фирмами.

То, что в основу нового ПК был положен более мощный 16-разрядный МП i8088, а также открытая архитектура, выгодно выделяло его среди других ПК. Результатом открытой архитектуры явилось появление весной 1982 г ПК других фирм, совместимых с IBM PC на программном и аппаратном уровне. Некоторые из них даже превосходили по мощности последнюю из моделей на i8088 – IBM PC XT (eXtended Technology).

В 1984 г IBM разработала новый ПК на базе i80286, получивший название IBM PC AT, к этому времени IBM PC – совместимые ПК производили уже около 50 компаний.

В дальнейшем последовало лавинообразное повышение спроса на IBM PC-совместимые ПК. В настоящее время IBM PC-совместимые ПК занимают более 80 % рынка ПК.

1 ПОНЯТИЕ ИНФОРМАЦИИ, ОБЩАЯ ХАРАКТЕРИСТИКА ПРОЦЕССОВ СБОРА, ПЕРЕДАЧИ, ОБРАБОТКИ И НАКОПЛЕНИЯ ИНФОРМАЦИИ

П о н я т и е и н ф о р м а ц и и

Теория информации как самостоятельная дисциплина, связанная с восприятием, передачей и переработкой, хранением и использованием информации, была основана американским ученым К. Шенноном в конце 40-х гг. XX в. Предложенная Шенноном теория основывалась на фундаментальном понятии количественной меры неопределенности – *энтропии* – и связанного с ней понятия – количества информации. Она трактует информацию, как средство, "снимающее неопределенность" (энтропию) события или объекта познания. Другим фактором в становлении теории информации стало осознание того, что носитель информации – сигнал – имеет случайную природу.

Сигнал – физический процесс, имеющий информационное значение. В информатике он употребляется в смысле "любой процесс, несущий информацию". К основным характеристики сигналов в информационных системах относятся:

1 *Скорость создания информации H* – энтропия источника, отнесенная к единице времени

$$H = -\sum_{i=1}^{n} P_i \log_2 P_i ,$$

где H – количество информации, n – количество возможных событий, P_i – вероятность отдельных событий. Пусть потенциально может осуществиться некоторое множество событий n, каждое из которых может произойти с некоторой вероятностью P_i, т.е. существует неопределенность. Предположим, что одно из событий произошло, неопределенность уменьшилась, т.е. наступила полная определенность, поэтому количество информации H является мерой уменьшения неопределенности.

2 *Скорость передачи информации R* – количество информации, передаваемое по каналу связи в единицу времени (например, $R = 20$ бит/с).

3 *Избыточность* – свойство сигналов, состоящее в том, что каждый элемент сигнала несет информации меньше, чем может нести потенциально, например, символ текста. Избыточность оценивается по формуле

$$r = \frac{n - n_0}{n},$$

где n – текущая длина (число символов) сигнала, n_0 – длина сигнала при максимальной информационной нагрузке (минимальная длина).

При отсутствии помех избыточность вредна. Она снижает скорость передачи по каналу связи, увеличивает требуемый объем памяти при запоминании и увеличивает число операций при обработке и пр.).

4 Пропускная способность канала связи C – максимальная скорость передачи информации: $C = \max R$. Для зрения и слуха человека $C \approx 5$ бит/с.

В неживой природе информацию отображают с отражением. Это 3-я мировая константа (энергия, вещество). В быту

под информацией понимают сообщения, которые нас интересуют.

В теории связи – набор символов, иногда без смыслового содержания.

Информация-отображение внешнего мира с помощью знаков и сигналов (отраженного многообразия).

Свойства информации

1 Объективность (не должна зависеть от субъективного мнения).
2 Полнота (ее должно быть достаточно для принятия решения).
3 Актуальность (важность и необходимость в конкретный момент).
4 Понятность.
5 Полезность.
6 Достоверность – отражение реального положения дел.

Человек воспринимает информацию через чувственное познание, которое протекает в следующих основных формах:

1 Ощущение, возникающее при непосредственном воздействии объекта на органы чувств.
2 Восприятие образа предмета при непосредственном воздействии его на органы чувств.
3 Представление образа предмета в сознании человека, когда воздействие на органы чувств отсутствует.

Упорядоченное отображение реального мира в сознании человека есть знание, следовательно, информация – это приращение знаний субъекта от отдельно изучаемого объекта.

В семиотеке (наука о знаках и знаковых системах) рассматривается следующая классификация сигналов в зависимости от способов отображения:

1 Сигналы – индексы (проявление естественных свойств предметов и объектов – улыбка, слезы, ...),
2 Сигналы – копии (воспроизводят внешнюю форму отраженных объектов – фото, следы животных, отпечатки пальцев),
3 Условные знаки, сигналы.

Основные понятия о структуре, принципах работы компьютера

Основные принципы работы компьютера

Компьютер – это техническое средство преобразования информации, в основу работы которого заложены те же принципы обработки электрических сигналов, что и в любом электронном устройстве:

1) входная информация, представленная различными физическими процессами, как электрической, так и неэлектрической природы (буквами, цифрами, звуковыми сигналами и т.д.), преобразуется в электрический сигнал;

2) сигналы обрабатываются в блоке обработки;

3) с помощью преобразователя выходных сигналов обработанные сигналы преобразуются в неэлектрические сигналы (изображение на экране).

Назначение компьютера – обработка различного рода информации и представление ее в удобном для человека виде.

С позиции функционального назначения компьютер – это система, состоящая из 4-х основных устройств, выполняющих определенные функции: запоминающего устройства или памяти, которая разделяется на оперативную и постоянную, арифметико-логического устройства (АЛУ), устройства управления (УУ) и устройства ввода-вывода (УВВ). Рассмотрим их роль и назначение.

Запоминающее устройство (память) предназначается для хранения информации и команд программы в ЭВМ. Информация, которая хранится в памяти, представляет собой закодированные с помощью 0 и 1 числа, символы, слова, команды, адреса и т.д..

Под записью числа в память понимают размещение этого числа в ячейке по указанному адресу и хранение его там до выборки по команде программы. Предыдущая информация, находившаяся в данной ячейке, перезаписывается. При программировании, например, на языке Паскаль или СИ, адрес ячейки связан с именем переменной, которое представляется комбинацией букв и цифр, выбираемых программистом.

Под считыванием числа из памяти понимают выборку числа из ячейки с указанным адресом. При этом копия числа передается из памяти в требуемое устройство, а само число остается в ячейке.

Пересылка информации означает, что информация читается из одной ячейки и записывается в другую.

Адрес ячейки формируется в устройстве управления (УУ), затем поступает в устройство выборки адреса, которое открывает информационный канал и подключает нужную ячейку.

Числа, символы, команды хранятся в памяти на равноправных началах и имеют один и тот же формат. Ни для памяти, ни для самого компьютера не имеет значения тип данных. Типы различаются только при обработке данных программой. Длину, или разрядность, ячейки определяет количество двоичных разрядов (битов). Каждый бит может содержать 1 или 0. В современных компьютерах длина ячейки кратна 8 битам и измеряется в байтах. Минимальная длина ячейки, для которой можно сформировать адрес, равна 1 байту, состоящего из 8 бит.

Для характеристики памяти используются следующие параметры:

1) емкость памяти – максимальное количество хранимой информации в байтах;

2) быстродействие памяти – время обращения к памяти, определяемое временем считывания или временем записи информации.

Арифметико-логическое устройство (АЛУ). Производит арифметические и логические действия.

Следует отметить, что любую арифметическую операцию можно реализовать с использованием операции сложения.

Сложная логическая задача раскладывается на более простые задачи, где достаточно анализировать только два уровня: ДА и НЕТ.

Устройство управления (УУ). Управляет всем ходом вычислительного и логического процесса в компьютере, т.е. выполняет функции "регулировщика движения" информации. УУ читает команду, расшифровывает ее и подключает необходимые цепи для ее выполнения. Считывание следующей команды происходит автоматически.

Фактически УУ выполняет следующий цикл действий:

1) формирование адреса очередной команды;

2) чтение команды из памяти и ее расшифровка;

3) выполнение команды.

В современных компьютерах функции УУ и АЛУ выполняет одно устройство, называемое центральным процессором.

Системы счисления компьютера

С помощью компьютера мы вовсе не обязательно занимаемся какими-либо расчетами. Компьютер позволяет нам работать с текстами, графическими изображениями, звуком. Программы, которые мы используем для этого, переводят наши команды в вычислительные инструкции для центрального процессора, а ответные действия процессора – в понятные нам образы. Чтобы понять принцип работы компьютера, нам надо более подробно рассмотреть процессы обработки данных центральным процессором (ЦП) и их хранения в памяти.

Естественным способом передачи и хранения электрического сигнала является система: "есть сигнал"/"нет сигнала". Для работы с данными, представленными в таком виде, требуется система счисления, основанная всего на двух цифрах: "0" и "1". Такая система называется двоичной. Для обозначения двоичных цифр применяется термин "бит" (binary digit – bit). В группе из 8 бит умещается $2^8 = 256$ целых чисел, такая группа получила название байт (byte).

Одного байта достаточно, чтобы дать уникальное 8-битовое значение каждой заглавной и строчной букве русского и латинского алфавитов, цифрам, знакам препинания и некоторым другим символам.

Наиболее распространенная таблица кодирования символов 8-битовым кодом называется ASCII (American Standart Coding for Information Intercharge).

На одной машинописной странице умещается около 2000 символов, следовательно, для ее хранения в памяти требуется около 2000 байт.

Когда счет идет на сотни тысяч и миллионы байт, оперировать с такими числами становится неудобно. Поэтому разработана система более крупных единиц измерения объемов информации, наименьшая единица измерения информации в которой 1 байт.

1 Кбайт = 1 024 байт или 2^{10} (килобайт),

1 Мбайт = 1 048 576 байт или 2^{20} (мегабайт),

1 Гбайт = 1 073 741 824 байт или 2^{30} (гигобайт),

1 Тбайт = 1 099 511 627 776 байт или 2^{40} (терабайт).

В цифровых вычислительных машинах информация записывается в виде числовых кодов (чисел). Способ представления чисел посредством числовых знаков (цифр) называется системой счисления. Правила записи и действий над числами в системах счисления, используемых в цифровой вычислительной технике, определяют арифметические основы цифровых ЭВМ.

Различают два основных вида систем счисления: непозиционные и позиционные. Непозиционные системы счисления характеризуются тем, что значение числа, выражаемое совокупностью цифр, определяется только конфигурацией цифровых символов. Примером непозиционной системы является римская система счисления. Наибольшее распространение получили позиционные системы счисления, в которых значение любой цифры определяется не только конфигурацией ее символов, но и местоположением – позицией, которое она занимает в числе. При этом под основанием позиционной системы счисления q понимается количество различных цифр, используемых для представления числа.

СРЕДИ ПОЗИЦИОННЫХ СИСТЕМ РАЗЛИЧАЮТ ОДНОРОДНЫЕ И СМЕШАННЫЕ СИСТЕМЫ СЧИСЛЕНИЯ. В ОДНОРОДНЫХ СИСТЕМАХ КОЛИЧЕСТВО ДОПУСТИМЫХ ЦИФР ДЛЯ ВСЕХ ПОЗИЦИЙ (РАЗРЯДОВ) ЧИСЛА ОДИНАКОВО. ОДНОРОДНОЙ ПОЗИЦИОННОЙ СИСТЕМОЙ ЯВЛЯЕТСЯ ОБЩЕПРИНЯТАЯ ДЕСЯТИЧНАЯ СИСТЕМА СЧИСЛЕНИЯ (Q = 10), ИСПОЛЬЗУЮЩАЯ ДЛЯ ЗАПИСИ ЧИСЕЛ ДЕСЯТЬ ЦИФР ОТ 0 ДО 9.

Примером смешанной системы счисления может служить система отсчета времени, где в разрядах секунд и минут используется по 60 градаций, в разрядах часов – 24 градации и т.д.

Любое действительное число A, записанное в однородной позиционной системе счисления как

$$A = a_n a_{n-1} a_{n-2} \ldots a_1 a_0 , a_{-1} a_{-2} \ldots a_{-m},$$

может быть представлено в виде суммы степенного ряда

$$A = a_n q^n + a_{n-1} q^{n-1} + \ldots + a_1 q^1 + a_0 q^0 + a_{-1} q^{-1} + \ldots + a_{-m} q^{-m}, \qquad (1)$$

где q – основание системы счисления ($q >= 2$, целое положительное число); a_i – цифры системы счисления с основанием q($a_i = 0, 1, 2, ..., q - 1$); i – номер (вес) позиции (разряда) числа. Может быть реализовано бесконечное множество различных систем счисления.

В вычислительных машинах в основном используются однородные позиционные системы. Кроме десятичной, в ЭВМ находят широкое применение системы с основанием q, являющимся степенью числа 2, а именно: двоичная, восьмеричная и шестнадцатеричная системы счисления.

Десятичная система счисления. В соответствии с (1) позиция цифры в числе определяет степень числа с основанием

10, на которое эта цифра умножается. *Например*, число 534,79 можно представить как

$$534{,}79 = 5{,}10^2+3{,}10^1+4{,}10^0+7{,}10^{-1}+9{,}10^{-2}.$$

В одном разряде может быть представлено десять цифр: от 0 до 9. Прибавление единицы к старшей цифре разряда (к цифре 9) означает перенос единицы в старший разряд, т.е. для записи числа 10 и больших чисел требуются два и более разрядов. Число $N = q^n$ (где q – основание системы; n – целое положительное число) представляется в виде единицы в старшем разряде с последующими n нулями. Например, $N = 10^4 = 10\,000$. Это правило распространяется на все однородные позиционные системы счисления.

Двоичная система счисления. Основание системы $q = 2$.

Для записи чисел используются две цифры: 0 и 1 (табл. 1)

1 Соответствие десятичной и двоичной систем счисления

Десятичные	0	1	2	3	4	5	6	7	8	9
Двоичные	0	1	10	11	100	101	110	111	1000	1001

Старшей цифрой разряда является 1. Поэтому в двоичной системе 1 + 1 = 10, так как прибавление единицы к старшей цифре данного разряда дает перенос единицы в старший разряд.

Пример арифметических действий в двоичной системе счисления:

Сложение	Вычитание	Умножение
0 + 0 = 0	0 – 0 = 0	0 · 0 = 0
0 + 1 = 1	1 – 0 = 1	0 · 1 = 0
1 + 0 = 1	1 -1 = 0	1 · 0 = 0
1 + 1 = 10	10 – 1 = 1	1 · 1 = 1

При выполнении арифметических действий в двоичной системе счисления следует помнить, что единица является старшей значащей цифрой двоичного разряда. Выполняя в заданном разряде вычитание из нуля единицы, следует занять единицу из старшего значащего разряда, в результате чего в младшем разряде образуются две единицы, и тогда вычитать.

ОПЕРАЦИЯ УМНОЖЕНИЯ СВОДИТСЯ К МНОГОКРАТНОМУ СЛОЖЕНИЮ И СДВИГУ. ПРИ ВЫПОЛНЕНИИ ДЕЛЕНИЯ ИСПОЛЬЗУЮТСЯ ПРАВИЛА УМНОЖЕНИЯ И ВЫЧИТАНИЯ.

НАПРИМЕР:

Сложение	Вычитание	Умножение	Деление
1011001	100111	11011	101101 | 1001
+	–	×	– 1001 101
11101	1001	1001	
1110110	11110	11011	1001
		11011	– 1001
		11110011	0000

Восьмеричная система счисления. Основание системы $q = 8$. Для записи чисел используется восемь цифр от 0 до 7. В силу того, что основание восьмеричной системы является третьей степенью числа 2, для представления одного восьмеричного разряда требуется три значащих двоичных разряда¦ (триада). Перевод чисел из двоичной системы чисел в восьмеричную и наоборот осуществляется по триадам. Разбиение двоичного числа на триады осуществляется влево и вправо от запятой, отделяющей целую часть числа от дробной, *например:*

$1\ 1010\ 101_2 = 325_8\ 0{,}1101_2 = 0{,}64_8$,

$1\ 100\ 101\ 011{,}111\ 01_2 = 1453{,}72_8$,

$247{,}56_8 = 10\ 100\ 111{,}101\ 11_2$.

Если крайние триады получаются неполными, то они дополняются нулями.

Шестнадцатиричная система счисления. Основание системы $q = 2^4 = 16$. Для записи чисел используется шестнадцать цифр. Из них первые десять – известные цифры от 0 до 9. В качестве дополнительных цифр используются заглавные латинские буквы *A*, *B*, *C*, *D*, *E* и *F*. Перевод чисел из двоичной системы в шестнадцатиричную и обратно осуществляется по тетрадам аналогично двоично-восьмеричному переводу *например:*

$3FA_{16} = 0011\ 1111\ 1010_2$,

$10\ 0110\ 1101{,}111_2 = 26D, E_{16}$.

Неполные тетрады дополняются нулями.

В ряде случаев для более компактной записи удобнее пользоваться не двоичным или десятичным, а шестнадцатиричным представлением чисел.

Л О Г И Ч Е С К И Е О С Н О В Ы Ф У Н К Ц И О Н И Р О В А Н И Я К О М П Ь Ю Т Е Р А

Логика – наука, изучающая технику суждений и рассуждений. Разделы логики: математическая логика, формальная логика, диалектическая логика.

Формальная логика – дисциплина, изучающая особенности человеческих суждений и рассуждений.

Математическая логика – дисциплина, изучающая технику математически точных теорий и доказательств.

Диалектическая логика – логика, изучающая закономерности процессов, развивающихся в природе, обществе и познании.

Для описания логики функциональных, аппаратных и программных средств ЭВМ используется алгебра логики (Булева алгебра), названная по имени создателя.

О с н о в н ы е п о н я т и я а л г е б р ы л о г и к и

Булева алгебра оперирует с логическими переменными, которые могут принимать 2 значения: 1 (истинно) и 0 (ложно).

Л о г и ч е с к и е ф у н к ц и и

Логической функцией набора $f(x_1 \ldots x_n)$ называется функция, аргументы которой могут принимать только 2 значения – истина или ложь. Область определения логической функции конечна и зависит от числа возможных аргументов. Любая логическая функция может быть задана с помощью таблицы истинности, в левой части которой записываются возможные значения аргументов, а в правой – соответствующие им значения функции.

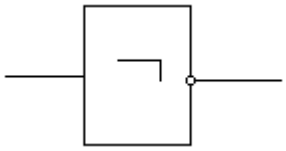
П о с т р о е н и е т а б л и ц и с т и н н о с т и

1 Не – инверсия – изменение значения аргумента на противоположное. Прибор, реализующий эту функцию – инвертор. Он имеет один вход и один выход. Его изображение на схемах:

Вход	Выход
0	1 (не ноль)
1	0 (не один)

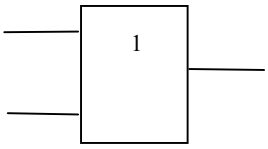
НЕ

СИ	Паскаль	Схемотехника
!	not	¬



2 Или – дизъюнкция – логическое сложение. Прибор, реализующий дизъюнкцию, называется "элемент или"; на схемах изображается следующим образом:

Вход 1	Вход 2	Выход
0	0	0
0	1	1
1	0	1
1	1	1

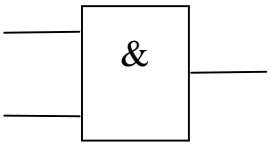


ИЛИ

СИ	Паскаль	Алгебра логики	Схемотехника
||	or	∨	1

3 И – конъюнкция – логическое умножение. Прибор, реализующий конъюнкцию, называется "элемент и"; на схемах изображается следующим образом:

Вход 1	Вход 2	Выход
0	0	0
0	1	0
1	0	0
1	1	1



И

СИ	Паскаль	Алгебра логики	Схемотехника
&&	and	∧ или •	&

На основе этих простейших элементов создаются сложные интегральные схемы, которые находят широкое применение в радиоэлектронике.

З а к о н ы а л г е б р ы л о г и к и

В алгебре логики выполняются следующие основные законы, позволяющие производить тождественные преобразования логических выражений.

Коммутативные законы (переместительные):

$x_1 \vee x_2 = x_2 \vee x_1$,

$x_1 \bullet x_2 = x_2 \bullet x_1$.

Ассоциативные законы (сочетательные):

$x_1 \vee (x_2 \vee x_3) = (x_1 \vee x_2) \vee x_3$

$x_1 \bullet (x_2 \bullet x_3) = (x_1 \bullet x_2) \bullet x_3$

Дистрибутивные законы (распределительные):

$x_1 \bullet (x_2 \vee x_3) = x_1 \bullet x_2 \vee x_1 \bullet x_3$ – дистрибутивность конъюнкции относительно дизъюнкции,

$x_1 \vee (x_2 \bullet x_3) = (x_1 \vee x_2) \bullet (x_1 \vee x_3)$ – дистрибутивность дизъюнкции относительно конъюнкции.

Правила де Моргана (теорема двойственности):

$\overline{x_1 \vee x_2} = \overline{x_1} \bullet \overline{x_2}$,

$\overline{x_1 \bullet x_2} = \overline{x_1} \vee \overline{x_2}$.

Правила операций с константами 0 и 1 (свойства констант):

$\overline{0} = 1, \overline{1} = 0$,

$x \bullet 1 = x,\ x \bullet 0 = 0$,

$x \vee 0 = x,\ x \vee 1 = 1$.

Правила операций переменной с ее инверсией:

$x \vee \overline{x} = 1$,

$x \bullet \overline{x} = 0$.

Закон поглощения:

$x_1 \vee (x_1 \bullet x_2) = x_1$,

$x_1 \bullet (x_1 \vee x_2) = x_1$.

Закон склеивания

$\overline{x}_1 x_2 \vee \overline{x}_1 \overline{x}_2 = \overline{x}_1$.

Закон идемпотентности (тавтологии):

$x \vee x = x$;

$x \bullet x = x$.

Закон двойного отрицания:

$\overline{\overline{x}} = x$.

Закон противоречия: $x \bullet x = 0$.

Закон исключающего третьего: $x \vee \overline{x} = 1$.

У п р о щ е н и е л о г и ч е с к и х в ы р а ж е н и й

Пусть дана логическая функция $f(x_1, x_2, x_3) = x_1 \overline{x}_2 \overline{x}_3 \vee x_1 \overline{x}_2 x_3 \vee x_1 x_2 x_3 =$

Применим к ней
распределительный закон и получим:

$= x_1(\overline{x}_2 \overline{x}_3 \vee \overline{x}_2 x_3 \vee x_2 x_3) =$

закон склеивания

$= x_1(\overline{x}_2 \vee x_2 x_3) =$

распределительный закон

$= x_1((\overline{x}_2 \vee x_2)(\overline{x}_2 \vee x_3)) =$

закон исключающего третьего

$= x_1(1(\overline{x}_2 \vee x_3)) =$

свойство констант

$= x_1(\overline{x}_2 \vee x_3)$.

После упрощения, как правило, получается комбинация из эквивалентных высказываний, в которых элементы не повторяются.

2 ТЕХНИЧЕСКИЕ И ПРОГРАММНЫЕ СРЕДСТВА РЕАЛИЗАЦИИ ИНФОРМАЦИОННЫХ ПРОЦЕССОВ

А п п а р а т н о е о б е с п е ч е н и е п е р с о н а л ь н о г о к о м п ь ю т е р а

Для функционирования компьютера необходимо два вида обеспечения. Это – аппаратное обеспечение (hardware) и программное обеспечение (software).

Рассмотрим аппаратную часть персонального компьютера.

Она состоит из:

1 Процессора, выполняющего управление компьютером;

2 Клавиатуры, позволяющей вводить символы в компьютер. Это устройство ввода;

3 Монитора, служащего для изображения текстов и графической информации – устройство вывода;

4 Накопителей (дисководов) для гибких магнитных дисков; накопителей на жестких магнитных дисках, или винчестера.

Кроме того, к персональным компьютерам могут подключаться: *принтер* (для вывода графической и текстовой информации); *мышь* (устройство, обеспечивающее ввод информации в компьютер); *сканер* (устройство для считывания графической и текстовой информации); *сетевой адаптер* (дает возможность подключить персональный компьютер в локальную или глобальную сеть); *стример* (устройство для быстрого хранения всей информации, находящейся на винчестере); графический планшет (для ввода контурных изображений); *плоттер* (для вывода чертежей на бумагу).

Приведем блок-схему системного блока компьютера, изображенную на рис. 1.

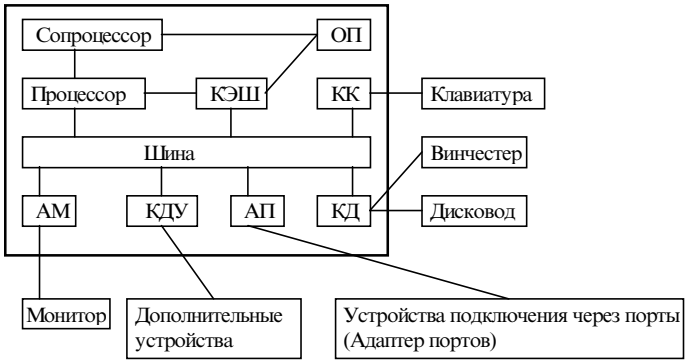


Рис. 1 Блок-схема системного блока

ОП – оперативная память; КК – контроллер клавиатуры; АМ – адаптер монитора; КДУ – контроллер дополнительных устройств; АП – адаптеры портов;
КД – контроллеры дисководов

К о н т р о л л е р ы и ш и н а

Для того, чтобы персональный компьютер мог работать, необходимо, чтобы в его оперативной памяти находилась программа и данные, и между ними происходил обмен. При работе программы часто бывает необходим ввод информации от пользователя или вывод ее на экран. Такой обмен называется вводом – выводом. Для его осуществления имеются два промежуточных звена:

1 Для каждого внешнего устройства ПК имеется электронная схема, которая им управляет. Ее называют контроллером или адаптером.

2 Все контроллеры или адаптеры взаимодействуют с микропроцессором и оперативной памятью через системную магистраль передачи данных, которую называют шиной. Системная шина является каналом соединения микропроцессора, оперативной памяти и интегральных устройств (рис. 2). Физически шина находится на материнской плате.

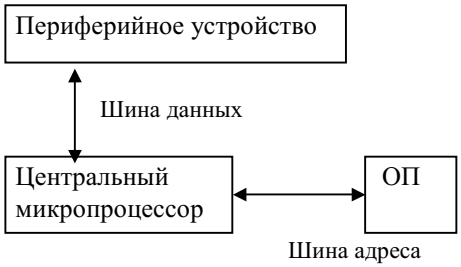


Рис. 2 Шина данных и шина адреса

Для обмена данными с памятью и устройствами ввода – вывода служат разные компоненты шины: взаимодействие микропроцессора с периферийными устройствами идет через шину данных, а адресация памяти происходит при помощи шины адреса.

Шина персонального компьютера IBM PC XT была 8-разрядной. Затем фирма IBM PC AT ввела стандарт 16-разрядной шины ISA. Все IBM – совместимые компьютеры перенимали стандарт ISA. В 1987 г. появилась 32-х разрядная

шина для IBM PS/2 , она называлась MCA. Эта шина была несовместима со стандартной – ISA. В 1989 году появилась новая 32-х разрядная шина EISA, совместимая с ISA. В настоящее время развивается концепция локальной шины, повышающей быстродействие за счет установки к ней дополнительной шины. Сейчас самый популярный стандарт – PCI (64 разряда). AGP-видео шина работает на частоте процессора.

Центральный микропроцессор

По одному названию микропроцессора можно составить представление о том, к какому классу оборудования принадлежит компьютер (табл. 2).

Разряды – Количество внутренних битовых разрядов является одним из важнейших факторов производительности микропроцессора.

Интерфейс с системной шиной. Количество разрядов интерфейса с шиной данных влияет на эффективность обмена данных с внешними устройствами. От разрядности адресного интерфейса напрямую зависит максимальный объем адресуемой оперативной памяти.

Тактовая частота. Необходима для синхронизации работы устройств компьютера. Влияет на скорость работы микропроцессора.

Адресация памяти. Количество адресуемой памяти равняется 2-м в степени разрядности адресной шины.

Оперативная память

Память IBM PC состоит из: ОЗУ(RAM) и ПЗУ(ROM). RAM доступна как для чтения, так и для записи. Применяется для хранения выполняемых в текущий момент программ и используемых ими данных. На современных персональных компьютерах должно быть не менее 16 Mb RAM , иначе персональный компьютер слишком быстро устареет. Для шины ISA невозможно установить более 16 Mb памяти. Важной характеристикой оперативной памяти является время выборки. Эта величина измеряется в наносекундах и обозначает минимальный промежуток времени, через который микропроцессор может обратиться к оперативной памяти. (табл. 2).

2 Процессоры фирмы Intel

Модель	Разряды	Интерфейс		Тактовая частота, MHz	Адресация памяти
		адрес	данные		
8086	16	20	16	5, 8, 10	1Мб
8086	16	20	8	5, 8	1Мб
80 286	16	24	16	8, 10, 12	16Мб
80 386 DX	32	32	32	10, 20, 25, 33	4Мб
80 386 SX	32	24	16	25, 33, 50	16Мб
486 DX	32	32	32	16, 20, 25, 33	4Мб
486 SX	32	32	32	50, 66	4Мб
486 DX2	32	32	32	75, 100	4Мб
486 DX4	32	32	32	60, 66, 50, 100	4Мб
Pentium	64	32	32	60 ... 200	4Мб
Pentium Pro	64	32	32	150 ...	4Мб
Pentium II	64	32	32	233 ... 450	4Мб
Pentium III	64	32	32	450 ...	4Мб

К Э Ш П А М Я Т Ь

Для ПК, у которых частота больше 28 МГц, необходимо обеспечить быстрый доступ к оперативной памяти, чтобы не простаивал микропроцессор. Для этого компьютеры могут оснащаться сверх быстрой памятью относительно небольшого объема (обычно от 64 – 256 Кб), в которой хранятся наиболее часто используемые участки оперативной памяти. КЭШ память располагается между микропроцессором и оперативной памятью. При обращении микропроцессора к памяти с начала производится поиск нужных данных в КЭШ памяти.

Мониторы

Предназначены для вывода на экран текстовой и графической информации. Могут работать в двух режимах: текстовом и графическом.

Текстовый режим: Экран монитора условно разбивается на отдельные участки – знакоместа. Чаще всего это 25 строк по 80 символов. На цветных мониторах, каждому знакоместу соответствует свой цвет фона и свой цвет символа. Структура байта атрибута символа приведена на рис. 3. Здесь младшие 4 бита (0-3) определяют цвет символа, причем биты 0-2 цвет, а 3 – интенсивность цвета. Старшие четыре бита (4-7) определяют цвет фона: 4-6 – цвет фона, а 7 – признак мерцания.

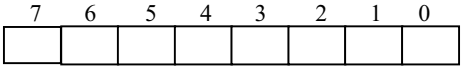


Рис. 3 Структура байта атрибута символа

Поскольку цветной дисплей имеет три основных цвета (красный, зеленый и синий), то для определения цвета отводятся три бита:

000 – черный цвет (0_{10});
001 – синий цвет (1_{10});
010 – зеленый цвет (2_{10});
011 – голубой цвет (3_{10});
100 – красный цвет (4_{10});
101 – фиолетовый цвет (5_{10});
110 – коричневый цвет (6_{10});
111 – светло-серый цвет (7_{10});
1000 – темно-серый (8_{10});
1001 – ярко-синий (9_{10});
1010 – ярко-зеленый (10_{10});
1011 – ярко-голубой (11_{10});
1100 – розовый (12_{10});
1101 – сиреневый (13_{10});
1110 – желтый (14_{10});
1111 – белый (15_{10}).

Графический режим предназначен для вывода на экран графиков рисунков и т.д. Надписи здесь могут иметь произвольный размер букв. В графическом режиме экран монитора состоит из точек или пикселей, который может быть цветным. Количество пикселей зависит от типа монитора. В графическом режиме для представления пикселя может быть использовано разное количество видеопамяти в зависимости от типа адаптера и выбранного видеорежима. Пиксель может быть представлен одним, двумя, четырьмя или восемью битами.

Графический режим монитора зависит от типа адаптера, работа которого поддерживается специальной программой, называемой драйвером. Типы адаптеров и соответствующих им драйверов приведены в табл. 3.

3 Драйверы и адаптеры мониторов

Адаптер	Драйвер	MAX разрешение	MAX кол-во цветов
IBM CGA, MCGA	CGA.BGI	640 × 480	256
IBM EGA, VGA	EGAVGA.BGI	640 × 480	16
Hercules монохромный	HERC.BGI	720 × 348	2
AT&T 6300 (400 строк)	ATT.BGI	640 × 400	2
IBM 8514	IBM8514.BGI	1024 × 868	256

Магнитные носители и накопители

Основными носителями информации IBM PC-совместимых компьютеров являются магнитные диски.

Наиболее широко используются жесткие диски (hard disk drive – HDD), или винчестеры, без которых трудно представить современный ПК. Жесткие диски обычно бывают несъемными.

Физически жесткий диск представляет собой несколько дисков из твердого материала (алюминий или, в последнее время, твердые виды стекла) с общей осью вращения, на которые нанесен магнитный слой. Каждой стороне каждого диска соответствует магнитная головка, которая может перемещаться по радиусу диска. Сторона каждого диска разделена на концентрические дорожки – цилиндры (до 1024 дорожек на магнитную головку). Каждая дорожка разделена на сектора (на старых 17 секторов, на новых 63 сектора). Для большей надежности жесткие диски выполняют изолированными от внешней среды.

В последнее время из-за увеличения плотности записи производителям жестких магнитных дисков стало мешать ограничение в 1024 дорожки на головку. В результате этого был создан стандарт LBA, который "обманывает" ПК, переводя лишние дорожки на несуществующие стороны. В результате этих действий ПК считает, что диск имеет больше головок.

Очень существенным может оказаться тип интерфейса жесткого диска. В настоящее время широко распространены два вида интерфейсов:

IDE-Integrated Drive Electronics (он же ATA) и SCSI (Small Computer System Interface).

Классический IDE имеет два основных ограничения – по объему диска (0,5 Гбайт) и по количеству устанавливаемых дисков (2 диска).

В настоящее время используется стандарт EIDE (ATA-2), в котором первое ограничение устранено путем принятия стандарта LBA, а количество дисков, которые можно подключить, увеличено до четырех. Главным достоинством интерфейса IDE является его невысокая стоимость, как дисков, так и контроллеров (в последних моделях ПК контролер устанавливается прямо на материнской плате).

SCSI является универсальным высокопроизводительным интерфейсом для устройств различного типа. Его модификации носят названия SCSI-2, Wide SCSI. Диски с интерфейсами SCSI имеют высокую скорость, но и их стоимость гораздо выше, чем у дисков с интерфейсом IDE.

Известными производителями жестких дисков являются такие фирмы, как Seagate, Quantum, Conner, Western Digital.

Для переноса небольших объемов информации с одного ПК на другой используются гибкие диски. Поэтому ПК должен иметь, по меньшей мере, один дисковод для работы с дискетами.

Используются дискеты двух размеров: 5,25 дюйма (постепенно выходят из употребления), и более распространенная дискета 3,5 дюйма.

	DS/DD	DS/HD	
5,25	360 Kb	1,2 Mb	
3,5	720 Kb	1,44 Mb	2,88 Mb

Для чтения-записи гибких дисков служат внешние магнитные накопители, называемые дисководами. Дисководы высокой плотности могут работать и с дискетами двойной плотности.

Дисководы CD-ROM

Наиболее распространены дисководы "только для чтения" -CD-ROM, использующие технологию цифровой записи, известной по аудио-CD (с помощью лазерного луча). Благодаря этому CD обладают большой скоростью работы и чрезвычайной емкостью около 600 Мбайт.

В последнее время CD-ROM очень популярны – на них поставляются программы и данные, имеющие большой объем. Кроме того, CD-дисководы позволяют проигрывать обычные аудио-CD.

Минимальная скорость передачи данных для CD-дисководов – 150 Кб/с. Выпускают дисководы с 2, 4, 6, 8, 10 кратным увеличением скорости. CD-дисководы используют интерфейсы Enhanced IDE и SCSI.

CMOS п а м я т ь и п р о г р а м м а Setup

Для хранения информации об оборудовании, а также о текущей дате и времени используется специализированная микросхема, содержащая таймер и 64 байта памяти с низким потреблением энергии. В выключенном состоянии ПК, CMOS память питается о аккумулятора, установленного на материнской плате.

Для работы с содержимым CMOS памяти, т.е. для изменении им формации об оборудовании, существует записанная в ПЗУ, специальная программа – Setup. Программу Setup можно вызвать нажатием клавиши <Del> при перезагрузке ПК.

П р и н т е р ы

Как правило, применяются принтеры следующих типов: матричные, лазерные, струйные.

Матричные. Принцип печати: печатающая головка содержит ряд тонких металлических стержней или иголок. В дешевых принтерах – 9 иголок. В более дорогих матричных принтерах – 24 или 48 иголок. Качества печати можно достигнуть увеличением количества проходов. Более быстрая печать обеспечивается принтером с 24-мя или 48-ми иголками.

Струйные. Изображение формируется микро каплями специальных чернил, выдуваемых на бумагу из сопел.

Лазерные. Это ксерография, изображение переносится на бумагу со специального барабана, к которому электрически притягиваются частички краски. Барабан электризуется с помощью лазера по командам компьютера.

П р о г р а м м н о е о б е с п е ч е н и е п е р с о н а л ь н о г о к о м п ь ю т е р а

Программы, работающие на ПК можно разделить на 3 категории:

1 Прикладные программы (обеспечивающие выполнение пользователем различных работ, то есть редактирование текстов, построение чертежей, и так далее).

2 Системные программы, выполняющие различные функции (проверка работоспособности узлов компьютера).

3 Инструментальная система (обеспечивающая создание новых программ для ПК).

Вторая часть пособия посвящена подробному описанию данного раздела.

О п е р а ц и о н н ы е с и с т е м ы

К системным программам относятся операционные системы. Это системная программа, которая загружается при включении компьютера. Она осуществляет диалог с пользователем, управление ПК, его ресурсами, запускает программы на выполнение, обеспечивает пользователю удобный интерфейс. Наиболее популярными операционными системами сегодня являются: MS DOS; PC DOS; DR DOS – однопользовательские системы, среди многопользовательских наиболее популярны: UNIX; OS/2; WINDOWS; WINDOWS NT; LINUX. Описание многопользовательских систем и работа в них приведены во второй части пособия.

О с н о в н ы е ч а с т и DOS.

1 Базовая система ввода-вывода (BIOS) находятся в ПЗУ, является встроенной в компьютер.

Назначение: Универсальные услуги ввода-вывода. Тестирование функционирования компьютера. Проверка работы памяти и устройств компьютера при его включении. Вызов загрузчика оперативной системы.

2 Загрузчик операционной системы – это короткая программа находящаяся в первом секторе загрузочной дискеты.

Назначение: Считывает в память два модуля операционной системы.

3 Дисковые файлы: IO.SYS, MSDOS.SYS, IBMBIO.COM, IBMDOS.COM, DRBIO.SYS, DRDOS.SYS загружаются в память и остаются там постоянно.

Назначение: IO.SYS – дополнение к базовой системе ввода-вывода. MSDOS.SYS- реализует основные высокоуровневые услуги DOS.

4 Командный процессор DOS находится в дисковом файле COMMAND.COM.

Назначение: Обрабатывает команды вводимые пользователем.

5 Внешние команды DOS – программы поставляемые с операционной системой.

6 Драйверы устройств.

Назначение: Обеспечивают обслуживание новых или нестандартное использование имеющихся устройств. Драйверы, загружаемые в память компьютера, указываются в файле CONFIG.SYS.

З а г р у з к а DOS

1 Считывается загрузчик операционной системы, находящийся в BOOT секторе.

2 Он считывает в память модули операционной системы и передает им управление.

3 Считывается конфигурация системы из файла CONFIG.SYS. В соответствии с указаниями, содержащимися в этом файле, загружаются драйверы устройств и устанавливаются параметры операционной системы.

4 Читается командный процессор COMMAND.COM и передается ему управление.

5 Командный процессор выполняет командный файл AUTOEXEC.BAT. В файле указывается команды и программы, выполняемые при каждом запуске компьютера.

6 Процесс загрузки заканчивается и система выдает приглашение, сообщая и показывая, что она готова к приему.

Работа с файлами в DOS

Создание текстовых файлов:
команда C:\ copy.con имя файла (если нет оболочки (NC, VC, DN...)).

УДАЛЕНИЕ ФАЙЛОВ: КОМАНДА DEL *.BAK

Переименование файлов: ren (rename) старое имя файла пробел новое имя файла.
Копирование файлов: copy , copy.prn имя файла.
Восстановление удаленных файлов: unerase.

Работа с каталогами:

---

cd – изменение текущего каталога.
dir – просмотр каталога.
md – создание каталога.
rd – уничтожение каталога.

3 МОДЕЛИ РЕШЕНИЯ ФУНКЦИОНАЛЬНЫХ И ВЫЧИСЛИТЕЛЬНЫХ ЗАДАЧ

Основные понятия и определения

Моделирование один из наиболее распространенных методов изучения различных процессов и явлений. В настоящее время многочисленные методы и приемы моделирования широко используются в научных исследованиях и в инженерной практике.

Под моделированием понимают процесс исследования свойств объекта (системы) с помощью его модели. При этом разрешают *физическое* моделирование и *математическое* моделирование.

Моделью называется объект, находящийся в отношении подобия к моделируемому объекту. Под подобием понимается взаимно однозначное соответствие между двумя объектами, при котором функции перехода от параметров, характеризующих один из объектов к параметрам другого объекта известны, а математические описания этих объектов могут быть преобразованы в тождественные. Соответственно физическая модель находится в отношении физического подобия к моделируемому объекту, а математическая модель в отношении математического подобия.

Следует помнить, что любая модель всегда основывается на некотором упрощении идеализации объекта. Модель не тождественна объекту, она является его приближенным отображением.

Физическое моделирование в силу сложности разработки физических моделей имеет ограниченное применение. Значительно более широкими возможностями обладает математическое моделирование. Под математическим моделированием понимается исследование различных процессов и явлений, имеющих различную физическую основу, с помощью их математического описания. С математическими моделями мы имеем дело постоянно. Так, например, для определения значения тока в электрической цепи мы пользуемся математической моделью, описываемой законом Ома: *I = U/R.*

Приведем *пример* модели фильтра низких частот.

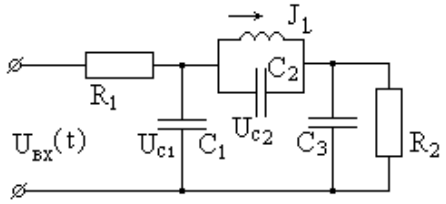


Рис. 4 Фильтр низких частот

Выполняемые объектом функции можно представить в виде математической модели, отражающей изменение вектора выхода y в зависимости от изменения вектора входа x. Обычно модель представляют отображением ì вида

$$\text{ì} = x \to y \qquad (2)$$

Например, для линейного усилителя отображению (2) соответствует уравнение

$$y = kx, \quad x \in X,$$

где k – коэффициент усиления.

Обязательному описанию подлежат все функции объекта. Взаимодействие объекта с другими частями системы задается значениями областей изменения x и y.

Для простых электрических схем непрерывного действия модель может записываться в виде системы дифференциальных уравнений или передаточной функции. *Например*, фильтру низких частот (рис. 4) соответствует следующая система трех дифференциальных уравнений, записанная в матричном виде:

$$\begin{pmatrix} \dot{u}_{c_1} \\ \dot{u}_{c_2} \\ \dot{u}_L \end{pmatrix} = \frac{1}{d}\begin{pmatrix} a_{11} & c_{2/R} & c_3 \\ a_{21} & -c_{2/R2} & -(c_1+c_3) \\ 0 & d/L & 0 \end{pmatrix}\begin{pmatrix} u_{c_1}(t) \\ u_{c_2}(t) \\ J_L \end{pmatrix} + \frac{1}{d}\begin{pmatrix} b_1 \\ b_2 \\ 0 \end{pmatrix} u_{вх}(t)$$

$$\dot{u}_{c_i} \stackrel{Ä}{=} \frac{d}{dt}u_{c_i}(t),\ \dot{J}_L \stackrel{Ä}{=} \frac{d}{dt}J_L(t),$$

где $d \stackrel{\Delta}{=} C_1C_2 + C_1C_3 + C_2C_3$;

$$a_{11} = -\frac{C_3}{R_1} - C_2\left(\frac{1}{R}+\frac{1}{R}\right),\ a_{21} = \frac{C_1}{R_2} - \frac{C_3}{R_1},\ b_1 = \frac{C_2+C_3}{R_1},\ b_2 = \frac{C_3}{R_1},$$

здесь $u_{c_i}(t)$, $u_{вх}(t)$ – напряжение соответственно на конденсаторе C_i, $i = 1, 2$ и на входе в момент времени t; $J_L(t)$ – ток через индуктивность L.

Математическая модель объекта, записанная в виде одного или нескольких аналитических выражений, называется *аналитической* моделью. Однако разработка аналитических моделей не всегда возможна или целесообразна в силу их сложности. В этом случае используются математические модели, которые реализуются в виде некоторой последовательности определенных действий. Такие модели получили название *алгоритмических* или *имитационных* моделей.

Математическая модель объекта может быть реализована с помощью специальных моделирующих установок. В этом случае имеет место *аппаратное моделирование.* В качестве таких моделирующих установок часто используют микропроцессорные системы, в составе которых имеется комплекс стандартных аппаратных моделей, реализующих различные виды математических зависимостей. В этом случае математическая модель представляется в виде совокупности схем, реализующих заданные математические зависимости.

Компьютеры позволяют решать целый ряд задач моделирования. Преимуществом аналогового моделирования является решение задач моделирования в реальном масштабе времени и простота организации процесса моделирования. Однако они позволяют реализовать только аналитические модели, обладают невысокой точностью и недостаточной универсальностью.

Наиболее эффективным средством реализации математических моделей любой степени сложности служит ЭВМ.

Методы исследования сложных систем

Одной из важных проблем в области разработки и создания современных сложных технических систем является исследование динамики их функционирования на различных этапах проектирования, испытания и эксплуатации. *Сложными системами* называются системы, состоящие из большого числа взаимосвязанных и взаимодействующих между собой элементов. При исследовании сложных систем возникают задачи исследования как отдельных видов оборудования и аппаратуры, входящих в систему, так и системы в целом.

К разряду сложных систем относятся крупные технические, технологические, энергетические и производственные комплексы.

При проектировании сложных систем ставится задача разработки систем, удовлетворяющих заданным техническим характеристикам. Поставленная задача может быть решена одним из следующих методов:

- методом синтеза оптимальной структуры системы с заданными характеристиками;
- методом анализа различных вариантов структуры системы для обеспечения требуемых технических характеристик.

Оптимальный синтез систем в большинстве случаев практически невозможен в силу сложности поставленной задачи и несовершенства современных методов синтеза сложных систем. Методы анализа сложных систем, включающие в себя элементы синтеза, в настоящее время достаточно развиты и получили широкое распространение.

Любая синтезированная или определенная каким-либо другим образом структура сложной системы для оценки ее показателей должна быть подвергнута испытаниям. Проведение испытаний системы является задачей анализа ее характеристик. Таким образом, конечным этапом проектирования сложной системы, осуществленного как методом синтеза структуры, так и методом анализа вариантов структур, является анализ показателей эффективности проектируемой системы.

Среди известных методов анализа показателей эффективности систем и исследования динамики их функционирования следует отметить:

- аналитический метод;
- метод натурных испытаний;
- метод полунатурного моделирования;
- моделирование процесса функционирования системы на ЭВМ.

Строгое аналитическое исследование процесса функционирования сложных систем практически невозможно. Определение аналитической модели сложной системы затрудняется множеством условий, определяемых особенностями работы системы, взаимодействием ее составляющих частей, влиянием внешней среды и т.п.

Натурные испытания сложных систем связаны с большими затратами времени и средств. Проведение испытаний предполагает наличие готового образца системы или ее физической модели, что исключает или затрудняет использование этого метода на этапе проектирования системы.

Широкое применение для исследования характеристик сложных систем находит метод полунатурного моделирования. При этом используется часть реальных устройств системы. Включенная в такую полунатурную модель ЭВМ имитирует работы остальных устройств системы, отображенных математическими моделями. Однако в большинстве случаев этот метод также связан со значительными затратами и трудностями, в частности, аппаратной

стыковки натурных моделей с ЭВМ.

Исследование функционирования сложных систем с помощью моделирования их работы на ЭВМ помогает сократить время и средства на разработку.

Затраты рабочего времени и материальных средств на реализацию метода имитационного моделирования оказываются незначительными по сравнению с затратами, связанными с натурным экспериментом. Результаты моделирования по своей ценности для практического решения задач часто близки к результатам натурного эксперимента.

Метод имитационного моделирования

Основан на использовании алгоритмических (имитационных) моделей, реализуемых на ЭВМ, для исследования процесса функционирования сложных систем. Для реализации метода необходимо разработать специальный *моделирующий алгоритм.* В соответствии с этим алгоритмом в ЭВМ вырабатывается информация, описывающая элементарные процессы исследуемой системы с учетом взаимосвязей и взаимных влияний. При этом моделирующий алгоритм строится в соответствии с логической структурой системы с сохранением последовательности протекаемых в ней процессов и отображением основных состояний системы.

Основными этапами метода имитационного моделирования являются:

- моделирование входных и внешних воздействий;
- воспроизведение работы моделируемой системы (моделирующий алгоритм);
- интерпретация и обработка результатов моделирования.

Общая схема метода имитационного моделирования приведена на рис. 5.

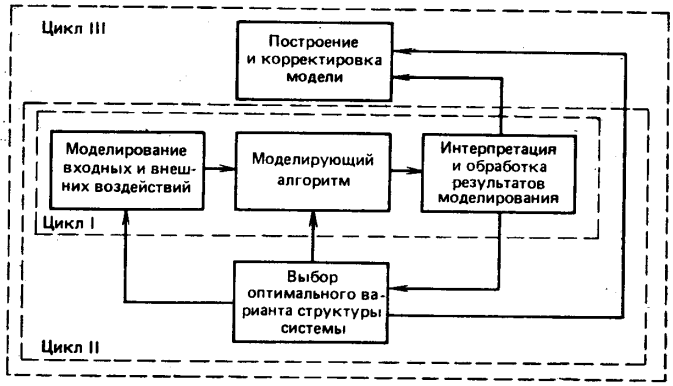


РИС. 5 ОБЩАЯ СХЕМА МЕТОДА ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ

Перечисленные выше этапы метода многократно повторяются для различных наборов входных и внешних воздействий, образуя внутренний цикл моделирования. Во внешнем цикле (цикл II) организуется просмотр заданных вариантов моделируемой системы. Процедура выбора оптимального варианта управляет просмотром вариантов, внося соответствующие коррективы в имитационную модель и в модели входных и внешних воздействии.

Процедура построения модели системы, контроля точности и корректировки модели по результатам машинного эксперимента задает и затем изменяет блоки внутреннего цикла в зависимости от фактических результатов моделирования. Таким образом, на схеме возникает внешний цикл (цикл III), отражающий деятельность исследователя по формированию, контролю и корректировке модели.

Метод имитационного моделирования позволяет решать задачи исключительной сложности. Исследуемая система может одновременно содержать элементы непрерывного и дискретного действия, быть подверженной влиянию многочисленных случайных факторов сложной природы, описываться весьма громоздкими соотношениями и т. п. Метод не требует создания специальной аппаратуры для каждой новой задачи и позволяет легко изменять значения параметров исследуемых систем и начальных условий. Эффективность метода имитационного моделирования тем более высока, чем на более ранних этапах проектирования системы он начинает использоваться.

Следует, однако, помнить, что метод имитационного моделирования является численным методом. Его можно считать распространением метода Монте-Карло на случай сложных систем. Как любой численный метод, он обладает существенным недостатком – его решение всегда носит частный характер. Решение соответствует фиксированным значениям параметров системы и начальных условий. Для анализа системы приходится многократно моделировать процесс ее функционирования, варьируя исходные данные модели. Таким образом, для реализации имитационных моделей сложных систем необходимо наличие ЭВМ высокой производительности.

Для моделирования системы на ЭВМ необходимо записать моделирующий алгоритм на одном из входных языков ЭВМ. В качестве входных языков для решения задач моделирования могут быть с успехом

использованы универсальные алгоритмические языки высокого уровня, СИ, Паскаль и др.

Аппаратно-программное моделирование сложных систем

Анализ развития наиболее сложных технических систем позволяет сделать вывод о все более глубоком проникновении ЭВМ в их структуру. Вычислительные машины становятся неотъемлемой, а зачастую и основной частью таких систем. Прежде всего это относится к сложным радиоэлектронным системам. Среди них различные автоматические системы, в том числе системы автоматической коммутации (электронные АТС), системы радиосвязи, радиотелеметрические системы, системы радиолокации и радионавигации, различные системы управления.

При построении таких систем в значительной степени используются принципы и структуры организации вычислительных машин и вычислительных систем (ВС). Характерной особенностью является наличие в системах нескольких процессоров, объединенных различными способами в специализированную ВС. При этом осуществляется переход от "жесткой" логики функционирования технических систем к универсальной "программной" логике. В силу этого все более значительную роль в таких системах, наряду с аппаратными средствами, играет *специализированное* системное и прикладное *программное обеспечение* (СПО).

На этапах разработки проектирования, отладки и испытания сложных систем с высоким удельным весом аппаратно-программных средств вычислительной техники ставится задача анализа и синтеза вариантов организации структуры аппаратных средств, а также разработки и отладки СПО большого объема. Эта задача может быть решена с помощью аппаратно–программного моделирования с использованием универсальных моделирующих комплексов, построенных на базе однородных ВС с программируемой структурой.

Аппаратно–программное моделирование можно считать частным случаем полунатурного моделирования. На первом этапе разрабатывается концептуальная модель заданного класса систем на основе анализа типовых процессов, структур и аппаратных блоков. Концептуальная модель реализуется на аппаратно-программных средствах моделирующего комплекса. При этом моделирующий комплекс может настраиваться на соответствующую структуру системы программным путем за счет возможности программирования структуры используемой микропроцессорной ВС. Часть аппаратных и программных средств микропроцессорной ВС моделирующего комплекса непосредственно отражает аппаратно-программные средства, входящие в исследуемую систему *(аппаратное моделирование),* другая часть реализует имитационную модель функциональных средств исследуемой системы, внешней обстановки, влияния помех и т.п. *(программное моделирование).*

Разработка аппаратно-программных моделирующих комплексов является сложной технической задачей. Несмотря на это, применение таких комплексов находит все большее распространение. При достаточной производительности вычислительных средств комплекса процесс исследования системы может вестись в реальном масштабе времени. В составе комплекса могут использоваться как универсальные микроЭВМ общего назначения, так и вычислительные средства, непосредственно входящие в исследуемую систему. Подобные моделирующие комплексы являются универсальными стендами для разработки и отладки аппаратно-программных средств, проектируемых систем заданного класса. Они могут использоваться в качестве тренажеров по обучению обслуживающего персонала.

4 АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Понятие алгоритма и его свойства

В основе решения любой задачи лежит понятие алгоритма. Под алгоритмом принято понимать точное предписание, определяющее вычислительный процесс идущий от варьируемых начальных данных к исходному результату (ГОСТ19.781-74 - определение алгоритма). При составлении алгоритма следует учитывать ряд требований, выполнение которых приводит к формированию необходимых свойств.

1 Алгоритм должен быть однозначным, исключающим произвольность толкования любого из предписаний и заданного порядка исполнения. Это свойство алгоритма называется *определенностью*.

2 Реализация вычислительного процесса, предусмотренного алгоритмом должна через определенное количество шагов привести к выдачи результатов или сообщения о невозможности решения задачи. Это свойство называется *результативностью*.

3 Решение однотипных задач с различными исходными данными можно осуществлять по одному и тому же алгоритму, что дает возможность писать типовые программы, для решения задач при различных вариантах задания значения исходных данных. Это свойство называется *массовостью*.

4 Предопределенный алгоритмом вычислительный процесс расчленить на отдельные этапы (элементарные операции). Это свойство алгоритма называется *дискретностью*.

СПОСОБЫ ОПИСАНИЯ АЛГОРИТМОВ

К изобразительным средствам описания алгоритмов относятся следующие способы их представления:

1) словесный – запись на естественных языках;

2) структурно-стилизованный – запись на алгоритмическом языке псевдокода;

3) графический – изображение схем из графических символов;

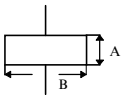
4) программный – текст на языке программирования.

Структурно-стилизованный способ записи алгоритмов основан на формализованном представлении предписаний, задаваемых путем использования ограниченного набора типовых синтаксических конструкций, представляемых в понятных для разработчика символах. Такие средства описания алгоритма часто называют *псевдокодами.*

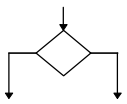
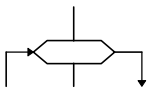
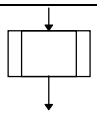
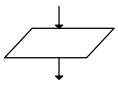
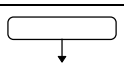
Язык графических символов

Для изображения структуры алгоритма используют совокупность блочных символов или блоков соединенных линиями передач управления. Такой метод называется методом блок схем. ГОСТ-19.003-80- форма символов, ГОСТ-19.002-80- правила составления алгоритмов. Некоторые символы блок-схем приведены в табл. 4.

4 Символы блок-схем

Название символов	Обозначение	Пояснение
Процесс		Блок обработки процесса (определяет вычислительное действие или последовательность вычислительных действий)

Продолжение табл. 4

Название символов	Обозначение	Пояснение
Решение		Проверка условий разветвления в алгоритме.
Модификация		Начало цикла. Внутри блока записываются параметры цикла, для которого указывается его начальное значение.
Предопределенный процесс		Вычисление по подпрограмме или стандартная подпрограмма.
Ввод-вывод		Ввод-вывод в общем виде.
Пуск – остановлено		Начало или конец алгоритма.
Комментарии		Комментарии, поясняющие работу алгоритма
Соединители		Если алгоритм не умещается на одном листе, то его можно объединить соединителями.

Разновидности структур алгоритмов

Логическая структура любого алгоритма может быть представлена комбинацией трех базовых структур. Это: следование, разветвление и цикл. Характерной особенностью этих структур является наличие в них одного входа и одного выхода.

1 *Базовая структура следования* (рис. 6) означает, что два оператора должны быть выполнены последовательно.

Совокупность базовых структур следования, выполняющих вычислительные операции, *называют линейным вычислительным алгоритмом.*

2 Структура "*если; то; иначе*" (рис. 7). Эта структура обеспечивает в зависимости от результатов проверки условия, выбор одного из альтернативных путей работы алгоритма. Каждый из путей ведет к общему выходу не зависимо от того, какой путь будет выбран.

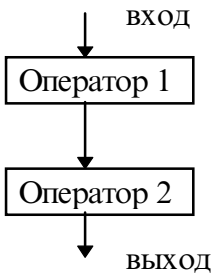


РИС. 6 СТРУКТУРА "СЛЕДОВАНИЕ"

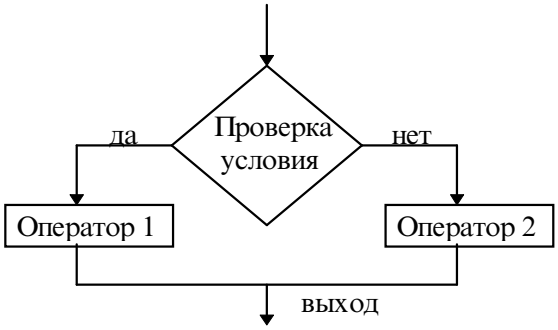


Рис. 7 Структура "если, то, иначе"

Алгоритм, в состав которого входит базовая структура разветвления, называется *разветвляющимся*.

3 Повторение (*цикл*) обеспечивает повторение или циклическую работу операторов. Различают две разновидности цикла: с предусловием (рис. 8); с постусловием (рис. 9).

Основным их различием является то, что операторы тела цикла "пока" выполняются в зависимости от условия, а в цикле "до" операторы тела цикла выполнятся хотя бы раз в не зависимости от условия.

Алгоритмы, имеющие в своем составе базовую структуру, цикла называются *циклическими алгоритмами.* Для организации цикла необходимы управляющие операторы, задание начального значения параметра цикла, изменение параметра цикла, проверка условия окончания цикла.

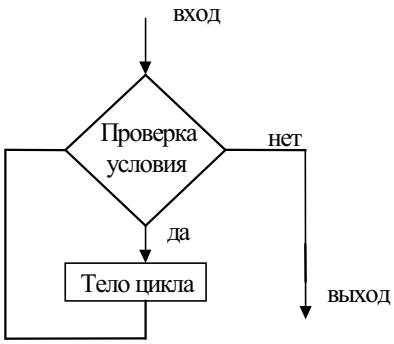


Рис. 8 Цикл с предусловием "пока"

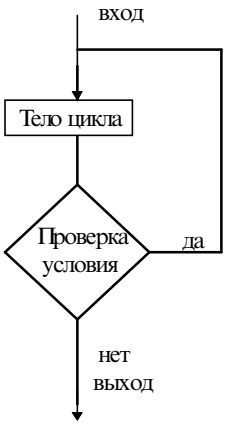


Рис. 9 Цикл с постусловием "до"

5 ЯЗЫКИ ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ

ЭЛЕМЕНТЫ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ СИ И ПАСКАЛЬ

Языки Turbo C++ (СИ) и Turbo Pascal 7.0 (Паскаль) поддерживают модульный принцип программирования. Программы обычно разбиваются на модули, состоящие из подпрограмм.

Лексемами называются минимальные лексически значимые единицы текста программа. В их число входят также зарезервированные слова, специальные символы, идентификаторы, метки, числа и строковые константы. Лексемы разделяются между собой одним или несколькими разделителями.

Идентификаторы - это имена, которые используются для обозначения типов, констант, переменных, процедур, функций, модулей, и программ. Идентификатор должен начинаться с буквы или символа, допускаются цифры и символы подчеркивания. Идентификаторы могут иметь любую длину, однако, значимыми для компилятора Паскаля будут первые 63 символа, для компилятора СИ - 31 символ. Компилятор языка СИ имеет возможность различать прописные и заглавные буквы. Поэтому комбинация прописных и заглавных букв в идентификаторах во всей программе должна быть одинаковой. В Паскале сложносоставные идентификаторы пишутся слитно, но каждая часть с большой буквы, а СИ используется знак подчеркивания. *Например*: my_program в СИ и MyProgram в Паскале.

Ссылка на переменную представляет собой идентификатор переменной с несколькими квалификаторами или без них. *Квалификаторы* служат для изменения значения ссылки на переменную.

Например: Идентификатор массива без квалификатора является ссылкой на весь массив, Results. Идентификатор массива с указанным индексом обозначает конкретный элемент массива, в данном случае переменную: Results[Current + 1].

В СЛУЧАЕ, ЕСЛИ КОМПОНЕНТОЙ ЯВЛЯЕТСЯ ЗАПИСЬ ИЛИ ОБЪЕКТ, ЗА ИНДЕКСОМ МОЖНО УКАЗАТЬ ОБОЗНАЧЕНИЕ ПОЛЯ; В ЭТОМ СЛУЧАЕ ССЫЛКА НА ПЕРЕМЕННУЮ ОЗНАЧАЕТ КОНКРЕТНОЕ ПОЛЕ ВНУТРИ КОНКРЕТНОЙ КОМПОНЕНТЫ МАССИВА: RESULTS[CURRENT + 1].DATA.

Если указываемая переменная – массив, то можно добавить индексы для обозначения элементов этого массива.

Строкой в СИ и Паскаль называются последовательность любых символов расширенного набора ASCII, заключенную в апостроф в Паскале и в двойные кавычки в СИ.

Комментарием считается любая комбинация произвольных символов заключенных в {} или (* *) в Паскале, в СИ /* */ или // – до конца строки. Комментарий, в котором за открывающейся фигурной скобкой следует символ $, интерпретируется как директива Паскаля.

Структура программы

Важнейшим с точки зрения построения программ на СИ и Паскале является принцип, согласно которому все именованные объекты должны быть описаны. Обращения к объекту становится возможным после того, когда дано его явное описание, однозначно задающее структуру. Программы содержат два раздела: в первом содержится объявление всех объектов, используемых в программе, за исключением тех, что описаны вне ее. Второй раздел содержит операторы, которые описывают действия над предварительно объявленными объектами, необходимыми для реализации алгоритмов. В Паскале этот раздел отделяется от первого словом begin, заканчивается словом end; в СИ вместо begin используется открывающаяся фигурная скобка {, а вместо end закрывающаяся }. После end в Паскале в этом разделе обязательно должна быть точка. В Паскале заголовок не является обязательным атрибутом программных средств. Однако, рекомендуется указывать имя программы. Делается это так. Program – имя программы. В СИ заголовок отсутствует.

Компилятор СИ более гибок и не требует жесткого формата структуры программ. Описание переменных может встречаться по мере их введения, однако для соблюдения хорошего стиля программирования желательно придерживаться "паскалевской" структуры программы.

Каждая программа, написанная на СИ должена иметь функцию main, так как ее работа начинается именно с нее.

Модуль – это новая програмная единица более высоксго уровня.

П Р О Г Р А М М А Н А С И М О Ж Е Т И М Е Т Ь С Л Е Д У Ю Щ У Ю С Т Р У К Т У Р У :

- включение используемых библиотечных файлов;
- описание глобальных переменных и функций;
- тексты функций;
- главную функцию с именем void main (void), а в ней:
- описание локальных переменных;
- операторы программы.

В стандартном Паскале программы имеют жесткий формат:

```
program имя программы;
uses включаемые модули;
 label
   метки;
 const
   объявление констант;
 type
   определение типов данных;
 var
   объявление переменных;
 procedure или function;
begin
   тело программы, процедуры или функции;
end.
```

Наличие всех пяти объявлений – label, const, type, var, procedure и function – в программе необязательно. Однако для стандартного Паскаля, если они имеются, порядок их следования строго регламентирован, и в программе они должны присутствовать только один раз. За секцией объявлений, следуют процедуры и функции, и только затем тело программы.

Turbo Pascal 7.0 и СИ++ обеспечивают более гибкую структуру программы. Однако следует учитывать, что все идентификаторы должны быть.

Стиль программирования. Хорошим стилем программирования считаетсястиль на писания программы, при котором она имеет четкую структуру, делающую ее доступной для понимания. Следует придерживаться стиля приведенного в разделе help СИ и Паскаля.

П р е п р о ц е с с о р, к о м п и л я т о р, з а г р у з ч и к СИ

Для того, чтобы исходная программа, написанная на языке СИ была оттранслирована и переведена в исполняемый машинный код – файл с расширением *.exe, она должна пройти через три процесса: препроцессирование, компиляция и загрузка.

1 В задачу предпроцессора входят подключение при необходимости к данной программе внешних файлов, указанных при помощи директивы #include < имя файла с расширением h > или #include <" имя файла с расширением">. Препроцессор подставляет на место этих директив тексты файлов. Если имя файла заключено в < >, то поиск файла производится в специальном разделе, где хранятся файлы с расширением *.h (в директории INCLUDE). Если имя файла заключено в " ", то поиск файла производится сначала в текущем разделе, а затем в разделе постановок. "имя файла" – это тексты программ, которые являются текстами внешних функций (по аналогии с модулями Паскаля).

Н е к о т о р ы е с т а н д а р т н ы е в к л ю ч а е м ы е б и б л и о т е к и:

#include<stdio.h> – ввод-вывод;
#include<conio.h> – работа с окнами;
#include<math.h> – математическая библиотека;
#include<string.h>-работа со строками;
#include<dos.h>-работа с прерываниями;
#include<stdlib.h>-работа с генератором случайных чисел;
#include<graph.h>-работа с графикой;
#include<alloc.h>-работа с динамической памятью.

С помощью директивы #define, вслед за которой пишется имя макро и значение макро, например, #define N 3 можно указать предпроцессору, чтобы при любом появлении в исходном файле данного имени макро он заменял это имя на соответствующее значение макро. Часто макро используются для того, чтобы увязать идентификатор и его значение. Например, # define Pi 3.14. В СИ принято имена макро писать с заглавной буквы.

2 Компилятор за несколько этапов транслирует то, что вырабатывает препроцессор в объектный файл с расширением *.OBJ, содержащий оптимизированный машинный код, при условии, что не встретились синтаксические и семантические ошибки. Если в исходном файле СИ – программы обнаруживаются ошибки, то программисту выдается их список.

3 Загрузчик связывает между собой объектный файл, получаемый от компилятора с программами из требуемых библиотек и возможно с другими файлами. В результате сборки получается файл с расширением *.EXE, который может быть использован персональным компьютером.

Т и п ы д а н н ы х

Это константы, переменные и структуры, содержащие числа (целые и вещественные), текст (символы и строки) или адреса (переменных и структур).

Под данными понимаются конкретные значения, которые обрабатываются во время выполнения программы.

Все типы данных в СИ можно разделить на 2 категории: скалярные и составные. Приведены на структурной схеме (рис. 10).

Для переопределения типа в СИ используется оператор typedef.

Пример:

typedef char str40[41];

typedef unsigned cher byte;.

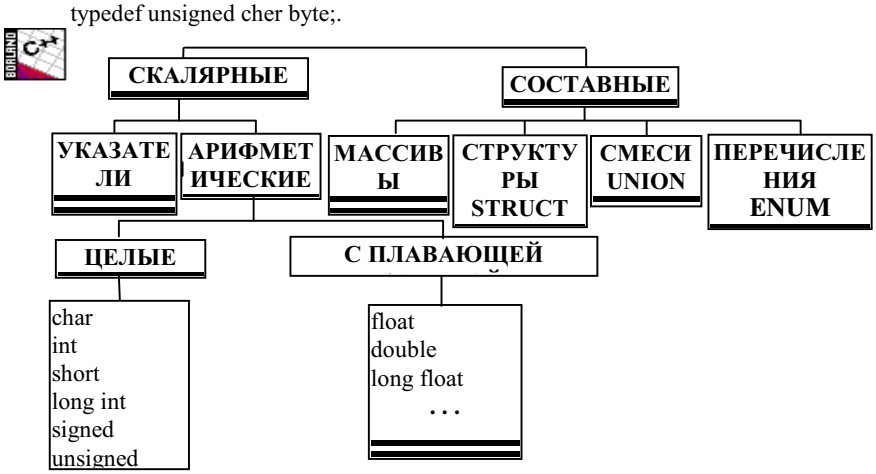


Рис. 10 Структурная схема типов данных в СИ

В Паскале насчитываются 4 основных класса типов данных:

1) простые типы;
2) структурированные типы;
3) ссылочные типы;
4) процедурные типы.

Структурная схема приведена на рис. 11.

На ряду с предопределенными типами в Паскале могут использоваться типы, задаваемые программистом. Все другие типы должны быть описаны в разделе объявлений. Простые типы разделяются на порядковые и вещественные. Порядковые типы разделяются на целые, множественные, символьные, перечисляемые, интервальные. Порядковое типы включают значения каждому из которых ставится соответствующий порядковый номер. Любому значению порядкового типа можно применить стандартную функцию Ord (x), которая возвращает порядковый номер этого значения. Любому значения порядкового типа можно применить стандартную функцию Pred (x), которая возвращает значение, предшествующее заданному. Любому значению порядкового типа можно применить стандартную функцию Succ (x), которая возвращает значение, следующее за заданным.

Типы данных в Паскале

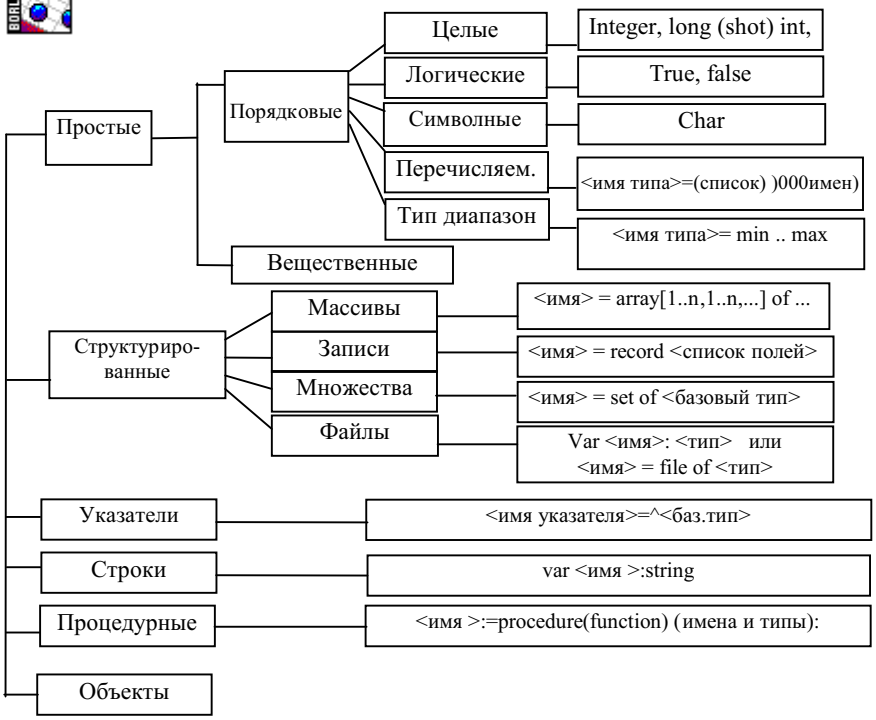


Рис. 11 Структурная схема типов данных в Паскале

Л О Г И Ч Е С К И Й Т И П В П А С К А Л Е

Данные логического типа могут принимать два значения: true and false. Эти значения определяется соотношением:

false <true;

Ord (false) = 0; {Определение значения порядкового тела}
Ord (true) = 1;
Succ (false) = true;
Succ (true) = false. {Последние значение порядкового типа}

П Е Р Е Ч И С Л Я Е М Ы Е Т И П Ы В П А С К А Л Е

Перечисляемый тип определяется конечным набором значений, которые указываются в круглых скобках. Тип объявляется следующим образом:

Type [Name] = (name1, name2, …, namen),

где [Name]- имя типа, а namei – имена перечисляемых переменных.

Например:

Type Color = (Blue, Red, Green).

И Н Т Е Р В А Л Ь Н Ы Е Т И П Ы В П А С К А Л Е

Величина интервального типа может принимать значение из диапазона значений любого порядкового типа. При объявлении этого типа задаются две константы, указывающие нижнюю и верхнюю границы диапазона допустимых значений. Обе константы должны принадлежать одному порядковому типу. Этот тип объявляется следующим образом:

Type [Name] = [const1]..[const2].

Например:

Type Diapazon = 1 ... 10;

Этот тип принято использовать при объявлениях массивов.

С Т Р У К Т У Р И Р О В А Н Н Ы Е Т И П Ы Д А Н Н Ы Х

Это массивы, записи, множества, и файлы. Помимо этого, есть также типы данных как указатели, строки, процедурные типы данных и объекты.

Определяются типами своих компонентов и способами их структуризации.

М а с с и в ы

В Паскале к регулярному типу относят одномерную или многомерную совокупность фиксированного числа однотипных элементов. При объявлении регулярного типа задается тип его элементов, а также тип индекса по каждому его измерению. Допустимыми типами индекса являются все порядковые типы за исключением LongInt. Удобно использовать величины интервального типа. Объявим массив из 100 элементов.

```
Type
    Data = array[1..100] of real;
```

Если элементы массива также являются массивами, то результирующую структуру можно рассматривать как массив, увеличенный на единицу размерности. Запись может быть разная:

Array[Boolean] of Array [1..10] of Array[Size] of real

или

Array[Boolean,1..10,Size] of Real;

Эти две записи эквивалентны.

```
const N = 3;
type
    m = Array[1..N] of Real; {m – это определение типа одномерный массив}
 var
    b : array[1..N] of m {b- это двумерный массив или массив массива}
 i, j : iteger;
```

В СИ для объявления массива в квадратных скобках указывают максимальное количество элементов массива. Необходимо помнить, что элементы в СИ нумеруются с нуля.

Пример:

```
# define N 3
 int mas [N], mass [N][N];
 int i, j;
```

здесь mas – одномерный массив, а mass – двумерный массив.

С Т Р О К О В Ы Й И С И М В О Л Ь Н Ы Й Т И П

Данные строковых типов представляют собой последовательности символов переменной длины. При объявлении строкой величины в квадратных скобках указывается максимальное число заключенных в ней символов в диапазоне от 1 до 255.

```
Const
    ErrorMes: String[25] = ‘Error’;
var MyString: String [25];
```

КАЖДЫЙ СИМВОЛ В STRING ПРОНУМЕРОВАН, ПЕРВЫЙ ИМЕЕТ НОМЕР 1. К ЛЮБОМУ СИМВОЛУ МОЖНО ОБРАЩАТЬСЯ, УКАЗЫВАЯ ЕГО ПОРЯДКОВЫЙ НОМЕР.

В Паскале 7.0 появился новый тип данных строки, заканчивающие нулевым символом #0. В таких типах можно разместить 65 535 символов. Эти строки хранятся в массивах и объявляются следующим образом:

Идентификатор строка:

Array[тип диапазон,] of Char;

Например:

```
Var
  Zstr : Array[0..N] of Char;
```

Для работы с такими строками необходимо активизировать директиву компилятора {$X+} {поддержка расширенного синтаксиса}.

В СИ строки, это массивы символов типа char, *например*:

```
char stroka [25].
```

М Н О Ж Е С Т В А В П А С К А Л Е

Множества – это наборы однотипных логически связанных друг с другом объектов. Характер связей между объектами лишь подразумевается программистом и никак не контролируется Паскалем.

```
Type
[Name] = set of [base Type]
```

Базовый тип не может быть типа word, integer, longint. Базовый тип не должен включать более 256 возможных значений. Для задания множества используется конструктор множества или список спецификаций элементов множества. Над множествами определены следующие операции:

* – пересечение множеств; результат содержит элементы общие для обоих множеств;

+ – объединение множеств; результат содержит элементы первого множества, дополненный недостающими элементами из второго множества;

– – разность множеств; результат содержит элементы 1-го множества, которые не принадлежат второму;

= – проверка эквивалентности возвращает "истину", если множества эквиваленты;

<> – проверка неэквивалентности возвращает "истину" если множества неэквиваленты;

<= – проверка вхождения первого во второе;

>= – проверка вхождения второго множества в первое;

in – проверка принадлежности элемента I множеству S

```
I in S.
```

Процедуры

Include (S,I) and Exclude (S,I) – соответственно включают и исключают элемент из множества (S – множество; I – элемент).

Пример:

```
Type DigitChar = Set of ‘0’..’9’;
 Var S1, S2, S3 : DigitChar;
 Begin
    S1:= [‘1’,’2’,’3’];
    S2:= [‘3’,’1’,’4’]; {конструктор множества}
    S3:= [‘2’,’5’];
      if '1' in S2 then
           Writeln ('Элемент 1 принадлежит множеству S2');
    end;
```

Список спецификаций элементов множества, отделяемых друг от друга запятыми, и обрамленный в квадратные скобки, называют конструктором множества.

Ф А Й Л О В Ы Е Т И П Ы П А С К А Л Я

Файловый тип представляет собой линейную последовательность компонентов или записей, которые могут принадлежать любому типу, за исключением файлового или иного другого, содержащего файловый.

```
Type
[Name] = file of [Type].
```

Файловый тип или переменную файлового типа можно задать одним из трех способов:

[Name] = File of[Type]; – типизированный файл;

[Name] = Text; – текстовый файл;

[Name] = File; – нетипизированный файл,

здесь [Name] – имя файлового типа (идентификатор); File of – зарезервированные слова (файл из); Text – имя

стандартного типа текстовых файлов; [Type] – любой тип, кроме файлового. Более подробно о фалах в разделе Файловый ввод-вывод.

СТРУКТУРЫ СИ

В ОТЛИЧИЕ ОТ МАССИВОВ, ВСЕ ЭЛЕМЕНТЫ КОТОРЫХ ДОЛЖНЫ БЫТЬ ОДНОГО ТИПА, СТРУКТУРЫ ОБЪЕДИНЯЮТ В ОДНОЙ ПЕРЕМЕННОЙ ЭЛЕМЕНТЫ РАЗНЫХ ТИПОВ.

Структуры конструируются следующим образом:

```
Struct имя_структуры
  {тип_1 поле_1;
   тип_2 поле_2;
   …
   тип_n поле_n;
        };
```

Описание переменной структурного типа выглядит следующим образом:

Struct struct_name x,y,z.

Структурным переменным можно присваивать значение агрегатно (сразу всей структуре), но их нельзя сравнивать на равенство или неравенство. Для сравнения структур следует писать специализированные функции, зависящие от приложения. К структуре можно применить операцию & (получение адреса) вычисления ее адреса.

&struct_name.

К полям структуры можно обращаться, указывая имя переменной, имя поля и разделяя эти имена точкой. Например:

```
Type_1 value1 = x.field_1;
Type_3 value3 = z.field_3;
```

Если переменная ptr определена как указатель на структуру, то для доступа к полям структуры можно использовать операцию "→" вместо " *ptr ".

Например:

```
Struct struct_name *ptr;
Type_1 value1 = ptr→field_1;
Type_2 value2 = ptr→field_2;
```

Оператор ptr→field_1эквивалентен выражению (*ptr).field_1.

В тех случаях, когда допускается использовать унарную операцию & получения адреса для структуры в целом, можно эту же операцию использовать и для получения адреса компонента структуры.

На стыке соседних полей структуры из-за необходимости выравнивания границ памяти под такие элементы могут возникать дыры, или пустоты. Возникновение пустот может привести к нарушению мобильности программ, сказывающему при переходе от одной реализации СИ к другой.

При вызове системы Turbo-СИ необходимо указать ключ – a, тогда компилятор должен выравнивать структуры (или объединения) по границе слова, добавляя к структуре дополнительные байты.

Указание ключа – a гарантирует следующее:

1 Структура будет начинаться с границы слова (четный адрес).

2 Каждое не символьное поле будет начинаться со сдвигом на четное число байтов от начала структуры.

3 В конце структуры будет (по необходимости) добавлен один байт, с тем чтобы вся структура содержала четное число байтов.

Имя-типа-структуры – это идентификатор, который именует тип структуры, определяемый списком объявлений элементов.

Описатель именует переменную структурного типа. В качестве описателя может быть имя переменной, модифицированной в указатель, массив или функцию.

Указатель рассматривается как указатель на структуру, массив – как массив структур, а функция – как функция, возвращающая структуру.

Список-объявлений-элементов содержит одно или несколько объявлений переменных или полей битов.

Каждая переменная, объявленная в этом списке, называется элементом структурного типа. Они могут быть любого основного типа, массивом, указателем, смесью или структурой.

Пример:

```
Struct teacher {
Char name[40];
Int number;
Float salary;}
list [20], *ptr;
Struct teacher_ school ();
```

В этом примере объявляется массив структурного типа teacher из 20 элементов, указатель на структуру типа teacher и функция, возвращающая структуру типа teacher.

Структура не может содержать в качестве элемента структуру такого же типа, но может включать указатель на структуру этого типа при условии, что в объявлении структуры указано имя типа. Это позволяет создавать связные списки структур.

Например:

```
Struct tree {int number;
        struct tree* left;
        struct tree* right;
           };
```

В Паскале данные этого типа принято называть записями.

З А П И С И

Запись – это структура данных, состоящая из фиксированного числа компонентов, называемых полями записи. В отличие от массива, компоненты (поля) записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля записи именуются.

Структура объявления типа записи имеет вид:

[имя типа] = record [список полей] end,

где [имя типа] – идентификатор, record, end – зарезервированные слова (запись, конец), [список полей]- список полей, представляет собой последовательность разделов записи, между которыми ставятся точка с запятой.

Каждый раздел состоит из одного или нескольких идентификаторов полей, отделяемых друг от друга запятыми. За идентификаторами ставится двоеточие и пишется тип поля, например:

```
Type
    BirthDay = record
    Day, Month, Year:Word
end;
var
    a,b : BirthDay;
```

В этом примере тип BirthDay (день рождения) есть запись с полями Day, Month, Year (день, месяц и год); переменные a,b содержат записи типа BirthDay.

Значения переменных типа запись можно присваивать другим переменным того же типа, например:

```
a:= b;.
```

К каждому из компонентов записи можно получить доступ, если использовать составное имя, указав имя переменной и через точку имя поля:

```
a.day:= 12;
b.year:= 1965;.
```

Для вложенных полей необходимо продолжать уточнения:

```
Type
    BirthDay = record
    Day, Month, Year:Word
end;
var
    p: record
    Name:String;
    Bd : BirthDay
end;
  begin
  if p.Bd.Year = 1965 then …
end;
```

Имена полей записи должны быть уникальными, в пределах той записи, где они объявлены, однако, если записи содержат поля-записи, то имена могут повторяться на разных уровнях (табл. 5).

5 С к а л я р н ы е т и п ы д а н н ы х

Обозначение типа данных		Длина в байтах	Диапазон значений
СИ	Паскаль		
int	Integer	2	-32768 ... +32767
long	LongInt	4	-2147483648...+2147483647
float	Real	4 6	3,4E-38 ... 3,4E38 2,9E-39..1,7E38
double	Double	8 8	1,7E-308 ... 1,7E308 5E-324 ... 1,7E308
long double	Extended	10	3,4E-493 … 1,1E4932
char	ShortInt	1	-128 ... +127
unsigned char	Byte	1	0 ... 255
unsigned	Word	2	0 ... 65535
short		2	-32768 ... +32767

Single	4	1,5E-45 … 3,4E38
comp	8	$-2^{63}+1$ ... $2^{63}-1$

Переменные, объявленные вне процедуры и функции, называются глобальными переменными и располагаются в сегменте данных. Переменные, объявленные в самой процедуре или функции, называются локальными переменными и располагаются в сегменте стека (подробнее в разделах: Указатели и Классы памяти).

К О Н С Т А Н Т Ы

Константами называют неизменные величины в программе. В СИ различают четыре типа констант: целые, с плавающей точкой символьные и стороковые. В Паскале различают именнованные константы и константы без имени, обращение к которым осуществляется двумя разными способами.

Константы без имени используются непосредственно в виде ее значения, как операнд выражения, именнованная константа подобна переменной.

Именнованные константы подразделяются на обычные и типизированные. Первые имеют неявно назначенные типы, которые компилятор распознает автоматически, типы вторых устанавливает программист (табл. 6).

6 Типы констант

Обычные константы	Типизированные константы
Const Min = 0; Max = 100; Ctnter = (Max – Min) div 2;	Const Max: Integer = 9999; Delta: Integer = 1000; Min: Integer = Max-Delta;

Константы обоих видов могут иметь любые типы кроме файлового. Объявление констант могут располагаться в разделе объявлений в произвольном порядке и количестве.

Значения выражений вычисляются на шаге компиляции и по типу полученного результата устанавливается тип константы.

С О В М Е С Т И М О С Т Ь Т И П О В

В выражениях и операциях сравнения требуется обеспечить совместимость типов.

Значение типа T2 является совместимым по присваиванию с типом T1 (т.е. допустимо T1: = T2), если выполняется одно из следующих условий:

- T1 и T2 имеют тождественные типы, и ни один из них не является файловым типом или структурным типом, содержащим компоненту с файловым типом на одном из своих уровней;
- T1 и T2 являются совместимыми порядковыми типами и значения типа T2 попадают в диапазон возможных значений T1;
- T1 и T2 являются вещественными типами и значения типа T2 попадают в диапазон возможных значений T1;
- T1 является вещественным типом, а T2 является целочисленным типом;
- T1 и T2 являются строковыми типами;
- T1 является строковым типом, а T2 является типом Char;
- T1 является строковым типом, а T2 является упакованным строковым типом;
- T1 и T2 являются совместимыми упакованными строковыми типами;
- T1 и T2 являются совместимыми типами множеств, и все значения типа T2 попадают в диапазон возможных значений T1;
- T1 и T2 являются совместимыми типами указателей;
- T1 и T2 являются совместимыми процедурными типами;
- T1 является типом процедурным типом, а T2 является процедурой или функцией с тождественным типом результата, с идентичным числом параметров и тождественностью между типами параметров;
- Объект типа T2 совместим по присваиванию с объектом типа T2, если T2 находится в области определения T1;
- Указатель типа P2, указывающий на тип объекта T2, совместим по присваиванию с указателем типа P1, указывающим на тип объекта T1, если T2 лежит в области определения T1.

На этапе компиляции и выполнения выдается сообщение об ошибке, если совместимость по присваиванию необходима, а ни одно из условий предыдущего списка не выполнено.

С е м ь о с н о в н ы х п о н я т и й п р о г р а м м и р о в а н и я

Для написания программы необходимо знать, как осуществляется:

- ввод-вывод;
- арифметические и логические операторы;
- упорядочение команд (условия, циклы, подпрограммы);
- принципы структурного прграммирования;
- динамическое распределение памяти;
- файловую систему;
- основы объектно-ориентированного программирования.

Все языки программирования, включая СИ и Паскаль, имеют свои особенности, но когда необходимо быстро изучить новый язык, можно посмотреть, как он реализует эти элементы, и начать работать.

Программа не имеет смысла, если не выводит какую-либо информацию. Вывод обычно принимает форму, которая зависит от выходного устройства: на экран (слова и изображения), на запоминающие устройства (дискеты и винчестер), в порты ввода/вывода. Операторы и функции ввода-вывода приведены в табл. 7.

7 ВВОД-ВЫВОД

П А С К А Л Ь	**С и**
В Ы В О Д Writeln (элемент, элемент, ...); Write (элемент, элемент, ...);	**В Ы В О Д** Fprintf (файл, "текст с форматами вводимых элементов", список элементов); Printf ("текст с форматами вводимых элементов", список элементов); Sprintf (строка, "текст с форматами вводимых элементов", список элементов);
В В О Д Read (элемент, элемент, ...); Readln (элемент, элемент, ...);	**В В О Д** Fscanf (файл, "текст с форматами вводимых элементов", список адресов элементов); Scanf ("текст с форматами вводимых элементов", список адресов элементов); Sscanf (строка, "текст с форматами вводимых элементов", список адресов элементов);

Подробнее о выводе и вводе в Паскале.

Назначение Writeln – запись (вывод) информации на экран. В формат процедуры:

Writeln (элемент, элемент, ...);

Элементом может быть:

- значение целое или вещественное (writeln (3);)
- символ (writeln (’a’);)
- строкой (writeln (’привет мир’);)
- булевским значением (writeln (false);)
- именованной константой (writeln (a);)
- функцией (writeln (cub);)
- указателем (writeln (Aptr^);).

Все элементы печатаются в строку в заданном порядке. После вывода курсор устанавливается на начало следующей строки. Если есть необходимость оставить курсор в этой же строке после последнего элемента, то используется:

Write (элемент, элемент, ...);.

При выводе элементов с помощью процедуры Writeln между ними автоматически пробелы не вставляются. При желании иметь их необходимо учесть:

Writeln (элемент, ' ', элемент, ' ', ...);

Пример:

В программе:	На экране:
Var A, B, C: Integer; Name: string; Begin A:= 1; B:= 2; C:= 3; Name := 'Frank'; Writeln (A, B, C); Writeln (A, ' ', B, ' ', C); Writeln ('Hi', Name); Writeln ('Hi, ', ' Name.');	 123 1 2 3 HiFrank; Hi, Frank.

Можно использовать параметры определения ширины поля для данного элемента. В этом случае оператор имеет формат:

Writeln (элемент: длина, ...);

где длина – целое выражение (литерал, константа, переменная, вызов функции), определяющее общий размер поля для вывода элемента.

Рассмотрим следующую программу и полученный в результате вывод:

В программе:	На экране:
A:= 10; B:= 2; C:= 100; Writeln (A, B, C); Writeln (A:2, B:2, C:2);	 102100 10 2100

Writeln (A:3, B:3, C:3); 10 2100
Writeln (A, B:2, C:4); 10 2 100.

Элемент дополняется начальными пробелами слева в соответствии с указанной длиной. Само значение выравнивается справа.

Если размер поля задан меньше, чем необходимо, то при выводе Паскаль увеличивает размер до минимально необходимого. Во втором операторе Writeln вышеприведенного примера для C = 100, длина поля меньше, чем нужно, т.е. задано 2, а нужно 3.

Этот метод применим для всех допустимых элементов: целого типа, вещественных чисел, символов, строк и булевских типов. Однако, при указании ширины (размера) поля для вещественных чисел выравнивание происходит слева и распечатывается в экспоненциальной форме.

В программе: На экране:
x:= 421.53;
Writeln (x); 4.2153000000E+02;
Writeln (x:8); 4.2E+02.

Поэтому, Паскаль добавляет второй операнд длины:
элемент : длина : количество цифр.

Вторая цифра указывает, сколько цифр выводить для числа с фиксированной точкой после точки:

В программе: На экране:
x:= 421.53;
Writeln (x:6:2); 421.53
Writeln (x:8:2); 421.53
Writeln (x:8:4); 421.5300.

В стандартном Паскале есть две основных процедуры ввода информации Read и Readln, которые используются для чтения данных с клавиатуры.

Их формат:
Read (элемент, элемент, ...);
Readln (элемент, элемент, ...);,
где каждый элемент – это переменная целого, вещественного, символьного типа или строка. Числа должны отделяться друг от друга пробелами или нажатием клавиши Enter.

Пример :

```
Uses Crt; {Включение модуля}
var
      a : integer;      {описание переменной a-целая}
      b : real;         {описание переменной b-вещественная}
Begin                        {начало тела программы}
     ClrScr;          {очистка экрана}
     Write (‘Введите значение переменной a =’);{вывод сообщения на экран}
     ReadLn (a);      {ввод значения переменной}
     b:= a/3;{оператор}
     WriteLn ('b =', b:5:3);{вывод}
ReadLn;{результат работы программы будет на экране до тех пор, пока не будет нажата клавиша Enter}
End.{конец тела программы}
```

ВЫШЕ ПРИВЕДЕН ПРИМЕР ПРОГРАММЫ НА ПАСКАЛЕ С ИСПОЛЬЗОВАНИЕМ ОПЕРАТОРОВ ВВОДА-ВЫВОДА.

Программа, написанная на СИ, использующая функции ввода-вывода, должна включать в себя файл stdio.h с помощью команды препроцессора

#include <stdio.h>

Файл stdio.h содержит:
1) определение типа данных FILE;
2) определение параметров, используемых в макровызовах и вызовах библиотечных функций ввода-вывода.

Форматизированный ввод и вывод осуществляют два семейства функций: scanf обеспечивает форматизированный ввод, а семейство printf – форматизированный вывод.

fscanf – форматизированный ввод из файла,
scanf – форматизированный ввод из потока stdin (клавиатуры),
sscanf – форматизированный ввод из строки СИ,
fprintf – форматизированный вывод в поток,
printf – форматизированный вывод в поток stdout,
sprintf – форматизированный вывод в строку СИ.

Поток – это либо файл на диске, либо физическое устройство, например, дисплей или печатающее устройство.

Пример программы использующей, функции ввода-вывода на Си:

```
#include <stdio.h> /*включение библиотеки*/
                                        main ()/* главная функция программы*/
{
int a; /* описание переменной a -целая*/
double b;/*описание переменной в-вещественная*/
printf ("введите целое число");/*вывод текста на экран*/
scanf ("%d", &a); /*ввод переменной a*/
b = a/3; /*оператор*/
printf ("\N b = %lg ", b); /*вывод значения переменой b*/
getch (); /*задержка экрана*/
```

}

При вызове любой функции в СИ за именем функции в круглых скобках указывается, что в нее передается. В первую функцию printf передается текстовая строка, которая будет печататься на экране монитора. Строка в СИ заключается в двойные кавычки. Во второй функции printf в выводимой строке присутствует непечатаемый управляющий код \n, обозначающий перевод курсора на новую строку и формат выводимой переменной b – %lg, сама переменная указавается за строкой после запятой. Значение переменной на экране будет печататься в строке на месте символа формата. В случае, если необходимо напечатать несколько переменных, то символы формата должны следовать в строке в той-же последовательности, что и переменные, следующие за строкой и перечисляемые через запятую. *Например:* printf ("a=%i b=%lg", a, b); символы формата должны соответствовать типам переменных (табл.7, 8 и 9).

8 Символы формата ввода- вывода в СИ

СИМВОЛ ФОРМАТА	Тип выводимого объекта
%c	для типа char
%s	символ типа string
%d или %i	тип int
%o	тип int в восьмеричном виде
%u	unsigned int
%ld	long в десятичном виде
%x	int в шестнадцатеричном виде
%lo	long в восьмеричном виде
%lu	unsigned long
%lx	long в шестнадцатеричном виде
%f	float double с фиксированной точкой
%e	float double в экспоненциальной форме
%g	float double в виде F или E в зависимости от значения
%lf	long float с фиксированной точкой
%le	long float в экспоненциальной форме
%lg	long float в виде F или E в зависимости от значения

Каждой переменной в операции ввода-вывода должна соответствовать спецификация соответствующего типа. Если используются несколько переменных, то всем им должны соответствовать спецификации соответствующих типов.

9 Управляющие коды ESC последовательности в СИ

\n	Новая строка
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\b	Возврат на символ
\r	Возврат на начало строки
\a	Звуковой сигнал

Операторы присваивания

Оператором называется элементарная структурная единица программирования.

Основной операцией является операция присваивания. В Паскале операция присваивания – это комбинация двоеточия и знака равенства: ":=". В СИ это просто знак равенства.

Пример:

Пусть даны две переменные x = 2 и y = 5, требуется поменять в памяти их значения местами так, чтобы x = 5, а y = 2. Для этого берется третья переменная z и производятся следующие действия:

1) z:=x;
2) x:=y;
3) y:=z;

Если произвести присвоение без использования z, x:=y, то содержимое ячейки памяти переменной x пропадает. При этом необходимо помнить, что присвоение идет справа налево, а не наоборот.

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ И ОПЕРАТОРЫ

Все операции присваивают переменным результат вычисления выражения. Это: сложение (+), вычитание (-), деление (/) и умножение (*), а также % для СИ mod для Паскаля – которая дает остаток от деления. Все операции в СИ, за исключением остатка от деления, определены для переменных типа int, float, char. Остаток не определен для типа float. В Паскале существует операция div –целочисленное деление (i div j- результат равен частному от деления, округленному до целого). Также в Паскале, поддерживаются унарные операторы. Используются следующие операции:

a+ = b a:= a + b;

a-= b a: = a – b;

a*= b a:= a*b;

a/ = b a:= a/b;

+ + a – увеличивает значение a на 1 до использования этой переменной;

a + + – увеличивает значение a после использования этой переменной в выражении;

– – a – уменьшает значение a на 1 до использования этой переменной;

a – – – уменьшает значение a после использования этой переменной в выражении;

sizeof (в скобках указывается тип или переменная) – можно применять к константам. В результате будет получено число байт, запрашиваемых оператором.

Операторы отношений сравнивают два значения, возвращая в результате булевское значение True или False в Паскале и 1 или 0 в СИ.

> – больше, чем;

>= – больше, чем или равно;

< – меньше, чем;

<= – меньше, чем или равно;

= в Паскале, = = в СИ – проверка на равенство;

<> в Паскале, != в СИ – проверка на неравенство;

Приоритет

Приоритеты арифметическмх операций такие же, как и в математике. Изменить приоритеты позволяют круглые скобки.

Приоритет одинаков у <= и >=, но выше, чем у = = и !=. *Например:*

If ((ch=getchar ())>'a') функция getchar возвращает символ из входной строки. У логических операций приоритет самый низкий.

Некоторые стандартные математические функции

10 Паскаль

Функция	Назначение
Round (x)	округление значения вещественного типа до целого
Trunc (x)	преобразование значения вещественного типа в ближайшее меньшее целое
Abs (x)	абсолютное значение аргумента
ArcTan (x)	определение арктангенса x
Cos (x)	определение косинуса x
Exp (x)	определение экспоненты аргумента
Frac (x)	определение дробной части аргумента
Int (x)	определение целой части
Ln (x)	определение натурального логарифма
Sqrt (x)	корень квадратный
Sqr (x)	квадрат аргумента
Dec (x)	уменьшение переменной на 1 или заданное значение параметра N
Inc (x)	увеличение на 1

В Паскале для возведения в степень используется следующее выражение: $x^y \Leftrightarrow \exp(y * \ln(x))$. Записи некоторых математических выражений:

Пример:

$x^{100} \Leftrightarrow \exp(100 * \ln(x));$

$\sqrt{1 + x^2} \Leftrightarrow \mathrm{sqrt}(1 + \mathrm{sqr}(x));$

$\cos^2 x^3 = 3\mathrm{qr}(\cos(x * x * x));$

$\log_2 \frac{x}{5} \Leftrightarrow \ln(x/5)/\ln(2).$

11 СИ

Функция	Назначение
Abs (x)	абсолютное значение аргумента
ArcTan (x)	определение арктангенса x
Cos (x)	определение косинуса x
Exp (x)	определение экспоненты аргумента
Log (x)	определение натурального логарифма
Sqrt (x)	корень квадратный
Pow (x, y)	возведение основания x в степень y

Блоки и составные операторы

Любая последовательность операторов в СИ, заключенная в фигурные скобки { } называется *составным оператором или блоком*. Внутри блока каждый оператор заканчивается;. Составной оператор используется везде, где синтаксис языка допускает использование простого оператора.

В Паскале составной оператор находится между словами begin и end.

Пустой оператор

Пустой оператор в этих языках представляется символом. Пустой оператор используется там, где синтаксис языка предусматривает присутствие оператора, а по логике программы оператор должен отсутствовать. В СИ необходимость в использовании пустого оператора часто возникает при программировании циклов, когда действия, которые могут быть выполнены в теле цикла, целиком помещаются в заголовок цикла.

12 ЛОГИЧЕСКИЕ ОПЕРАЦИИ И ОПЕРАТОРЫ

Логические операторы:	Паскаль	Си
и	and	&&
или	or	||
не	not	!

Условные операторы и конструкции выбора

Это выполнение одной или набора команд, если выполняется некоторое условие (и, если условие не выполняется, то эти команды пропускаются или выполняется другой набор команд) или если элемент данных имеет указанное значение или диапазон значений.

В СИ все выражения, реализующие условия для каждой конструкции выбора должны заключаться в круглые скобки (). Операторы ветвления в СИ и Паскале приведены в табл. 13

13 Операторы ветвления в СИ и Паскале

Паскаль	**С И**
if условное выражение then оператор 1;	if (условное выражение) оператор1;

If условное выражение then оператор 1
else оператор 2;
	if (условное выражение)
	оператор1;
	else
	оператор 2;

Case выражение of константное выражение 1: оператор 1; константное выражение 2: оператор 2; константное выражение 3: оператор 3; константное выражение n: оператор n; end; {case}	switch (выражение) { case константное выражение 1: оператор 1; brak case константное выражение 2: оператор 2; brak case константное выражение 3: оператор 3; brak case константное выражение 4: оператор 4; brak case константное выражение n: оператор n; brak

	default: операторы; }
Нет аналога	условная операция – ?

Стиль программирования. Операторы находящиеся после заголовка условия, переносятся на новую строку и сдвигаются на две позиции относительно заголовка.

Операторы if/else применяются, когда необходимо выполнить часть программы, если заданное условие имеет значение True или False в Паскале в СИ (1)-истинно или (0) – ложно, или когда заданное выражение принимает определенное значение.

Оператор if:
if выражение then оператор 1
else оператор 2,

где выражение – любое булевское выражение (вырабатывающее в результате True или False); оператор 1 и оператор 2 – операторы Паскаля. Если выражение принимает значение True, то выполняется оператор 1; в противном случае – оператор 2.

Оператор может быть составным, т.е. состоять из нескольких операторов (смотри блок операторов), пустым (смотри пустой оператор). Если в заголовке выражение условного оператора вырабатывает нулевое значение, то управление передается следующему оператору. Если выражение вырабатывает не нулевое значение, то выполняется оператор, стоящий после условного выражения. В операторе Паскаля перед словом else нет точки с запятой, а синтаксис СИ требует в этом месте точку с запятой и условное выражение, ограниченное круглыми скобками.

if (условное выражение)
оператор 1;
else оператор 2;

Синтаксис языка СИ предусматривает, что else всегда относится к ближайше-
му if.

Пример:

```
    int i = 4, j = 6, k = 8;
if (i < k)
if (i > j)
     printf ("оператор 1");
else printf ("оператор 2");
```

Поскольку в первом выражении – истина, а во втором – ложь, то программа напечатает оператор 2.

Два важных момента, на которые следует обратить внимание при использовании if/then/else.

Во-первых, оператор else не является обязательным, другими словами, допустимо использовать оператор if в следующем виде:

if выражение
then оператор 1
или

```
if (условное выражение)
оператор 1;
```

В этом случае оператор 1 выполняется только тогда, когда выражение имеет значение True. В противном случае оператор пропускается 1 и выполняется следующий оператор.

Во-вторых, если необходимо выполнить более одного оператора, в случае, когда выражение принимает значение, True или False, то следует использовать составной оператор.

```
if B = 0.0 then
      Writeln ('деление на нуль невозможно.').
else
  begin
      Ratio: = A / B;
      Writeln ('Отношение =' , Ratio)
end;
```

В данном *примере* используется один оператор в предложении if и составной в else.

О П Е Р А Т О Р В Ы Б О Р А C A S E В П А С К А Л Е

Оператор case в Паскале – мощное средство выбора альтернатив. Позволяет уменьшить количество операторов if.

Оператор case состоит из выражения (селектора) и списков операторов, каждому из которых предшествует метка того же типа, что и селектор. Это значит, что в данный момент выполняется тот оператор, у которого значение совпадает с текущим значением селектора. Если совпадения значений не происходит, то не выполняется ни один из операторов, входящих в case или же выполняются операторы, стоящие после необязательного слова else.

Метка case состоит из любого количества констант или диапазонов, разделенных запятыми, за которым следует двоеточие.

Например:

$$y = \begin{cases} x, & 1 \le x \le 5; \\ \cos(x), & 6 \le x \le 10; \end{cases}$$

```
case x of
   1..5: y:= x;
   6..10: y:= cos(x);
end; {case}
```

Диапазон записывается в виде двух констант, разделенных двумя точками «..». Тип константы должен соответствовать типу селектора. Оператор, стоящий после двоеточия, выполняется в том случае, если значение селектора совпадает со значением константы или, если его значение попадает в диапазон значений.

Оператор выбора switch в СИ

```
switch (выражение)
{
  case константное выражение 1: оператор 1; break;
  case константное выражение 2: оператор 2; break;
.
.
case константное выражение n: оператор n; break;
default: операторы;
}
```

Конструкция switch заменяет разветвленный многократный оператор if else. Из-за нечеткости конструкции if else в языке СИ при программировании блоков условий больше усилий тратится на преодоление трудностей языка, а не на выражение логики программы. Поэтому многократное использование оператора if else принято заменять на оператор switch.

После выполнения выражения в заголовке оператора его результат последовательно сравнивается с константными выражениями, начиная с самого верха до тех пор, пока не будет установлено их соответствие. Как только соответствие установлено начинают выполняться операторы внутри соответствующей метки case. Если среди операторов отсутствует оператор break, то управление переходит на следующее константное выражение и проверка продолжается. Для того чтобы этого избежать, операторы каждой метки заканчивают оператором break. После выполнения последовательности операторов одной ветви case, которая завершается оператором break, происходит выход из оператора switch. Обычно switch применяют тогда, когда хотят, чтобы была выполнена одна из последовательностей операторов из нескольких возможных. Каждая последовательность операторов может содержать ноль или более операторов. Фигурные скобки в этом случае не ставятся.

Пример:

```
Switch (n)
{
   case 1: printf ("n = %d", n); break;
   case 2: printf ("n = %d", n+n); break;
   case 3: printf ("n = %d", n*n); break;
   default printf ("n не принадлежит [1;3]");
}.
```

Ветка default может отсутствовать. Если она есть, то последовательность операторов, стоящих после default выполняется тогда, когда сравнение со всеми константными выражениями ложно.

Оператор go to

Используется для передачи управления внутри программы от одного оператора к другому. В языках СИ и Паскаль существуют мощные операторы ветвления и циклов, позволяющие избежать использования данного оператора, затрудняющего работу компилятора по оптимизации программы. Применение этого оператора считается плохим стилем программирования, а сам оператор называют "позор программиста".

Синтаксис: goto идентификатор;

Управление безусловно передается на оператор программы, помеченной тем же идентификатором (меткой).

При его использовании следует:

не входить внутрь блока из вне;

не входить внутрь операторов if, if/else, switch;

не входить внутрь итерационной структуры операторов цикла.

Условная операция ? в СИ

Эта операция используется только в СИ вместо конструкции if/else, если входящие в нее операторы являются простыми выражениями.

результат = (выражение) ? выражение 1: выражение 2;

Пример использования:

```
int i = 2, j = 6, result;
 result = (i<j) ? i : j;
```

printf ("%d", result);

Переменной result будет присвоено значение i если выражение истинно, j – ложь.

Циклы

Циклы, или итерационные структуры (табл. 14) позволяют выполнять повторение отдельных операторов. Число повторений в некоторых циклах фиксировано, а в других определяется в процессе счета одной или нескольких проверок условия.

Типовые конструкции вычислительных алгоритмов с циклами

1 *Рекурсивные алгоритмы.* Циклические алгоритмы, в которых значение некоторой функции на каждом этапе вычислений зависит от значений этой функции, полученной на предыдущем шаге, называется рекурсивным алгоритмом. В основе рекурсивного алгоритма лежит рекуррентная формула. Наиболее простой ее вид: x=x+1.

2 *Итерационный цикл.* Его особенность – то, что число повторения операций в теле цикла заранее неизвестно. На каждом шаге вычисления происходит последовательное приближение и проверка условия достижения искомого результата. Выход из цикла осуществляется в случае выполнения данного условия.

3 *Вложенные циклы.* Возможны случаи, когда внутри одного цикла необходимо повторить некоторую последовательность операторов, т.е. организовать внутренний цикл. Такая структура получила название цикла в цикле или вложенного цикла. Глубина вложения циклов может быть различной. При использовании такой структуры для экономии машинного времени необходимо выносить из внутреннего цикла во внешний все операторы, параметры которых не зависят от внутреннего цикла.

14 Операторы циклов в Си и Паскале

Паскаль	**Си**
While условное выражение do оператор;	While (истинное условное выражение) оператор;

for с шагом 1 (инкрементный)
for индекс:= выражение1 to выражение2 do оператор;
for с шагом –1 (декриментный)
for индекс:= выражение1 downto выражение2 do оператор;
repeat
операторы
until ложное условное выражение
	for (выражение инициализации;
	условное выражение;
	выражение итерации)
	оператор;
	do
	операторы;
	while (истинное условное выражение);

Стиль программирования. Операторы тела цикла располагаются под заголовком и сдвигаются на две позиции относительно него.

Циклы с предусловием

Цикл while

Проверка условия в цикле while производится перед выполнением тела цикла. Если результат условного выражения TRUE не равен нулю (в СИ), т.е. выражение истинно, то выполняется блок операторов тела цикла. Перед входом в цикл while первый раз обычно инициализируют одну или несколько переменных, чтобы условное выражение имело значение. Операторы тела цикла должны изменять значения одной или нескольких переменных условного выражения с тем, чтобы условное выражение обратилось в 0 (в СИ) или приняло значение FALSE (в Паскале) и прекратилось выполнение тела цикла. Цикл while завершается в следующих случаях: если обратилось в ноль условное выражение заголовка цикла; если в теле цикла использован оператор break; если использован оператор return – возврат из функции. В первых двух случаях управление передается на оператор, расположенный непосредственно за циклом, в третьем случае происходит возврат из функции. Чтобы прекратить выполнение программы бесконечного цикла надо нажать Ctrl+C. Если это не помогает, то попробовать Ctrl+Alt+Del.

Пример:

```
#include<stdio.h>     // библиотека ввода вывода
#include<conio.h>     // библиотека работы с окнами
```

```
main ()                    // главная функция (основной блок программы)
 {
 int Count; // описание переменных
 Count = 1; // инициализация для определения обоих частей условного выражения
 while (Count <= 10)
 {
       printf "Здравствуй и прощай!");
 Count++; // изменение значения переменной для организации следующего шага цикла
 }
       printf ("Это конец");
getch ();
 }
```

Частая ошибка зацикливания в СИ состоит в том, что вместо оператора сравнения на равенство = = используется =.

Полностью аналогично работает цикл while в Паскале, *например*:

```
program Hello; {имя программы}
 var
    Count: Integer; {описание переменной}
 Begin {начало основного блока программы}
      Count := 1; {присвоение начального значения переменной}
 while (Count <= 10) do
   begin
      Writeln ('Здравствуй и прощай!');
   Inc (Count); {изменение шага переменной для организации цикла}
   end;
   Writeln ('Это конец');
 end. {конец основного блока программы}
```

Во-первых, сначала переменной Count присвоится значение равное 1. Затем, при входе в цикл проверяется условие: значение Count меньше или равно 10. Если да, то выполняется тело цикла (операторы, находящиеся между ключевыми словами begin....end в Паскале и { } в СИ.) На экран выводится сообщение "Здравствуй и прощай". Значение Count увеличивается на 1. Выполняется возврат на начало цикла. Значение Count проверяется заново и тело цикла выполняется вновь, до тех пор пока значение переменной Count удовлетворяет условию. Как только значение Count становится равным 11, цикл завершается, и на экран выводится сообщение "Это конец".

Ц И К Л F O R В П А С К А Л Е

Вариант этого цикла в Паскале как эффективен, так и ограничен. Обычно, набор операторов выполняется фиксированное число раз, пока переменная (индексная) принимает значение в указанном диапазоне. Модифицируем знакомую программу Hello следующим образом.

```
program Hello
var
     Count: Integer;
begin
     for Count: = 1 to 10 do
     Writeln ('Здравствуй и прощай!');
     Writeln ('Это конец');
end.
```

При выполнении этой программы видно, что цикл for выполняется так же, как и цикл while.

Синтаксис цикла for:

for индекс:= выражение1 to выражение2 do оператор;

Здесь индекс – скалярная переменная (целого типа, символьного, булевского и любого перечислимого типа); выражение 1 и выражение 2 – выражения типа, совместимого с типом индекса; оператор – одиночный или составной оператор. Индекс увеличивается на 1 после каждого выполнения цикла. Индекс можно уменьшать на 1. Для этого ключевое слово to заменяется на downto.

Главный недостаток цикла for – это возможность уменьшить или увеличить индекс только на 1.

Основные преимущества – краткость, возможность использования символьного и перечислимого типа в диапазоне значений.

Ц и к л f o r в С И

Наиболее общей формой циклов в СИ является цикл for. Цикл for в СИ намного мощнее, чем в Паскале и других языках. Конструкция его выглядит следующим образом:

for ([выражение инициализации];[условное выражение];[выражение итерации])

оператор;

Выражение инициализации используется для установки начальных значений переменных, управляющих циклом.

Условное выражение – это выражение, при истинности которого операторы тела цикла будут выполняться.

Выражение итерации определяет изменение значений переменных, управляющих циклом.

Схема выполнения цикла for:

1 Выполняется выражение инициализации;

2 Вычисляется условное выражение;

3 Если значение условного выражения истина, то выполняются операторы тела цикла;

4 Выполняется выражение итерации;

5 Выполняется условное выражение и т.д.;

6 Как только условное выражение равно нулю, управление передается оператору, следующему за циклом.

Этот цикл относится к циклам с предусловием. Проверка условия всегда выполняется в начале цикла. Если условие ложно, то тело цикла может ни разу не выполниться.

Пример использования оператора for

```
for (i = 0;i<10; i ++)
printf ("Квадрат числа %d%d", i, i*i); // вычисляется квадрат чисел от 0 до 9.
```

Цикл for можно свести к циклу while (табл. 15).

15 Цикл wite

For (выражение 1;выражение 2;выражение 3) оператор;	выражение 1; while (выражение 2) {оператор; выражение 3;}

Допускается использовать в заголовке цикла несколько выражений инициализации и несколько выражений итерации. Это дает возможность повысить гибкость цикла.

Пример 1: программа, которая записывает строку в СИ в обратном порядке:

```
main ();
{
     int top, lot; char string[100] , temp;
 for (top=0, lot=100;top<lot;top++, lot--)
{temp=string[top];
     strring[top]=string[lot];
     string[lot]=temp;
}}
```

Пример 2: вывести все четные числа от 100 до 0 в порядке убывания.

```
1. for (i=100;i>=0;i-=2)
     printf ("%d", i);
2. for (i=100;i>=0;i--)
     if (i%2==1)continue;
     else printf ("%d", i);
3. for (i=100;i>=0;printf ("%d", i), i-=2);
```

Все три цикла делают одно и тоже, но написаны в трех разных вариантах.

For можно использовать и как бесконечный цикл, *например*:

```
for (;;)
{операторы;}
```

Для выхода из такого цикла обычно используется оператор break.

Пример цикла for, который может обеспечивать задержку:

```
for (i=0;i<1000;i++);
```

В других языках программирования в этом цикле нельзя сделать дробный шаг, т.е. параметр цикла for в Паскале может быть только целым, а в СИ вещественным.

Циклы с постусловием

Особенность: операторы тела цикла выполняются хотя бы один раз независимо от условия.

Цикл do while в СИ

Синтаксис:

```
do
     {оператор;}
 while (условное выражение);
```

Цикл do while прекращает выполняться, когда условное выражение равно нулю или становится ложным.

Схема выполнения цикла do while:

1 Выполняется оператор;

2 Выполняется выражение. Если выражение не равно нулю, то выполнение продолжается с первого пункта. Если выражение равно нулю, то выполняется оператор, следующий за циклом.

```
#include<stdio.h>                    // библиотека ввода-вывода
main ()                               // главная функция (основной блок программы)
{
     int A, B;                        // описание переменных (целое)
     float Ratio;                     // описание переменных (вещественное)
```

```
    char Ans;                          // описание переменных (символ)
do
{
    printf ("Введите два числа");      // вывод сообщения на экран
    scanf ("%i", &A);                  // ввод A
    scanf ("%i", &B);                  // ввод B
Ratio = A / B;                         // оператор
    printf ("\nОтношение равно %lg", Ratio); // вывод значения переменной
    printf ("\nПовторить? (Y/N)");     // вывод вопроса
    scanf ("%c", &Ans);                // ввод ответа
}
while ( Ans! = 'N');                   // конец цикла
}
```

Ц И К Л R E P E A T U N T I L В П А С К А Л Е

Цикл repeat until рассмотрим на примере программы, где повторение операторов зависит от ответа на вопрос.

```
program DoRatio;
var
    A, B: Integer;
    Ratio: Real;
    Ans: Char;
begin
  repeat
    Write ('ВВедите два числа');
    Readln (A, B);
    Ratio := A / B;
    Writeln ('Отношение равно', Ratio);
    Writeln ('Повторить? (Y/N)');
    Readln (Ans);
until Upcase (Ans) = 'N';
end.
```

В этой программе повторяется выполнение операторов, пока ответ на вопрос – N (Повторить? Y/N). Другими словами repeat и until, повторяются, до тех пор, пока значение выражения при until не будет True.

Синтаксис цикла:

```
repeat
оператор;
оператор;
.....
оператор;
until выражение.
```

Существуют три основных отличия этого цикла от цикла while:

- операторы в цикле repeat выполняются хотя бы один раз, потому что проверка выражения осуществляется в конце тела цикла. В цикле while, если значение выражения False, тело его пропускается сразу – цикл repeat выполняется пока выражение не станет True, в то время, как цикл while выполняется до тех пор, пока выражение имеет значение True. При замене одного типа цикла на другой необходимо на это обращать особое внимание. Рассмотрим программу HELLO, где цикл while заменен на цикл repeat:

```
program Hello; {начало программы}
var
  Count: Integer; {описание переменной (целая)}
Begin {начало основного блока программы}
   Count := 1; {присвоение начального значения}
Repeat {начало цикла}
  Writeln ('Здравствуй и прощай!'); {вывод сообщения на экран}
  Inc (Count); {увеличение значения переменной на 1}
until Count > 10; {конец цикла}
  Writeln ('Это конец'); {вывод сообщения на экран}
end. {конец программы}
```

Отметим, что теперь переменная Count проверяется на значение, большее 10 (а в while было Count <= 10).

В цикле repeat может использоваться просто группа операторов, а не составной оператор. При использовании этого цикла не используются слова begin...end, как в случае с циклом while.

Операторы цикла repeat выполняются хотя бы один раз, в то время, как операторы цикла while могут ни разу не выполниться в зависимости от значения выражения.

О п е р а т о р ы b r e k, c o n t i n u e, r e t u r n

break

Обеспечивает прекращение выполнения самого внутреннего из объемлющих его операторов таких, как switch, do while, while, for, repeat/until, case (т.е. немедленное прекращение работы цикла). После выполнения этого оператора управление передается оператору, следующему за прерванным.

continue

Работает подобно оператору break, но в отличии от него, continue прерывает выполнение тела цикла и передает выполнение на следующую итерацию (шаг).

Пример:

```
for (n=0;n<100;n++)
{
 if (n%2!=0) continue;
    printf ("%d", n);
 }
```

Когда n становится нечетной, выражение n%2!=0 получает значение 1 и выполняется оператор continue, который передает управление на следующую итерацию цикла for не выполняя оператор printf.

Оператор continue, также как и break прерывает самый внутренний из объемлющих его циклов (в случае вложенных циклов).

return

Завершает выполнение функции, в которой он задан и возвращает управление в вызывающую функцию в точку, следующую за вызовом функции.

return (выражение);

Значение выражения, если оно заданно возвращается в вызывающую функцию, в качестве значения вызывающей функции. Если в скобках выражение опущено, то управление автоматически передается в вызываемую функцию после выполнения последнего оператора функции (вызываемой). Возвращаемое функцией значение в этом случае не определенно. Если функция не возвращает значения, то ее следует объявлять типом void.

1) return используют: если надо выйти из функции;

2) если функция возвращает значение.

Пример 1:

```
int sum (int a, int b)
{
   return (a+b);
}
main ()
{
   int a=2, d=4;
   int s;
   s=sum (a, b);
 }
```

Пример 2 (функция, которая что-то печатает):

```
void print (char x)
{
 if (x==0)
 {
      printf ("Плохой аргумент");
      return;
 }
      printf ("Другой аргумент");
      printf ("%c", x);
}
```

Оператор return используется для выхода из функции printf, если аргумент равен нулю.

П Р О Ц Е Д У Р Ы И Ф У Н К Ц И И

С Т Р У К Т У Р А П Р О Ц Е Д У Р Ы И Ф У Н К Ц И И В П А С К А Л Е

В Паскале есть два вида подпрограмм: процедуры и функции. Главное различие между ними – это то, что функция возвращает значение и может быть использована в выражении:

```
x := sin (A);
```

в то время, как процедура только может быть вызвана:

```
writeln ("Это проверка");
```

Процедуры и функции, известные под общим именем как подпрограммы могут быть описаны в любом месте программы, но до тела главной программы. Формат процедур:

```
procedure имя процедуры (параметры);
label
   метки;
const
   объявление констант;
type
   определения типов данных;
var
   объявления переменных;
procedure или function;
begin
   тело главной процедуры;
end;
```

Функции имеют такой же формат, как и процедуры, только они начинаются с заголовка function и заканчиваются типом данных возвращаемого значения:

```
function имя функции (параметры): тип данных;
```

Имеются только два различия между программами и процедурами или функциями:

- процедуры или функции имеют заголовок procedure или function, соответственно, а не program;
- процедуры или функции заканчиваются точкой с запятой (;), а не точкой (.).

Процедуры или функции могут иметь описания своих констант, типов данных, переменных и свои процедуры и функции. Но все эти элементы могут быть использованы только в тех процедурах и функциях, где они объявлены.

Пример функции расчета факториала

```
Function fact (n: integer): real;
 f:=1;
for i:=1 to n do
 f:= f*i;
 fact:=f;
end.
```

Пример: процедуры расчета суммы ряда $S = \sum_{i=1}^{10} i$

```
Procedure sum;
   Var
        i: integer;
        S: real;
   begin
        S:=0;
   for i=1 to 10 do
        S:= S+i;
        Write (S);
   End.
```

Пример иллюстрирующий гибкость процедур и функций в Паскале.

Пример: процедуры получения двух значений и функции, определяющий их отношение:

```
program DoRatio;
var
     A, B: Integer;
     Ratio: Real;
procedure GetData (var X, Y: Integer);
begin
     Writeln ('Введите два числа:');
     Readln (X, Y);
end;
function GetRatio (I, J: Real);
begin
     GetRatio (I / J);
end;
 begin
     GetData (A, B);
     Ratio := GetRatio (A, B);
     Writeln ('Отношение равно ', Ratio);
end.
```

Этот пример показывает, как используются и работают процедуры и функции.

После компиляции и запуска программы первым выполняется оператор GetData (A, B). Этот оператор известен как вызов процедуры. При обработке вызова выполняются операторы в GetData, при этом X и Y (формальные параметры) заменяются на A и B (фактические параметры). Ключевое слово var перед X и Y в операторе вызова GetData говорит о том,

что фактические параметры должны быть переменными и что значения переменных могут быть изменены и возвращены вызывающей программе. При завершении работы GetData управление возвращается в главную программу на оператор, следующий за вызовом GetData.

Следующий оператор – вызов функции GetRatio. Отметим некоторые отличия. Во-первых, GetRatio возвращает значение, которое должно быть использовано; в этом случае, оно присваивается Ratio. Во-вторых, значение присваивается GetRatio в главной программе; этим функция определяет, какое значение возвращается. В-третьих, нет ключевого слова var перед формальными параметрами I и J. Это означает, что они могут быть любыми целочисленными выражениями, такими как Ratio:= GetRatio (A+B, 300); и что если даже их значения будут изменены в функции, то новые значения не возвратятся обратно в вызывающую программу.

Кстати, это не является отличием процедуры от функции. Можно использовать оба типа параметров для обоих типов программ.

Ф У Н К Ц И И В С и

В языке СИ нет понятия процедура. *Функцией* называется независимая совокупность объявлений и операторов, предназначенная для выполнения определенной задачи.

Каждая функция должна иметь имя, которое используется для вызова функции. Имя одной из функций языка СИ – main. Это имя зарезервировано и должно присутствовать в любой программе на СИ. В программе могут присутствовать и другие функции, при этом main не обязательно должна быть первой. Хотя именно с функции main начинается выполнение СИ-программ. При вызове функции ей могут быть переданы данные посредством аргумента функции. Функция может возвращать значения. Это значение и есть основной результат выполнения функции, который при выполнении программы подставляется на место функции. Вернуть можно только одно значение. Могут так же определяться функции, которые не имеют ни каких параметров и не возвращают никакого значения. С использованием функций в СИ связаны 3 понятия:

1 объявление функции;
2 определение функции;
3 вызов функции.

Объявление функции или прототип задает имя функции, тип и число ее параметров, объявления и операторы, которые определяют действия функции. В объявлении функции так же может быть задан тип значения возвращаемого функцией, а так же спецификатор класса памяти функции (см. раздел 5.11.1).

В языке СИ нет требования, чтобы определение функции обязательно предшествовало вызову функции. Однако, для того, чтобы компилятор мог выполнить проверку соответствия типов, передаваемых аргументу типом формальных параметров в определении функции, до вызова функции нужно поместить объявление этой функции. Это объявление называется прототипом. *Прототип* функции имеет тот же формат, что и определение функции с той лишь разницей, что не имеет тела функции и должен заканчиваться " ; ". Последовательность объявлений и операторов называется телом функции.

Если объявление функции не задано, то по умолчанию строится прототип на основании информации из первой ссылки на функцию, будь то вызов функции или определение. Однако, такой прототип может неадекватно представлять последующий вызов или определение функции. Задание прототипа позволяет компилятору выдавать диагностические сообщения об ошибке, либо корректным образом регулировать несоответствие аргументов.

Определение функции задает имя формальных параметров и тело функции. Оно может определять тип возвращаемого значения и класс памяти функции.

Синтаксис определения функции: [спецификатор класса памяти] [спецификатор типа возвращаемого функцией значения] имя функции ([типы и имена формальных параметров] {Тело функции}

***Класс памяти функции* может быть либо статик, либо экстерн. Если класс памяти функции не указан, то по умолчанию считается экстерн.**

Тип возврата ставится, когда функция возвращает значение. Если спецификатор типа не задан, то по умолчанию ставится int. Если функция не возвращает ни какого значения, то ставится void. Описатель функции может быть задан со звездочкой, это означает, что функция возвращает указатель. Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип.

Список формальных параметров – это последовательность объявления формальных параметров, разделенных запятыми. *Формальные параметры* – это переменные, которые принимают значения, передаваемые функцией во время вызова. Полное объявление формальных параметров функции задается в круглых скобках, следующих за именем функции в определении функции. Элемент списка формальных параметров имеет два формата.

1 Спецификатор типа;

2 Описатель – задается через запятые, они могут оканчиваться символами: , . . . – неопределенное число параметров. Это означает, что число аргументов функции переменно. Если функция не передает параметры, то пишется (void). Порядок и типы формальных параметров должны быть одними и теми же в определении функции и во всех объявлениях.

Пример. Рассмотрим три вида функции вычисления куба числа. Функция $y = x^3$.

1. Функции передается параметр и она возвращает параметр

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

```
        int cub (int x); // объявление функции;
        int cub (int x) // определение функции
{
        int y;
        y=x*x*x; – тело функции;
        return (y);
}
void main (void)
{
        int x;
        int y;
        scanf ("%d", f x);
 y=cub (x); – вызов функции;
        printf ("%d", y); getch ();
}
```

2 Функции передается параметр, но она ничего не возвращает

```
#include<stdio.h>
void cub (int x);               // объявление функции;
void cub (int x)                // определение функции;
{
        int y;
        y=x*x*x; // тело функции;
        printf ("%d", y);
}
void main (void)
{
        int x;
        int y;
        scanf ("%d", f x);
 cub (x); – вызов функции; getch ();
}
```

3 Функции ничего не передается и она ничего не возвращает

```
#include<stdio.h>
void cub (void);            // объявление функции;
void cub (void)             // определение функции;
{
        int y, x;
        scanf ("%d", &x); // тело функции;
        y=x*x*x;
        printf (''%d'', y);
}
void main (void)
{
 cub ();                    // вызов функции;
}
```

Тело функции – составной оператор, содержащий операторы, определяющие действия функции. Он так же может содержать объявления переменных, используемых в этих операторах соответственно. Параметры функции передаются по значению, и могут рассматриваться как локальные переменные, место для которых распределяется при вызове функции. Они инициализируются значениями переданных аргументов и теряются при выходе из функции. Поэтому в теле функции нельзя изменять значения параметров, так как функция работает с копиями аргументов. Однако, если в качестве параметра передать указатель переменной можно использовать операции раз адресации этой переменной (см. раздел 5.11.1). Для осуществления доступа и изменения значений этой переменной (см. табл. 16).

Так как параметры в функции, написанной неправильно, принимаемые функцией являются копиями передаваемых параметров, она обменивает значения этих параметров, не оказывая влияния на значение аргументов, и эта функция не выполняет своей задачи. Правильная работа функция заключается в том, что при выводе функции ей должны передаваться в качестве аргументов адреса передаваемых x и y, значения которых нужно поменять.

Пример.

ФУНКЦИЯ ОБМЕНА ДВУХ ПЕРЕМЕННЫХ

П Р А В И Л Ь Н О	**Н Е П Р А В И Л Ь Н О**
void chen (int*x, int*y)	Void chen (int x, int y)

{
int buf;
buf=*x;
*x=*y;
*y=buf;
}
	{
	int buf;
	buf=x;
	x=y;
	y=buf;
	}

Вызов функций имеет следующий формат:

имя функции ([список передаваемых параметров или список выражений]).

Список передаваемых параметров или список выражений представляет собой список фактических аргументов, которые посылаются в функцию. Фактически аргумент может быть любой величиной основного типа (структурой, указателем, перечислением), хотя массивы и функции не могут быть параметрами функции. Но указатели на массив могут быть переданы в функцию.

Выполнение вызова функции происходит следующим образом:

1 Вычисляется выражение. Затем, если известен прототип, тип результирующего аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо выдается диагностическое сообщение. Число выражений в списке выражений должно совпадать с числом формальных параметров. Если в прототипе функции вместо списка формальных параметров задано ключевое слово void, это означает, что в функцию не передается никаких параметров, и в определении функции не должно быть формальных параметров.

2 Происходит замена формальных параметров на фактические.

3 Управление передается на первый оператор функции.

4 Выполнение оператора return возвращает управление и возможно значение вызывающую функцию. Если оператор return не задан, то, после выполнения последнего оператора тела функции, управление возвращается в точку вызова.

Пример передачи массива в функцию как параметр:

```
#include<stdio.h>
    int data[2][2]={-12, 14, -10, 16};
main ()
{
 extern void modify (int *a, int size);
  int i;
//modify (data, 4);
for (i=0;i<4;i++)
   printf (" \ndata[%d]=%d\n", i, * (data+1));
}
void modify (int *a, int size)
{
   int i;
for (i=0;i<size;i++)
   a[i]++;
}
```

Пример передачи массива в функцию по значению.

```
#include<stdio.h>
#define size 5
 void main (void)
{
   int mas[size];
   int i, j;
   printf ("\n max=%d", max (mas, size));
   printf ("\n min=%d", min (mas, size));
}
int max (int data[], int s)
}
   int maximum=data[0];
   int i, j;
for (i=0;i<s;i++)
   if (data[i]>maximum) maximum=data[i];
   return (maximum);
}
int min (int data[], int s)
{
   int minimum=data[0];
   int i, j;
for (i=0;i<s;i++)
   if (data[i]<=minimum) minimum=data[i];
   return (minimum);
}
```

При задании массива в качестве параметра функции передается адрес первого элемента массива. Если в теле функции заменяются значения элементов массива, то изменяются непосредственно сам передаваемый массив.

Если в описании функции задано, что параметр передается по ссылке (т.е. он описан как указатель на тип), то в качестве параметра при вызове функции передается адрес переменной.

Р е к у р с и в н ы е в ы з о в ы

Любая функция на языках СИ и Паскаль может быть вызвана рекурсивно, то есть она может вызвать саму себя. Компилятор допускает любое число рекурсивных вызовов. При каждом вызове, для формальных параметров (переменная с классом auto (раздел 5.11.1)), выделяется новая память. Таким образом, их значение из предшествующих рекурсивных вызовов не затирается. Параметры предшествующих незавершенных вызовов недоступны. Переменные, объявленные классом памяти static, не требуют новой памяти.

Классический пример рекурсии – это математическое определение факториала

$$n! = \begin{cases} 1, & n = 0; \\ n \cdot (n-1)!, & n > 1. \end{cases}$$

```
long fact (int n)
{
return (n<=1)?1:n*fact (n-1);
}
```

Пример использования пустого оператора:

```
 int lin (char*str)
{
 int index = 0;
while (str[index++])
;
return (index);}
```

Стиль программирования: рекомендуется (;), относящуюся к пустому оператору, помещать на отдельной строке иначе при чтении программы можно не обратить внимание на пустой оператор.

Ф а й л о в ы й в в о д - в ы в о д

Ф а й л ы в С И

Язык СИ не содержит средств поддержки ввода/вывода. Каждая реализация СИ должна сопровождаться библиотеками и макросами, обслуживающими ввод/вывод. По мере развития СИ функции ввода/вывода становились стандартизированными. Такая стандартизация обеспечила высокий уровень мобильности программы на СИ.

Термин "поток" происходит из представления о последовательной структуре информационных записей. Состав потока задается структурой FILE, описание которой задается в файле stdio.h.

Макрос EOF определяется следующим образом:

#define EOF (-1)

Он в операциях ввода/вывода служит для обозначения и проверки конца файла.

Функция fopen используется для открытия потока (файла). Интерфейс с функцией fopen описывается следующим образом:

FILE fopen (char filename, char type);

В качестве первого параметра функции должно передаваться правильное имя файла.

Второй параметр определяет тип открываемого файла.

Допустимы следующие типы файлов:

"r" – открыть уже существующий файл на ввод;

"w" – создать новый файл или очистить уже существующий файл и открыть его на вывод;

"a" – создать новый файл для вывода или осуществить вывод в конец уже существующего файла;

"r+" – открыть существующий файл для обновления, которое будет проводиться с начала файла;

"w+" – создать новый или открыть существующий файл для обновления его содержимого;

"a+" – создать новый файл или подстроиться в конец существующего файла для обновления его содержимого.

Функция flclose выглядит следующим образом:

int fclose (FILE *stream);

С помощью этой функции файл закрывается.

Функция fseek описывается следующим образом:

int fseek (FILE *stream, long offset, int wherefrom); .

Эта функция служит для произвольного доступа к байтам, обычно внутри двоичных потоков.

Первый аргумент задает поток, к которому должен осуществляться прямой доступ.

Второй аргумент указывает число байт смещения от точки, определяемой третьим параметром функции.

Третий параметр указывает точку, от которой следует начинать отсчет смещения, заданного вторым аргументом.
Значение 0 – смещение от начала файла.
Значение 1 – смещение от текущей позиции файла.
Значение 2 – смещение от конца файла.

Пример программы на Си: Сформировать файл из некоторых чисел. Записать во второй файл количество положительных, отрицательных и нулевых элементов первого файла.

```
// Подключаемые библиотеки
#include<stdio.h>          // Библиотека ввода-вывода
#include<conio.h>          // Библиотека работы с окнами
#define N 6                // Макроопределение (размерность массива)

 void main (void)          // Главная функция (основной блок программы)
{
                           // Раздел объявления локальных переменных
 FILE *f1;                 // Указатель на первый файл
 FILE *f2;                 // Указатель на второй файл
        int mas[N];        // Массив, элементами которого заполняется первый файл
        int i, buf, nul, otr, pol; // i – переменная счетчика цикла
// buf – переменная, в которую будут считыватся значения из первого файла
// nul, otr, pol – переменные количества положительных, отрицательных и нулевых
//элементов первого файла соответственно
          clrscr ();       // Очистка экрана
              printf ("Составитель: Николаев Максим Петрович");
              printf ("\n\t\t Факультет КТФ, группа Р-12 \n\n"); // Заполнение массива
 for (i=0; i<N; i++)
 {
        printf ("Введите %i элемент массива: ", i);
        scanf ("%i", &mas[i]); // Ввод элементов массива
 }
 // Создание в текущем каталоге файлов с именами file1 и file2
 // и открытие их на запись
 f1=fopen ("file1", "w+");
 f2=fopen ("file2", "w+");
 // Запись в файл с именем file1 элементов массива mas[N]
 for (i=0; i<N; i++) fwrite (&mas[i], 1, sizeof (int), f1);
 fclose (f1); // Закрытие файла file1
        pol=0; // Присвоение начального значения переменным
        otr=0; // количества положительных, отрицательных и
        nul=0; // нулевых элементов
 // Открытие файла с именем file1 на чтение
 fopen ("file1", "r+");
 // Чтение из файла информации в переменную buf до тех пор,
 // пока не наступит конец файла
 while (fread (&buf, sizeof (int), 1, f1)!=0)
 {
     // Если очередное значение переменной buf положительное, то
     // значение переменной pol увеличивается на 1
 if (buf>0) pol=pol+1;
     // Если очередное значение переменной buf отрицательное, то
     // значение переменной otr увеличивается на 1
 if (buf<0) otr=otr+1;
     // Если очередное значение переменной buf равно нулю, то
     // значение переменной nul увеличивается на 1
 if (buf==0) nul=nul+1;
 }
 // Запись в файл file2 количества положительных, отрицательных
 // и нулевых элементов первого файла file1
     fprintf (f2,"В ПЕРВОМ ФАЙЛЕ СОДЕРЖИТСЯ:\n");
     fprintf (f2, "\n положительных элементов: %i", pol);
     fprintf (f2, "\n отрицательных элементов: %i", otr);
     fprintf (f2, "\n нулевых элементов: %i", nul);
```

```
// Вывод сообщения о завершении работы программы
    printf ("РАБОТА С ФАЙЛАМИ ЗАВЕРШЕНА");
    printf ("\n Нажмите любую клавишу");
    getch (); // Задержка экрана до нажатия любой клавиши
}
```

Файлы в Паскале

Файловая переменная в Паскале – это любая переменная файлового типа. В Паскале имеются три класса файлов: типизированный, текстовый, нетипизированный.

Перед использованием файловой переменной она должна быть связана с внешним файлом с помощью вызова процедуры Assign. Внешним файлом обычно является поименнованный файл на диске, но он также может представлять собой устройство, как например, клавиатуру или дисплей. Во внешних файлах сохраняется записанная в файл информация или они служат источниками информации, которая считывается из файла.

Пример:

```
Const
      Path = 'a:\MyDir';
Var
      MqFile: File of Integer;
       Temp : string.
          Begin
      Assing (MyFile, 'a:\MyDir\Data.dat');
      Assing (MyFile, Path+'Data.dat');
          Reset (MyFile);
 if IoResult=0 then
          Writeln ('файл готов к работе')
 Else
      Writeln ('неверно задано имя файла');
      Apprend (MyFile); {если не существует то создается}
 While not SeekEOF Do
          Begin
      Readln (Temp);{считывание строки с клавиатуры}
      Writeln (MyFile, Temp); {запись строки в файл.}
          end;
 close (MyFile);
 reset (MyFile);
 While not SeekEOF Do
          Begin
      Readln (MyFile, Temp); {считывание строки из файла}
      Writeln (Temp); {вывод строки на диске}
      end;
 close (MyFile);
 end.
```

Когда связь с внешним файлом установлена, для подготовки ее к операции ввода или вывода файловая переменная должна быть "открыта". Существующий файл можно открыть с помощью процедуры Reset, а новый файл можно создать и открыть с помощью процедуры Rewrite. Текстовые файлы, открытые с помощью процедуры Reset, доступны только для чтения, *а текстовые файлы*, открытые с помощью процедуры Rewrite и Append, доступны только для записи.

Типизированные и нетипизированные файлы всегда допускают как чтение, так и запись, независимо от того, были они открыты с помощью процедуры Reset или с помощью процедуры Rewrite.

Любой файл представляет собой линейную последовательность элементов, каждый из которых имеет сложный тип (или тип записи) файла. Каждый элемент файла имеет номер. Первый элемент файла считается нулевым элементом.

Обычно доступ к файлам организуется последовательно, то есть, когда элемент считывается с помощью стандартной процедуры Read или записывается с помощью стандартной процедуры Write, текущая позиция файла перемещается к следующему по порядку элементу файла. Однако к типизированным и нетипизированным файлам можно организовать прямой доступ с помощью стандартной процедуры Seek, которая перемещает текущую позицию файла к заданному элементу. Текущую позицию в файле и текущий размер файла можно определить с помощью стандартных функций FilePos и FileSize.

Пример работы с типизированными файлами

Произвольный доступ

```
Var MyFile: File of Integer;
```

```
    I, K, Temp: Integer;
  Begin
Assiqn (MyFile; 'data.dat');
    ReWrite (MyFile);
  for I:=1 to 5 Do
      Begin
        Write (MyFile, I); {запись числа в файле}
      for k: =I-1downto 0 Do
      Begin
        Seek (MyFile, K); {переустановка указателя файла}
        Read (MyFile, Temp); {считывание}
      end;
        Seek (MyFile, I); {восстановление указателя файла}
      end;
Close (MyFile);
end;
```

Результат работы программы записывается в файл data.dat

12345

121321432154321.

Работа программы по шагам.

i=1 k=0 1

i=2 k=1 2

k=0 1

i=3 k=2 3

k=1 2

k=0 1 и т.д..

Когда программа завершит обработку файла, он должен закрываться с помощью стандартной процедуры Close. После полного закрытия файла, связанный с ним внешний файл обновляется. Затем файловая переменная может быть связана с другим внешним файлом.

По умолчанию, при всех обращениях к стандартным функциям и процедурам ввода/вывода автоматически производится проверка на наличие ошибок. При обнаружении ошибки программа прекращает работу и выводит на экран сообщение об ошибке. С помощью директив компилятора {$I+} и {$I-} эту автоматическую проверку можно включить или выключить. Когда автоматическая проверка отключена, то есть когда процедура или функция была скомпилирована с директивой {$I-}, ошибки ввода/вывода, возникающие при работе программы, не приводят к ее останову. При этом, чтобы проверить результат выполнения операции ввода/вывода, нужно использовать стандартную функцию IOResult (табл. 17, 18).

17 ПРОЦЕДУРЫ ВВОДА/ВЫВОДА

Наименование	**Н А З Н А Ч Е Н И Е**
ChDir	Меняет текущий справочник
Erase	Удаляет внешний файл
GetDir	Возвращает текущий справочник на заданном диске
MkDir	Создает подсправочник
Reset	Открывает существующий файл
Rewrite	Создает и открывает новый файл
RmDir	Удаляет пустой подсправочник
Seek	Передвигает текущую позицию файла на указанную компоненту (не используется с текстовыми файлами)
Truncate	Усекает размер файла до текущей позиции в файле. (Не используется с текстовыми файлами)

1 8 Ф У Н К Ц И И В В О Д А / В Ы В О Д А

Наименование	Назначение
Eof	Функция EOF возвращает результат TRUE или FALSE в зависимости от достижения конца файла и потому может быть использована в условных выражениях.

FilePos	Возвращает текущую позицию в файле. (Не используется с текстовыми файлами)
FileSize	Возвращает текущий размер файла. (Не используется с текстовыми файлами)
IOResult	Возвращает целое значение, являющееся состоянием последней выполненной операции ввода/вывода.

П Р И М Е Р :

Сформировать файл из некоторых чисел. Записать во второй файл количество положительных, отрицательных и нулевых элементов файла.

```
uses CRT; {Подключение библиотеки ввода-вывода}
const n=10; {Максимальный размер массива}
{Раздел объявления переменных}
var
        File1, File2: FILE of Integer; {Переменные файлового типа}
        Road1, Road2: String[14]; {Строки для хранения имен файлов}
        a : Array[1..n] of Integer; {Массив для хранения введенных чисел}
        i, pol, otr, nul: Integer; {i-счетчик цикла}
{pol – количество положительных элементов массива}
{otr – количество отрицательных элементов массива}
{nul – количество нулевых элементов массива}
{Основной блок программмы}
begin
        ClrScr; {Очистка экрана}
        WriteLn ('Выполнил Николаев М.П., КТФ, гр. Р-12');
        Road1:='Test1.dat'; {Задание имен файлов}
        Road2:='Test2.dat';
Assign (File1, Road1); {Связь файловой переменной с внешним файлом Road1}
        Rewrite (File1); {Создание и открытие файла Test1.dat}
Assign (File2, Road2); {Связь файловой переменной с внешним файлом Road2}
        Rewrite (File2); {Создание и открытие файла Test2.dat}
        pol:=0; {Инициализация переменных}
        otr:=0;
        nul:=0;
for i:=1 to n do
        begin
            WriteLn ('Введите ', i, ' элемент массива'); {Запрос у пользователя  элементов массива}
            Read (a[i]); {Ввод элементов массива}
        Write (File1, a[i]); {Запись их в файл Test1.dat}
end;
Close (File1); {Закрытие файла Test1.dat}
        WriteLn; {Пропуск строки}
        WriteLn ('_____');
        RESET (FILE1); {ОТКРЫТИЕ ФАЙЛА TEST1.DAT ДЛЯ ПРОВЕРКИ ПРАВИЛЬНОСТИ ЗАПИСИ}
```

```
while not eof (File1) do {Чтение файла до тех пор, пока указатель}
{текущей компоненты файла находится перед}
{последней компонентой файла}
 begin
        Read (File1, a[i]); {Считывание чисел из файла}
        Write (a[i]:2); {Распечатка считываемых чисел}
 If a[i]>0 then pol:=pol+1; {Подсчет числа положительных, отрицательных}
if a[i]<0 then otr:=otr+1; {и нулевых элементов}
if a[i]=0 then nul:=nul+1;
  Inc (i); { Приращение счетчика цикла }
end;
        WriteLn; { Пропуск строки }
        WriteLn ('_____');
        Write (File2, pol); { Запись числа положительных, отрицательных }
        Write (File2, otr); { и нулевых элементов в файл Test2.dat }
        Write (File2, nul);
Close (File2); { Закрытие файлов Test1.dat и Test2.dat }
Close (File1);
Assign (File2, Road2); { Открытие файла Test2.dat }
        Reset (File2); { для проверки правильности записи и печати результатов}
        Read (File2, pol); WriteLn ('Количество положительных: ', pol:3);
        Read (File2, otr); WriteLn ('Количество отрицательных:', otr:3);
        Read (File2, nul); WriteLn ('Количество нулевых:', nul:3);
Close (File2); { Закрытие файла Test2.dat }
        ReadKey; { Задержка экрана }
```

end.

Далее рассмотрим текстовые файлы.

Текстовые файлы

В этом разделе описываются операции ввода/вывода, использующие файловую переменную стандартного типа Text. Заметим, что в Turbo Pascal тип Text отличается от типа file of char.

При открытии текстового файла внешний файл интерпретируется особым образом: считается, что он представляет собой последовательность символов, сгруппированных в строки, где каждая строка заканчивается символом конца строки (end of line), который представляет собой символ перевода каретки, за которым, возможно, следует символ перевода строки.

Для текстовых файлов существует специальный вид операций чтения и записи (Read и Write), которые позволяют считывать и записывать значения, тип которых отличается от типа Char. Такие значения автоматически переводятся в символьное представление и обратно. Например, Read (F, i), где i – переменная целого типа, приведет к считыванию последовательности цифр, интерпретации этой последовательности, как десятичного числа, и сохранению его в i.

Как было отмечено ранее, имеются две стандартные переменные текстового типа – Input и Output. Стандартная файловая переменная Input – это доступный только для чтения файл, связанный со стандартным файлом ввода операционной системы (обычно это клавиатура), а стандартная файловая переменная Output – это доступный только для записи файл, связанный со стандартным файлом вывода операционной системы (обычно это дисплей). Перед началом выполнения программы файлы Input и Output автоматически открываются, как если бы были выполнены следующие операторы :

```
Assign (Input, ''); Reset (Input);
Assign (Output, ''); Rewrite (Output);
```

Аналогично, после выполнения программы эти файлы автоматически закрываются.

Если программа использует стандартный модуль Crt, то файлы Input и Output не будут по умолчанию относиться к стандартным файлам ввода/вывода.

Для некоторых стандартных процедур, список которых приведен в данном разделе, не требуется явно указывать в качестве параметра файловую переменную. Если этот параметр опущен, то по умолчанию будут рассматриваться Input и Output, в зависимости от того, будет ли процедура или функция ориентирована на ввод или вывод. *Например:*

Read (X) соответствует Read (Input, X) и Write (X) соответствует

Write (Output, X).

Если при вызове одной из процедур или функций из этого раздела задается файл, то он должен быть связан с внешним файлом с помощью процедуры Assign и открыт с помощью процедуры Reset, Rewrite, или Append. Если для ориентированной на вывод процедуры или функции указывать на файл, который был открыт с помощью процедуры Reset, то выведется сообщение об ошибке.

Аналогично, будет ошибкой задавать для ориентированной на ввод процедуры или функции файл, открытый с помощью процедур Rewrite или Append (табл. 19, 20).

19 ПРОЦЕДУРЫ

Наименование	Назначение
Append	Открывает существующий файл для добавления
Flush	Выталкивает буфер файла вывода
Read	Считывает одно или более значений из текстового файла в одну или более переменных
Readln	Выполняет те же действия, что и Read, а потом делает пропуск до начала следующей строки файла
SetTextBuf	Назначает буфер ввода/вывода для текстового файла
Write	Записывает в текстовый файл одно или более значений
Writeln	Выполняет те же действия, что и Write, а затем добавляет к файлу маркер конца строки

20 ФУНКЦИИ

Наименование	Назначение
Eoln	Возвращает для файла состояние end-of-line (конец строки)
SeekEof	Возвращает для файла состояние end-of-file (конец файла)
SeekEoln	Возвращает для файла состояние end-of-line (конец строки)

Read ([имя]; список ввода); – для считывания информации из текстового или с клавиатуры.
Writeln ([имя], список вывода); – для вывода информации в текстовый файл или на дисплей.

Нетипизированные файлы

Нетипизированные файлы представляют собой каналы ввода/вывода нижнего уровня, используемые в основном для прямого доступа к любому файлу на диске, независимо от его типа и структуры.

Любой нетипизированный файл объявляется со словом file без атрибутов, *например:*

```
var
    Datafile : file;
```

Для нетипизированных файлов в процедурах Reset и Rewrite допускается указывать дополнительный параметр, чтобы задать размер записи, использующийся при передаче данных.

По умолчанию длина записи равна 128 байт. Предпочтительной длиной записи является длина записи, равная 1, поскольку это единственное значение, которое точно отражает размер любого файла (если длина записи равна 1, то неполные записи невозможны). 512 байтов – размер сектора физической единицы хранения информации на файловых носителях.

За исключением процедур Read и Write для всех нетипизированных файлов допускается использование любой стандартной процедуры, которую разрешено использовать с типизированными файлами. Вместо процедур Read и Write здесь используются соответственно процедуры BlockRead и BlockWrite, позволяющие пересылать данные с высокой скоростью (табл. 21).

Пример работы:

```
Var
    InFile, OntFile: File;
    Buffer: Array [1, … , 512] of Byte;
  Begin
Assign (InFile, 'TlisFile'); {файл источник данных}
Assign (OutFile, 'TlisFile'); {файл принятия данных}
    Reset (InFile, 512);
    Rewrite (OntFile, 512);
While not EOF (InFile) do{цикл копирования данных из источника в приемник }
  Begin
    BlockRead (InFile, Buffer, 1);
    BlockWrite (OntFile, Buffer, 1);
 End;
    Close (OntFile);
    Erase (InFile); {удаление файла–источника}
 End.
```

21 П Р О Ц Е Д У Р Ы

Наименование	Назначение
BlockRead	Считывает в переменную одну или более записей
BlockWrite	Записывает одну или более записей из переменной

П Е Р Е М Е Н Н А Я F I L E M O D E

Переменная FileMode, определенная в модуле System, устанавливает код доступа, который передается в DOS, когда типизированные и нетипизированные файлы (но не текстовые) открываются с помощью процедуры Reset.

По умолчанию задается значение FileMode равное 2, которое разрешает и чтение, и запись. Присваивание другого значения переменной FileMode приводит к тому, что все последующие вызовы процедуры Reset будут использовать этот режим.

Диапазон возможных значений FileMode зависит от используемой версии DOS. Однако для всех версий задаются следующие режимы :

0 : только чтение
1 : только запись
2 : чтение/запись

Версия 3.X DOS задает дополнительные режимы, которые главным образом связаны с разделенным использованием файлов в сетях. (Для получения более подробной информации, обратитесь к руководству программиста по DOS).

Примечание: Новые файлы, созданные с помощью процедуры Rewrite, всегда открываются в режиме Read/Write, соответствующем значению переменной FileMode = 2.

У С Т Р О Й С Т В А

В операционной системе DOS внешняя аппаратура, как например, клавиатура, устройство печати, дисплей, рассматриваются, как устройства. С точки зрения программиста устройство можно рассматривать как файл и с ним можно работать с помощью того же набора стандартных функций, что и с файлом. В Turbo Pascal поддерживается два типа устройств – устройства DOS и устройства для текстовых файлов.

Устройства DOS

Устройства DOS реализованы с помощью зарезервированных имен устройств, которые имеют специальный смысл. Устройства DOS полностью прозрачны : в Turbo Pascal неизвестно даже, когда файловая переменная связана с устройством, а когда с файлом на диске. *Например*, программа:

```
var
Lst : Text;
begin
Assign (Lst, 'LPT1');
Rewrite (Lst);
```

```
Writeln (Lst, 'Hello World ...');
Close (Lst);
end.
```

выведет строку 'Hello World ...' на устройство печати, хотя синтаксис точно такой же, как если бы она выводилась в файл на диске.

Устройства, реализованные в операционной системе DOS, используются для одновременного ввода или вывода в зависимости от назначения.

УСТРОЙСТВО CON

Устройство CON означает консоль, посредством которой выводимая информация пересылается на экран дисплея, а вводимая информация воспринимается с клавиатуры. Если не было изменено направление ввода или вывода, то стандартные файлы Input и Output и все файлы, которым присвоено пустое имя, ссылаются на устройство CON.

Вводимая с устройства CON информация является строчно – ориентированной и использует средства редактирования строки, которые описаны в руководстве по DOS. Символы считываются из буфера строки, а когда буфер становится пустым, вводится новая строка. При нажатии клавиш CTRL-Z генерируется символ конца файла (end-of-file), после которого функция eof будет возвращать значение True.

УСТРОЙСТВА LPT1, LPT2, LPT3

В качестве возможного построчного устройства печати допускается использование трех устройств печати. Если присоединено одно устройство печати, на него обычно ссылаются как на устройство LPT1. Для этого устройства можно также использовать синоним PRN.

Построчное устройство печати – это устройство, предназначенное только для вывода. При любой попытке использовать процедуру Reset для открытия файла, связанного с одним из этих устройств, немедленно генерируется признак конца файла.

Примечание: Стандартный модуль Printer описывает текстовую файловую переменную с именем Lst и устанавливает ее связь с устройством LPT1. Чтобы обеспечить вывод какой-либо информации из вашей программы на устройство печати, включите в предложение uses в своей программе модуль Printer, а для вывода используйте процедуру

Write (Lst, ...).

УСТРОЙСТВА COM1 И COM2

Устройствами коммуникационного порта являются два последовательных коммуникационных порта. Вместо COM1 можно использовать синоним AUX.

УСТРОЙСТВО NUL

Нулевое устройство игнорирует любую запись на него и немедленно генерирует признак конца файла при попытке считывания с этого устройства. Его следует использовать, если вы не хотите создавать отдельный файл, а в программе требуется указать имя входного или выходного файла.

СТРОКИ

Строки в СИ

Важным подклассом одномерных массивов являются массивы символов, или строки.

Наиболее важным фактом, относящимся к строкам, является то, что константы строки заканчиваются нулевым символом '\0'. Строка-константа (строка-литерал) ограничивается двойными кавычками. В конец каждой строки компилятор подставляет признак конца строки-литерала, символ '\0'.

```
#include <stdio.h>
char message[50]="He said that he was busy";
printf ("\n %s", message);
```

Функции из стандартной библиотеки для работы со строками в СИ

char *strcat (char *dest, char *source) – конкатенируем строки dest и source.

char *strncat (char *dest, char *sours, unsigned maxlen) – присоединяет maxlen символов строки source к строке dest.

char *strchr (char *source, char ch) – поискв строке source первого вхождения символа ch.

int strcmp (char *s1, char s2) – возвращает 0, если s1 = s2, возвращает <0, если s1<s2, и возвращает >0, если s1>s2. Int strcmp (char *s1, chat *s2, int maxlen). Сравниваются только первые maxlen символов двух строк.

char *strcpy (char *dest, char *source) – копирование строки source в строку dest.

char *strncpy (char *dest, char *source, unsigned maxlen) – копирование maxlen символов из строки sourcев строку dest.

int strlen (char *s) – выдает число символов в строке s без нулевого символа конца строки.

char *strlwr (char *s) – переводит всю строку s в нижний регистр (в строчные буквы).

char *strupr (char *s) – в прописные буквы

char *strdup (char *s) – вызывает функцию malloc и отводит место под копию s.

char *strset (char *s, char *ch) – заполняет всю строку s символами ch.

char *strcspn (char *s1, char *s2) – возвращает длину начального сегмента строки s1, которая состоит исключительно из символов, не содержащихся в строке s2.

char *strpbrk (char *s1, char *s2) – просматривает строку s1 до тех пор, пока в ней не встретится символ, содержащийся в s2.

char *strrchr (char *s, char *c) – просматривает строку s до последнего появления в ней символа c.

char *strtok (char *s1, char *s2) – предполагается, что строка s1 состоит из фрагментов, разделенных одно- или многосимвольными разделителями из строки s2. При первом обращении к strtok выдается указатель на первый символ первого фрагмента строки s1. Последующие вызовы с заданием нулю вместо первого фрагмента будут выдавать адреса дальнейших фрагментов из строки s1 до тех пор, пока фрагментов не останется.

Пример использования строковых функций в СИ:

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

int main(void)
{
char *b="tyrbo ", *a="c++", *c, *k="r";
char *d="IRINA";
inti;
 clrscr();
   c=strpbrk(b,"y"); // просматривает строку b до тех пор, пока в ней не встретится //символ'y'.
                printf("1 strpbrk %s\n", c);                    на экране:1 strpbrk yrbo
   strtok(b,a);// предполагается, что строка b состоит из фрагментов, разделенных //одно- или многосимвольными разделителями из строки a.
   printf("2 strtok %s\n", b); на экране: 2 strtok tyrbo
  strlwr(d);// переводит всю строку d в нижний регистр (в строчные буквы)
   printf("3 strlwr %s\n",d);на экране: 3 strlwr irina
strupr(d);// в прописные буквы
   printf("4 strupr %s\n",d);на экране: 4 strupr IRINA
 c=strcat(b,a);// объединяет строки a и b
   printf("5 strcat %s\n", c);на экране: 5 strcat tyrbo c++
c=strchr(b,'y');// поиск в строке b первого вхождения символа y.
   printf("6 strchr %s\n", c); на экране: 6 strchr yrbo c++
       i=strcmp(a,b);// возвращает 0, если a==b, возвращает <0, если a<sb, и //возвращает >0, если a>b.
   printf("7 strcmp %i\n", i); на экране: 7 strcmp -73
  strcpy(b,a);// копирование строки b в строку a.
   printf("%s\n", b);на экране: ++
  getch();
}
```

В стандартной библиотеке (Turbo) СИ stdio.h имеются две функции, являющиеся расширениями функций printf и scanfи поддерживающие ввод/вывод не в файл, а в строки.

Функция sprintf выводит в строку, адрес которой задается первым параметром: int sprintf (char *dest, char *format, …);

Функция sscanf читает из строки, адрес которой задается первым параметром int sscanf (char *dest, char *format, …).

С т р о к и в П а с к а л е

При описании в программе строковых переменных указывают максимально возможное количество символов в строке, например:

Var A:String[10];

Если этот параметр опущен, например:

Var A: String;

то это означает, что максимальное число символов в строке равно 255.

Таким образом строковую переменную можно рассматривать как массив символов, т.е. если мы в переменную A поместим строку 'Паскаль' (A:='Паскаль'), то A[1]='П', A[2]='а', A[3]=='с' и т.д. Пусть максимальное число символов в строке A равно 10, тогда эту переменную можно изобразить в виде схемы:

A

7	П	А	с	К	А	л	ь			
0	1	2	3	4	5	6	7	8	9	10

На самом деле, как видно из этой схемы, переменная A занимает в памяти на один байт больше максимального числа символов в строке, потому что в нулевом элементе данного массива хранится информация о текущем количестве символов в строке. В нашем примере слово *Паскаль* содержит 7 букв (символов), следовательно, A[0]='7' (A[0] не число, а символ '7').

Строковые функциии процедуры в Паскале

Объединение строк. Пусть A, B, C – переменные типа String, A = 'Turbo', a B ='Pascal', тогда в результате выполнения оператора

C:=A+"+B;

мы получим в переменной C строку Turbo Pascal.

Копирование ("вырезка").

Copy (St, Index, Count);

Функция копирует из строки St Count символов, начиная с символа с номером Count. Например, в результате выполнения

C:= Copy (A, 2, 3);

мы получим в переменной C строку 'urb'.

Удаление символов из строки.

Delete (St, Index, Count);

Процедура удаляет Count символов из строки St, начиная с символа с номером Index. Например, в результате выполнения

Delete (B, 4, 3);

мы получим в переменной B строку 'Pas'.

Вставка символов в строку

Insert(subst, st, index);

Процедура вставляет подстроку subst в строку st, начиная с символа с номером index. Например, в результате выполнения

Insert (A, B, 4);

мы получим в переменной B строку 'PasTurbocal'.

Определение длины строки

Length(st);

Функция целого типа (Integer) возвращает длину строки st. Например, пусть k – переменная типа Integer, тогда в результате выполнения

k:= Length(A)

получим в этой переменной число 5 (k = 5).

Поиск подстроки в строке.

Pos (Subst, St);

Функция целого типа (Integer) отыскивает в строке St первое вхождение подстроки Subst и возвращает номер позиции, с которой она начинается; если подстрока не найдена, то возвращается ноль. Например,

k:=Pos('rb',A); даст k=3, а

k:=Pos (A,B); даст k=0.

Преобразование числа в строку

Str(X,St); или

Str (X : Width: Decimals,St);

Процедура преобразует число X (Integer или Real) в строку St. Иногда указывают Width – общее число символов в X и Decimals – число знаков после десятичной запятой. Например, пусть X – переменная типа Real и X = 1.2345, тогда в результате выполнения

Str(X, C); мы получим C = ' 1.2345000000E+00', а в результате

Str(X : 6 : 4, C); мы получим C = '1.2345'.

Преобразование строки в число

Val(St, X, CODE);

Процедура преобразует строку St в целую или вещественную переменную X. Параметр CODE (переменная типа Integer), содержит ноль, если преобразование прошло успешно, или номер ошибочного символа в строке, в противном случае. Например, переменная C (типа String) содержит строку '123.456', тогда

Val (C, X, k); даст X = 123.456, k = 0. Если C = '123,456', то k = 4, а переменная X будет не определена.

Графика

Для построения графических изображения на экране дисплея в библиотеку функций СИ включен набор графических функций, которые могут быть вызваны из программы на языке СИ. Для использования функций графической библиотеки оборудование ПЭВМ должно поддерживать растровую графику.

Растровая графика базируется на понятии пикселя – наименьшей адресуемой точке на экране монитора.

Структура графической программы

Любая графическая СИ-программа организуется по следующей схеме:

- установка видеорежима;
- создание и манипулирование графическими объектами;
- восстановление первоначальной видеоконфигурации перед выходом из программы.

В СИ-программу, использующую графические функции, должен быть включен (директивой #include) файл <graphics.h>. Файл graphics.h объявления структур и символических констант, используемые графическими функциями, а так же прототипы графических функций. В состав графической библиотеки входит около 80 функций. По назначению их можно условно разбить на следующие семь групп:

- инициализация графической системы и графического режима;
- работа с растром пикселей как с двумерной структурой точек;
- управление цветом;
- базовые функции доступа к видеопамяти;
- графические примитивы;
- вывод графических текстовых сообщений;
- обработка ошибочных ситуаций.

Установка видеорежима

Первым шагом любой графической программы является установка видеорежима, обеспечивающего требуемые графические операции. Видеорежим определяет размер экрана в пикселях и количество допустимых цветов.

Графические драйверы, входящие в Turbo СИ, содержатся в шести отдельных файлах, имеющих расширение .bgi. Каждый файл содержит бинарный образ драйвера для одного или нескольких близких по типу адаптеров.

22 Файлы графических драйверов BGI

Номер файла	Имя файла	Размер (байт)	Дисплейный адаптер, обслуживаемый соответствующими драйверами
1	CGA.BGI	6253	CGA, MCGA
2	EGAVGA.BGI	5363	EGA, VGA
3	IBM8514.BGI	6665	IBM8514
4	HERC.BGI	6125	HERCULES
5	ATT.BGI	6269	ATT400
6	PC3270.BGI	6029	3270PC

Кроме констант, указанных в табл. 22, тип graphics_drivers включает еще две: DETECT=0 и CURRENT_DRIVER=-1.

Простейшим для программиста способом включения драйвера в программу является его автоматическая загрузка с помощью функции initgrahp. Эта функция, помимо прочих выполняемых ею действий, ищет на диске BGI-файл, содержащий требуемый драйвер, загружает файл целиком в динамически выделяемую область памяти и настраивает ядро графической системы на работу с этим драйвером.

Недостатком этого способа является обращение к диску для чтения BGI-файла во время выполнения задачи. Если такое условие не смущает программиста, можно ей пользоваться. Для использования графики необходимо включить в графическую библиотеку GRAPHICS.LIB

Полный синтаксис основной функции инициализации библиотеки таков:

Void far initgraph (int far *graphdriver, int far *graphmode,

Char far *pathtodriver);

Аргументами этой функции являются указатели на переменные, содержащие номер графического драйвера, номер графического режима этого драйвера и путь к директории содержащей BGI-файл драйвера.

Если переменной *graphdriver перед вызовом функции было присвоено значение DETECT (или 0), то сначала запускается процедура автоматического тестирования аппаратуры с целью определения типа дисплейного адаптера. Как только процедура тестирования вернет неотрицательное значение (т.е. обнаружит искомый видеоадаптер), дальнейшее тестирование прекращается, и функция initgraph переходит к загрузке соответствующего BGI-файла.

Любая процедура тестирования видеоадаптера, установленная программистом при обнаружении адаптера, возвращает рекомендуемое для драйвера этого адаптера значение графического режима. Потому, если значение *graphdriver было

DETECT, то аргумент graphmode функции graphdriver игнорируется. Если ни одна из процедур тестирования аппаратуры не обнаружила в составе ПК искомый дисплейный адаптер, функция initgraph прекращает работу с кодом –2.

Пример:

```
#include <stdio.h>
#include <graphics.h>
void prim(void)
{
intgdriver=DETECT, gmode, errorcode;
initgraph (&gdriver, &gmode, \\TC\\BGI\\);
errorcode=graphresult();
if (errorcode != grok)
printf ("Graphics error: %s \n", grapherrormsg (errorcode));
printf ("Press any key to halt:");
getch();
exit (1); /*return with error code*/
closegraph();
}
main ()
{
extern void prim (void);
void prim ();
}
```

Работа с растром точек

Растр пикселей – это двумерная совокупность точек, представляющая экран дисплея. В буфере дисплейного адаптера могут помещаться один или несколько таких растров – одинаковых прямоугольных массивов пикселей. Каждый такой массив – есть образ экрана и называется страницей. Количество страниц видеопамяти зависит от типа адаптера и установленного в данный момент графического режима. Размер страницы зависит от графического режима (табл. 23).

Чтобы в прикладной программе иметь возможность отобразить на экране любую из имеющихся страниц видеопамяти, в графической библиотеке предусмотрена функция

void far setvisualpage (int page);

Функция немедленно отобразит на экране ту страницу видеопамяти, номер которой был передан ее в качестве аргумента. Страницы нумеруются, начиная с 0.

На каждой странице видеопамяти существует система координат (X, Y). Начало системы координат лежит в левом верхнем углу страницы (экрана). Ось X проходит по верхнему краю страницы слева направо, ось Y – по левому краю сверху вниз.

23 Графические режимы

Номер режима	Имя режима	Размер страницы, пиксель	Количество цветов	Количество страниц	Режим BIOS
0	CGACO	320 × 200	C0	1	04H
1	EGAHI	640 × 350	16	2	10H
0	EGALO	640 × 200	16	4	0EH
0	VGALO	640 × 200	16	4	0EH
2	VGAHI	640 × 480	16	1	12H

Определить максимальное значение координат точек, допустимое в данном графическом режиме, можно с помощью двух функций: intfargetmaxx(void); intfargetmaxy(void); значения, возвращаемые этими функциями, зависят только от текущего графического режима.

В распоряжении программиста, кроме страницы как целого, имеется графическое окно со всей системой координат. Оно имеет переменные размеры и может размещаться в любом месте страницы.

У программиста есть возможность управлять размерами и расположением графического окна динамически. Делается это с помощью функции.

void setviewport(int left, int top, int right, int bottom, int clip);

Первые четыре аргумента этой функции есть координаты левой, верхней, правой и нижней границ графического окна в системе координат страницы. Ни одна из этих границ не может лежать за пределами границы.

При установке нового графического окна текущая графическая позиция автоматически помещается в его начало координат. В дальнейшем ее координаты могут изменяться с помощью функций void far marto (int x, int y); void far marler (int dx, int dy); первая помещает текущую графическую позицию по указанным координатам графического окна. Вторая перемещает текущую графическую позицию на вектор (dx, dy).

Координаты текущей графической позиции в системе координат графического окна возвращаются функцией

Int far getx(void); int far gety(void);

При переустановке графического окна функцией setviewport содержимое страницы видеопамяти не изменяется. Для очистки графического окна используется функция

void far clearviewport(void);

Текущая графическая позиция перемещается в начало координат графического окна.

Функция void far cleardevice (void) очищает всю активную страницу.

Управление цветом

Установка текущего цвета для любого графического видеорежима

setcolor(color);

long gettextcolor(void); текущий цвет текста;

settext color; устанавливает цвет текста;

getbkcolor(void); цвет фона;

setbkcolor(bkcolor); устанавливает цвет фона;

long bkcolor; цвет фона;

short color; цвет текста;

Функции settextcolor и setbkcolor возвращают значение предыдущего цвета текста и фона.

Все значения цвета от 16 до 31 – того же цвета, что и от 0 до 15, но только с мерцанием.

Графические примитивы

К группе контурных графических примитивов относятся следующие функции.

```
void line (int x1, int y1, int x2, int y2);
void linerel (int dx, int dy);
void lineto (int x, int y);
void rectangle (int left, int top, int right, int bottom);
void drawpoly (int numpoints, int *polypoints);
void circle (int x, int y, int radius);
void arc (int x, int y, int stangle, int endangle, int radius);
void ellipse (int x, int y, int stangle, int endangle, int xradius, int yradius);
```

Функции line, linerel и lineto соединяют две точки плоскости отрезком прямой. Для первой из них обе соединяемые точки указываются явно своими координатами. Функции linerel и lineto в качестве первой точки используют текущую графическую позицию, а вторую выбирают аналогично функциям moverel и moveto, т.е. через приращение координат или по явному указанию; при этом текущая графическая позиция перемещается во вторую точку. Все три функции пользуются системой координат текущего графического окна.

Функция rectangle рисует на странице видеопамяти контур прямоугольника в соответствии с указанным левым верхним и правым нижним углами.

Функция drawpoly рисует ломаную линию, соединяя отрезками прямых последовательность точек на плоскости. В первом аргументе ей передается количество этих точек, а второй указывает на массив целых чисел. Каждая пара чисел из этого массива интерпретируется как пара координат (x, y) очередной точки.

Функции arc и ellipse вычерчивает дуги окружности и эллипса, соответственно ограниченных углами stangle и endangle. Для дуги задается ее центр. Дуга эллипса определяется углами, центром и радиусом двух осей.

Пример построения графиков

```
/*****************************************************************/
                    /* Данная программа строит график функции по точкам. Координаты точек*/
    /*считываются из файла coords.txt*/
/*****************************************************************/

// Подключаемые библиотеки
#include<stdio.h> // Библиотека ввода-вывода
#include<conio.h> // Библиотека работы с окнами
#include<graphics.h> // Библиотека графических функций
#include<math.h> // Библиотека математических функций
#include<alloc.h> // Библиотека работы с динамической памятью
```

```
#include<stdlib.h> // Системная библиотека
#include<string.h> // Библиотека работы со строками

// Прототипы функций
void avtor(void); // Функция, выводящая на экран информацию
                  // об авторе данной программы
void fileinfo(void); // Функция, вычисления числа точек графика,
                   // находящихся в файле
void readfile(void); // Функция чтения значений из файла в массивы
void getmemory(void); // Функция распределения массивов в динамической памяти
void freememory(void); // Функция освобождения динамической памяти
void tablica(void); // Функция, выводящая на экран монитора таблицу с
                  // координатами точек графика
void init(void); // Функция инициализации графического режима с автоматическим
                   // определением типа графического адаптера и проверкой
                   // возможности инициализации графического режима
void convert(float ch); // Функция преобразования переменной типа float в
                        // переменную типа char
void masshtab(void); // Функция рассчета масштаба по осям
void systemcoord(void); // Функция построения системы координат
void setka(void); // Функция построения координатной сетки
void grafic(void); // Функция построения графика функции
// Раздел объявления глобальных переменных
float *x, *y; // Указатели на одномерные массивы значений абсцисс
     // и ординат точек графика
int n; // Число точек графика, прочитанных из файла
int i, j, k; // Переменные – счетчики цикла
char *s; // Указатель на строку, полученную при преобразовании
      // переменной типа floatв переменную типа char
float xmax,ymax,xmin,ymin; // Максимальные и минимальные значения
                           // координат точек графика
int mx,my; // Масштаб по осям
int x0, y0; // Абсциссаи ордината точки пересечения координатных
     // осей на экране монитора
int maxx, maxy; // Максимальные координаты осей для
                  // заданного графического режима

// Функция, выводящая на экран информацию об авторе данной программы
void avtor(void)
{
textbackground(1); // Установка синего цвета фона в текстовом режиме
textcolor(14); // Установка желтого цвета символов в текстовом режиме
clrscr(); // Очистка экрана в текстовом режиме
gotoxy(22,8); // Смещение курсора в точку экрана с координатами (22,8)
printf("ПРОГРАММА ПОСТРОЕНИЯ ГРАФИКОВ ФУНКЦИЙ");
gotoxy(20,15); // Смещение курсора в точку экрана с координатами (20,15)
printf("Составитель: Грибков Алексей Николаевич");
gotoxy(36,16); // Смещение курсора в точку экрана с координатами (36,16)
printf("Факультет КТФ, группа Р-11");
gotoxy(35,21); // Смещение курсора в точку экрана с координатами (23,24)
printf("1 9 9 9г.");
gotoxy(23,24); // Смещение курсора в точку экрана с координатами (23,24)
printf("Нажмите любую клавишу для продолжения");
getch(); // Ожидание нажатия любой клавиши
}

// Функция распределения массивов в динамической памяти (кучи)
void getmemory(void)
{
// Распределение в динамической памяти массива x
for(i=0;i<n;i++) x=(float*)calloc(i,sizeof(float));
// Распределение в динамической памяти массива y
```

```
for(i=0;i<n;i++) y=(float*)calloc(i,sizeof(float));
}

// Функция освобождения динамической памяти
void freememory(void)
{
free(x); // Удаление массива x из динамической памяти
free(y); // Удаление массива y из динамической памяти
}

// Функция инициализации графического режима с автоматическим определением
// типа графического адаптера и проверкой возможности инициализации
// графического режима
void init(void)
{
// Раздел объявления локальных переменных
int graphdriver, graphmode; // Переменные для инициализации
                            // графического режима
int errorcode; // Код ошибки инициализации
graphdriver=DETECT; // Автоматическое определение
graphmode=0;// типа графического адаптера
// Драйвер egavga.bgi находится в каталоге bgi
initgraph(&graphdriver, &graphmode, "..\\bgi");
maxx=getmaxx()+1; // Нахождение максимальных координат
maxy=getmaxy()+1; // осей, для заданного графического режима
errorcode=graphresult(); // Чтение результата инициализации
if(errorcode!=grOk) // Если графика не инициализируется, то на экран
  {                 // выводится сообщение об ошибке
   clrscr(); // Очистка экрана в текстовом режиме
   // Вывод сообщения об ошибке на экран монитора
   printf("Ошибка инициализации графики: %s\n", grapherrormsg(errorcode));
   printf("Нажмите любую клавишу");
   getch(); // Задержка экрана до нажатия любой клавиши
   exit(1); // Выход из программы
  }
}
// Функция, вычисления числа точек графика, находящихся в файле
void fileinfo(void)
{
// Раздел объявления локальных переменных
static char buf[10]; // Статическая переменная для хранения значений,
// считанных из файла, в данном случае используется не как переменная
// для хранения значений, а как вспомогательная переменная, позволяющая
// подсчитать число раз обращений к файлу
FILE *f; // Файловая переменная
  f=fopen("coords.txt","rt"); // Открытие файла coords.txt на чтение
i=0; // Присвоение начального значения переменной счетчика цикла
while(!feof(f)) // Цикл продолжается до тех пор,
{ // пока не наступит конец файла
  fscanf(f,"%s",&buf); // Чтение значений из файла
  i++; // Увеличение счетчика цикла
}
// Фактически, число точек графика, содержащееся в файле в два раза
// меньше числа раз обращений к файлу, так как каждая точка графика
// определяется двумя координатами, а за одно обращение к файлу
// считывается только одна координата точки
n=i/2; // Вычисление числа точек гафика
fclose(f); // Закрытие файла coords.txt
}

// Функция чтения значений из файла в массивы
void readfile(void)
```

```
  {
  // Раздел объявления локальных переменных
  static char buf[10]; // Статическая переменная для хранения значений,
                       // считанных из файла
  FILE *f; // Файловая переменная
  // Открытие файла coords.txt на чтение
  f=fopen("coords.txt","rt");
  i=0; j=0; k=0; // Присвоение начального значения
                 // переменным счетчика цикла
  while(!feof(f)) // Цикл продолжается до тех пор,
  {               // пока не наступит конец файла
  fscanf(f,"%s",&buf); // Считываем значения из файла в переменную buf
  // Если из файла считывается элемент с четным порядковым
  // номером, то присваиваем его значение соответствующему элементу
  // массива координат x, если же из файла считывается элемент с
  // нечетным порядковым номером, то присваиваем его значение
  if(k%2==0) { x[i]=atof(buf); i++; } // соответствующему элементу
  if(k%2!=0) { y[j]=atof(buf); j++; } // массива координат y
  k++; // Увеличение счетчика цикла
  }
  fclose(f); // Закрытие файла coords.txt
}

  // Функция преобразования из числа в строку
  // Передаваемый параметр – преобразуемое число
  void convert(float ch)
  {
  // Раздел объявления локальных переменных
  long int c,d; // Переменные, для хранения значений первой и второй
                // цифр, стоящих после запятой в преобразуемом числе
  char *sm;    // Вспомогательная символьная переменная
  strcpy(sm,","); // Присвоение значения символьной переменной
  ltoa(ch,s,10); // Преобразование переменной типа long int в
                 // переменную типа char (помещаем в строку
                 // целую часть преобразуемого числа)
  if(ch<0) ch=-ch; // Если преобразуемое число отрицательное, то оно
                 // меняется на свое положительное значение
  d=(long int)ch; // Переменной d присваивается значение целой части
                // преобразуемого числа
  if(ch!=d) // Если преобразуемое число имеет дробную часть, то
    { // его преобразование продолжается дальше
   strcat(s,sm); // После целой части числа, в строку записывается
                // разделительная запятая
  c=(long int)((ch-d)*10); // Вычисляется первая цифра
                           // числа, стоящая после запятой
  ltoa(c,sm,10); // Эта цифра записывается во вспомогательную
                 // символьную переменную
  strcat(s,sm); // Первая цифра преобразуемого числа, стоящая
                // после запятой записывается в определенную
                // позицию строки
  c=(long int)((ch-d)*100-c*10); // Вычисляется вторая цифра
                                 // числа, после запятой
  ltoa(c,sm,10); // Эта цифра записывается во вспомогательную
                 // символьную переменную
  strcat(s,sm); // Вторая цифра преобразуемого числа, после запятой
                // записывается в определенную позицию строки
  }
}

// Функция, выводящая на экран монитора таблицу с
// координатами точек графика
void tablica(void)
```

```
{
  textbackground(0); // Установка черного цвета фона в текстовом режиме
  textcolor(15); // Установка белого цвета символов в текстовом режиме
  clrscr(); // Очистка экрана в текстовом режиме
  printf("\t\t\t\tТаблица точек графика");
  // Вывод на экран верхней части таблицы
  gotoxy(14,3);
  printf("--------------T------------------T------------------¬");
  gotoxy(14,4);
  printf("¦ ¦¦¦");
  // Вывод названий столбцов таблицы
  gotoxy(16,wherey());
  printf("Номер точки");
  gotoxy(31,wherey());
  printf("Абсцисса точки");
  gotoxy(50,wherey());
  printf("Ордината точки");
  gotoxy(1,wherey()+1);
  j=5;
  for(i=0;i<n;i++)
       {
       // Печать таблицы
       gotoxy(14,j);
       printf("+--------------+------------------+------------------+");
       gotoxy(14,j+1);
       printf("¦ ¦¦¦");
       // Заполнение таблицы
       gotoxy(21,j+1);
       printf("%i",i);
       gotoxy(33,j+1);
       printf("%4.2f",x[i]);
       gotoxy(52,j+1);
       printf("%4.2f",y[i]);
       j=j+2; // Увеличение счетчика цикла
       }
  // Вывод на экран окончания таблицы
  gotoxy(14,j);
  printf("L--------------+------------------+------------------");
  gotoxy(23,24); // Смещение курсора в точку экрана с координатами (23,24)
  printf("Нажмите любую клавишу для продолжения");
  getch(); // Задержка изображения экрана до нажатия любой клавиши
}

  // Функция рассчета масштаба по осям
  void masshtab(void)
  {
    // Нахождение максимальной координаты точки графика по оси ox
    j=0; // Присвоение начального значения переменной счетчику цикла
    xmax=x[j]; // Присвоение переменной xmax значения нулевого
           // элемента массива
  while(j<n-1)
  {
       // Если очередное значение элемента массива больше значения
       // переменной xmax, то присваиваем переменной xmax значение
       // этого элемента массива
       if(x[j+1]>xmax) xmax=x[j+1];
       j++; // Увеличение счетчика цикла
       }
  // Нахождение максимальной координаты точки графика по оси oy
  j=0; // Присвоение начального значения переменной счетчику цикла
  ymax=y[j]; // Присвоение переменной ymax значения нулевого
                 // элемента массива
```

```
while(j<n-1)
{
// Если очередное значение элемента массива больше значения
// переменной ymax, то присваиваем переменной ymax значение
// этого элемента массива
if(y[j+1]>ymax) ymax=y[j+1];
j++; // Увеличение счетчика цикла
}
// Нахождение минимальной координаты точки графика по оси ox
j=0; // Присвоение начального значения переменной счетчику цикла
xmin=x[j]; // Присвоение переменной xmin значения нулевого
      // элемента массива
while(j<n-1)
{
// Если очередное значение элемента массива больше значения
// переменной xmin, то присваиваем переменной xmin значение
// этого элемента массива
if(x[j+1]<xmin) xmin=x[j+1];
j++; // Увеличение счетчика цикла
}
// Нахождение минимальной координаты точки графика по оси oy
j=0; // Присвоение начального значения переменной счетчику цикла
ymin=y[j]; // Присвоение переменной ymin значения нулевого
      // элемента массива
while(j<n-1)
{
// Если очередное значение элемента массива больше значения
// переменной ymin, то присваиваем переменной ymin значение
// этого элемента массива
if(y[j+1]<ymin) ymin=y[j+1];
j++; // Увеличение счетчика цикла
}
// В случае если график не пересекает соответствующую ось,
// максимальные и минимальные значения координат берутся
// непосредственно на координатных осях
if(xmax<=0) xmax=0; // Если график лежит левее оси абсцисс
if(xmin>=0) xmin=0; // Если график лежит правее оси абсцисс
if(ymax<=0) ymax=0; // Если график лежит ниже оси ординат
if(ymin>=0) ymin=0; // Если график лежит выше оси ординат
// Нахождение масштаба по оси абсцисс
mx=(int)((maxx-40)/(xmax+fabs(xmin)));
// Нахождение масштаба по оси ординат
my=(int)((maxy-40)/(fabs(ymax-ymin)));
// Вычисление абсциссы точки расположения оси oy на экране монитора
x0=(int)(fabs(xmin)*mx+20);
// Нахождение ординаты точки расположения оси ox на экране монитора
y0=maxy-(int)(fabs(ymin)*my+20);
}

// Функция построения координатной сетки
void setka(void)
{
cleardevice();          // Очистка экрана в графическом режиме
setbkcolor(0);        // Установка черного цвета фона
setcolor(7);           // Установка цвета символов
setlinestyle(1,0,0); // Установка типа линии
for(i=0;i<n;i++)
{
 line(20,y0-y[i]*my,maxx-20,y0-y[i]*my); // Построение
 line(x0+x[i]*mx,20,x0+x[i]*mx,maxy-20); // координатной сетки
}
}
```

```
// Функция построения системы координат
void systemcoord(void)
{
setcolor(2); // Установка зеленого цвета символов (осей координат)
setlinestyle(0,0,3); // Установка типа линии
settextstyle(1,0,1); // Установка шаблона выводимого на экран текста
line(x0,20,x0,maxy-20); // Построение оси ординат
line(x0,20,x0-5,30); // Задание направления оси ординат
line(x0,20,x0+5,30);
setcolor(15); // Установка белого цвета символов
outtextxy(x0-20,0,"y"); // Вывод названия оси
setcolor(2); // Установка зеленого цвета символов
line(20,y0,maxx-20,y0); // Построение оси абсцисс
line(maxx-20,y0,maxx-40,y0-5); // Задание направления
line(maxx-20,y0,maxx-40,y0+5); // оси абсцисс
setcolor(15); // Установка белого цвета символов
outtextxy(maxx-25,y0+5,"X"); // Вывод названия оси
settextstyle(0,0,0); // Установка шаблона выводимого на экран текста
// Нанесение значений координат на ось oy
for(i=0;i<n;i++)
   {
  if(y[i]!=0) // Если значение ординаты не равно нулю, то наносим
               // ее значение на ось, если значение ординаты равно нулю,
               // то не имеет смысла выводить ее, так как она
{            // будет выводится как абсцисса точки по оси оси ox
     setcolor(12); // Установка светло-красного цвета символов
     circle(x0,y0-y[i]*my,2); // Рисование окружности, для обозначения
      // точки на координатной оси
     setcolor(11); // Установка светло-голубого цвета символов
     convert(y[i]); // Преобразование числа из типа float в тип char
     outtextxy(x0-20,y0-y[i]*my,s); // Вывод значения ординаты точки
     }
}
// Нанесение значений координат на ось ox
for(i=0;i<n;i++)
  {
   setcolor(12); // Установка светло-красного цвета символов
   circle(x0+x[i]*mx,y0,2); // Рисование окружности, для обозначения
                            // точки на координатной оси
   setcolor(11); // Установка светло-голубого цвета символов
   convert(x[i]); // Преобразование числа из типа float в тип char
   outtextxy(x0+x[i]*mx-10,y0+10,s); // Вывод значения абсциссы точки
   }
}

void grafic(void)
{
setcolor(14); // Установка белого цвета символов
setlinestyle(0,0,3); // Установка шаблона линии
for(i=0;i<n-1;i++)
   {
   // Рисование графика
   line(x[i]*mx+x0,y0-y[i]*my,x[i+1]*mx+x0,y0-y[i+1]*my);
   }
getch();// Задержка изображния экрана до нажатия любой клавиши
closegraph(); // Закрытие графического режима
exit(1); // Выход из программы
}
void main(void) // Главная функция (основной блок программы)
{
fileinfo(); // Вычисление количества точек графика, записанных в файле
```

```
  getmemory(); // Распределение массивов, содержащих значения координат
       // точек графика, в динамической памяти
  readfile(); // Чтение координат точек графика из файла
  avtor(); // Вывод информации об авторе данной программы
  tablica(); // Вывод таблицы с координатами точек графика
  init(); // Инициализация графического режима
  masshtab(); // Определение масштаба по осям
  setka(); // Построение координатной сетки
  systemcoord(); // Построение системы координат
  grafic(); // Построение графика функции
  freememory(); // Освобождение динамической памяти, занимаемой массивами
}
```

Пример построения графика на Паскале:

```
          {---------------------------------------------------------------------}
          {Данная программа позволяет рассчитать значения функции в заданном}
          {числе точек, на заданном интервале и построить график функции на}
{этом интервале с автоматическим масштабированием во весь экран.}
{---------------------------------------------------------------------}
{* Внимание! При наличии вертикальных асимптот функции на *}
{* рассматриваемом интервале программа не будет работать.*}
{---------------------------------------------------------------------}
PROGRAM FUNCTION_TAB_AND_GRAPH;
uses GRAPH, CRT; {Подключение библиотек функций и процедур работы с}
          {графикой и ввода-вывода}
{Объявление переменных}
var gd, gm, x, y, xmax, xmin, x0, y0, mx, my : Integer;
   a, b, ymax, ymin, dx, d, c, h : Real;
   d1 : String[5];
   s : Char;
 {Задается функция для построения таблицы и графика}
Function f(x:Real):Real;
 begin
   f := Sin(x*x)+Cos(x);
end;{Function f}
 {В данной процедуре производится построение графика функции}
Procedure GraphFunc(a,b:real);
 begin
gd := Detect;              {Инициализация графического режима}
gm := 0;                   { с автоматическим определением }
InitGraph(gd,gm,'d:\bp\bgi'); {типа графического адаптера}
{Находится масштаб по оси абсцисс mx}
if Frac(a)<0 then xmin := Trunc(a)-1
    else xmin := Trunc(a);
if Frac(b)>0 then xmax := Trunc(b)+1
    else xmax := Trunc(b);
mx := Trunc(600/(xmax-xmin));
ymax := f(a);
ymin := f(a);
d := a;
dx := (b-a)/600; {Шаг изменения аргумента, соответствующий одному}
repeat {пикселю изображения на экране}
    d := d + dx;
    if f(d) > ymax then ymax := f(d); {Поиск максимального и минимального}
    if f(d) < ymin then ymin := f(d); {значения функции}
until d >= b;
{Находится масштаб по оси ординат my}
if Frac(ymin)<0 then ymin := Trunc(ymin)-1
    else ymin := Trunc(ymin);
if Frac(ymax)>0 then ymax := Trunc(ymax)+1
```

```
    else ymax := Trunc(ymax);
my := Trunc(440/(ymax-ymin));
if ymax*ymin <= 0 then
  begin { Находится машинная ордината оси абсцисс y0}
    y0 := 460+Trunc(ymin)*my;
    Line(10,y0,620,y0); {Строится ось абсцисс и задается ее направление}
    Line(615,y0+5,620,y0);
    Line(615,y0-5,620,y0);
  end;
if (ymax*ymin>0) and (ymax>0)
  then y0 := 460+Trunc(ymin)*my;
if (ymax*ymin>0) and (ymax<0)
  then y0 := 20-Round(ymax*my);
if xmax*xmin <= 0 then
  begin{ Находится машинная абсцисса оси ординат }
    x0 := 20-xmin*mx;
    Line (x0,20,x0,460);{ Если график пересекает ось ординат, }
    Line (x0-5,25,x0,20); { она строится и задается ее направление }
    Line (x0+5,25,x0,20);
end;
SetLineStyle(CenterLn,0,NormWidth); {Устанавливается стиль линии}
x := 20; {Находится начальная машинная абсцисса для построения координатной сетки}
d := xmin; {Устанавливается начальное значение аргумента,}
{выводимого по оси абсцисс}
If (y0>20) and (y0<460) then gd := y0 else gd := 460;
repeat
    SetColor(2); {Устанавливается цвет символов}
    if d=0 then SetColor(15); {В случае совпадения линии масштаба с осью}
    {абсцисс устанавливаем цвет координатной линии совпадающим с осью}
    Line(x,20,x,460); {Строится масштаб по оси абсцисс}
    Str(d:3:0,d1); {Производит преобразование переменных из типа Real в Char}
    SetColor(15);
    OutTextXY(x-5,gd+5,d1); {Выводятся значения аргумента по оси абсцисс}
    d := d+1;
    x := x+mx;
until x > 630;
{Устанавливается начальное значение функции, выводимой по оси ординат}
d := ymin;
y := 460;  { Находится начальная машинная }
{ ордината для построения координатной сетки }
If (x0>20) and (x0<460) then gd:=x0 else gd := 20;
repeat
    SetColor(2);
    if d=0 then SetColor(15);
    Line(10,y,620,y); { Строится масштаб по оси ординат }
    Str(d:3:0,d1);
    SetColor(15);
    OutTextXY(gd-5,y-10,d1);
    y := y-my;
    d := d+1;
until y<15;
{Поиск начальной машинной абсциссы для построения графика}
if (a<0) and (Frac(a)<>0) then x := Round((1+Frac(a))*mx)+20;
if Frac(a)=0 then x := 20;
if (a>=0) and (Frac(a)<>0) then x := Round(Frac(a)*mx)+20;
d := dx;
MoveTo(Trunc(mx*dx)+x,y0-(Trunc(f(a)*my))); {Позиционирование курсора}
SetLineStyle(SolidLn,0,NormWidth);
repeat {Построение графика функции на заданном интервале}
    LineTo(Trunc(mx*dx)+x,y0-Trunc(f(a)*my));
```

```
      dx := dx+d;
      a := a+d;
  until a>b;
  OutTextXY(20, 470, 'Для выхода из программы нажмите любую клавишу.');
  ReadKey; { Задержка экрана }
  CloseGraph; { Закрытие графического режима }
end;{Procedure GraphFunc}

begin {Основной блок программы}
  ClrScr;   {Очистка экрана в текстовом режиме}
  WriteLn('Составитель Поляков Д.Г., КТФ, гр. Р-11');
  WriteLn('Введите границы интервала');
  Read(a,b); {Вводятся границы интервала}
  if b<a then {В случае, если начальное значение аргумента}
     begin   {превосходит конечное, то меняем их местами}
       d := b; b := a; a := d;
  end;
  WriteLn('Введите количество разбиений');
  Read(d); {Вводится число промежутков разбиения интервала}
  h:=(b-a)/d; {Рассчитывается шаг изменения аргумента для заполнения таблицы}
  WriteLn('Шаг равен:',h:3:3);
  {Выводится таблица значений функции, соответствующих значениям аргумента,}
  {изменяющегося с вычисленным выше шагом, и номер точки разбиения}
  WriteLn('-------------T-----------------T----------------¬ ');
  WriteLn('¦     ¦         ¦        ¦ ');
      GotoXY(2,WhereY-1); {Производится автоматическая адресация курсора}
      Write('Номер точки'); {в начало каждой графы таблицы для печати текста}
      GotoXY(16,WhereY);
      Write('Знач. аргум.');
      GotoXY(34,WhereY);
      Write('Знач. функции');
      GotoXY(1,WhereY+1);
      y:= WhereY+1;
      c:= 1;
      d:= a;
  repeat
       WriteLn('+-------------+-----------------+----------------+ ');
       Write('¦  ¦     ¦    ¦ ');
       GotoXY(4,y); {Производится автоматическая адресация курсора}
       Write(c:3:0);   { в каждую графу таблицы для печати соответствующего }
       GotoXY(16,y);   { численного значения }
       Write(d:3:4);
       GotoXY(34,y);
       Write(f(d):3:5);
       GotoXY(1,y+1);
       d := d+h;
       y := y+2;
       c := c+1;
   until d >= b+h;
    WriteLn('L-------------+-----------------+---------------- ');
    { Выводится запрос на построение графика функции }
    WriteLn('Строить ли график функции на заданном промежутке? (Y/N)');
    s := ReadKey;  {Считывается вводимый с клавиатуры символ}
    if s in ['y','Y'] then
     begin
       if a=b then  {Проверка возможности построения графика}
         begin
    WriteLn (' Построение графика невозможно ');
```

```
        WriteLn (' Нажмите любую клавишу ');
        ReadKey;  {Выход из программы при}
        Halt(1);  {нажатии любой клавиши}
      end
    else GraphFunc(a,b); {Вызывается процедура построения графика функции
}
      end;
      end.
```

Указатели и распределение памяти в СИ и Паскале

УКАЗАТЕЛИ В Си

Все рассмотренные типы данных содержат непосредственно данные. *Указатель* содержит другой тип информации – адрес. *Указатель* – это переменная, содержащая адрес памяти, где какие либо данные помещаются. Другими словами, он указывает на данные, как адресная книга или оглавление.

Символ операции * используется для задания указателя на объект. Например, рассмотрим следующее описание:

```
int *x;
```

его следует понимать, как "x является указателем на целое". Указатель на тип void совместим с любым указателем. Поэтому можно производить следующее присваивание, *например*:

```
void *x;
int *y;
y=x;
```

В общем случае описывается так: *тип * переменная_указатель;* такая запись означает, что переменная_указатель *является указателем на тип.*

Двумя наиболее важными операциями, связанными с указателем, являются *операции обращения по адресу* * (иногда называется операцией снятия ссылки) и *операция определения адреса &.*

Операция обращения по адресу * служит для присваивания или считывания значения переменной, размещенной по адресу переменная_указатель, при помощи лево-определенного выражения * переменная_указатель. Например, *ptr_var=value;

Операция определения адреса & возвращает адрес памяти своего операнда. Операндом должна быть переменная. Выполняется следующим образом:

```
адрес=& переменная;
```

где адрес – это соответствующее лево-определенное выражение, куда помещается адрес, а переменная – имя переменной, определенной выше в программе. Размер возвращаемого адреса зависит от модели памяти.

Присвоение *x=16 означает, что по адресу, задаваемому в x, помещается значение 16. Здесь операция * используется как обращение по адресу (снятие ссылки).

При работе с указателями их необходимо размещать в динамической памяти (куче). Это делается при помощи функций библиотеки <alloc.h> *например:*

```
#include<alloc.h>
main()
{int *x;
```

```
   x=(int *) malloc (sizeof(int)); // размещение в памяти
   free(x); //освобождение памяти
}
```

УКАЗАТЕЛИ В ПАСКАЛЕ

Обычно указатель указывает на данные определенного типа. Рассмотрим следующий пример:

```
type
  Buffer = string[255];
  BufPtr = ^Buffer;
var
  Buf1: Buffer;
  Buf2: BufPtr;
```

Тип данных Buffer определен, как строковая переменная размером 255 байт, в то же время BufPtr – указатель на Buffer. Переменная Buf1 имеет тип Buffer и занимает (255+1) байт памяти. Переменная Buf2 имеет тип BufPtr, это 32-битовый адрес и занимает 4 байта памяти.

Куда указывает Buf2? В данный момент никуда. Прежде чем использовать BufPtr, необходимо зарезервировать (распределить) память и запомнить ее адрес в Buf2, используя процедуру New:

New(Buf2);

Поскольку Buf2 указывает на тип Buffer, то этот оператор выделит в памяти 256-байтовый буфер и его адрес поместит в Buf2.

Как использовать Buf2 и данные, на которые он указывает? С помощью оператора ^. Например, необходимо поместить строку и в Buf1 и в буфер, на который указывает Buf2:

Buf1: = 'Эта строка помещается в Buf1';

Buf2^: = 'Эта строка помещается по указателю Buf2';

Отметим различие между Buf2 и Buf2^: Buf2 означает 4-байтовую переменную указателя; Buf2^ – 256-байтовую строковую переменную, адрес которой в Buf2.

Освободить память, на которую указывает Buf2 можно, используя процедуру Dispose. Эта процедура освобождает память.

После того, как при помощи Dispose освобождена память, следует присвоить указателю значение nil, т.е. указатель ни на что не указывает:

```
Dispose(Buf2);
Buf2 := nil;
```

Заметим, что значение nil присваивается указателю Buf2, а не Buf2^.

АДРЕСНЫЕ ОПЕРАТОРЫ

В Паскале поддерживаются два специальных оператора над адресами: вычисление адреса (@) и оператор косвенной ссылки (^).

Оператор @ возвращает адрес заданной переменной; если Sum переменная целого типа, то @Sum – адрес в памяти этой переменной.

Аналогично, если ChrPtr – это указатель на тип Char, то ChrPtr^ – это символ, на который указывает ChrPtr^.

Карта памяти в Паскале

Карта памяти в Паскале имеет следующий вид (рис. 12)

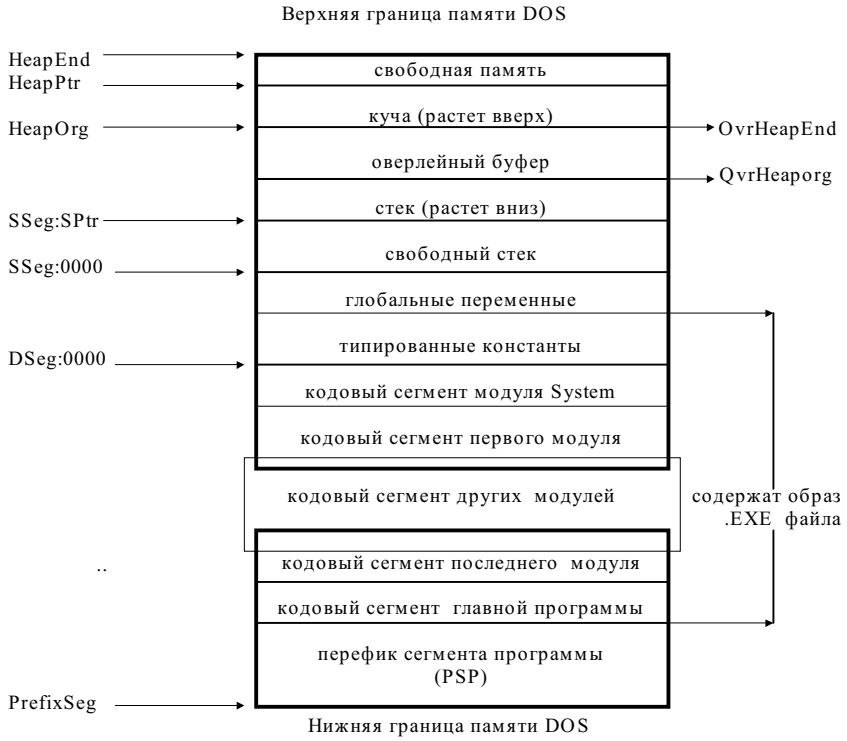


Рис. 12 Распределение памяти для программы на Паскаль

Префикс сегмента программы (Program Segment Prefix – PSP) – это 256-ти байтовая область, создаваемая DOS при загрузке программы. Адрес сегмента PSP хранится в переменной PrefixSeg. Каждый модуль (и главная программа и каждый модуль) имеет свой кодовый сегмент. Главная программа занимает первый кодовый сегмент; кодовые сегменты, которые следуют за ним, занимают модули (в

порядке, обратном тому, как они следовали в операторе uses), и последний кодовый сегмент занимает библиотека времени выполнения (модуль System). Размер одного кодового сегмента не может превышать 64 К, но общий размер кода ограничен только имеющейся памятью.

Сегмент данных

Максимальный размер сегмента данных равен 65 520 байт. При компоновке программы (что автоматически осуществляется в конце компиляции программы) глобальные переменные всех модулей, используемых программой, а также собственные глобальные переменные программы размещаются в сегменте данных.

Если для глобальных переменных требуется более 65 520 байт, то следует распределить большие структуры в виде динамических переменных.

Сегмент стека

Размер сегмента стека устанавливается с помощью директивы компилятора {$M} и лежит в пределах от 1024 до 65 520 байт. По умолчанию размер стека равен 16 384 байт.

При каждой активизации (вызове) процедуры или функции в стек помещается набор локальных переменных. При завершении работы память, занимаемая локальными переменными, освобождается. В любой момент выполнения программы общий размер локальных переменных в активных процедурах и функциях не должен превышать размера сегмента стека.

Директива компилятора {$S} используется для проведения проверок переполнения стека в программе. В состоянии {$S+}, принятом по умолчанию, генерируется код, осуществляющий проверку переполнения стека в начале каждой процедуры или функции. В состоянии {$S-} такие проверки не проводятся. Переполнение стека может вызвать аварийное завершение работы системы, поэтому не следует отменять проверки стека, если нет абсолютной уверенности в том, что переполнения не произойдет.

Сегмент данных (адресуемый через DS) содержит все глобальные переменные, и затем все типизированные константы. Регистр DS никогда не изменяется во время выполнения программы. Размер сегмента данных не может превышать 64 К.

При запуске программы регистр сегмента стека (SS) и указатель стека (SP) устанавливаются так, что SS:SP указывает на первый байт после сегмента стека. Регистр SS никогда не изменяется во время выполнения программы, а SP может передвигаться вниз, пока не достигнет конца сегмента. Размер стекового сегмента не может превышать 64 К; размер по умолчанию – 16 К, он может быть изменен директивой компилятора {$M}.

Буфер оверлеев используется стандартным модулем Overlay для хранения оверлейного кода. Размер оверлейного буфера по умолчанию соответствует размеру наибольшего оверлея в программе; если в программе нет оверлеев, размер буфера оверлеев равен 0. Размер буфера оверлеев может быть увеличен с помощью вызова программы OvrSetBuf модуля Overlay; в этом случае размер кучи соответственно уменьшается, смещением вверх HeapOrg.

Куча хранит динамические переменные, т.е. переменные, распределенные через вызов стандартных процедур New и GetMem. Куча занимает всю или часть свободной памяти, оставшейся после загрузки программы. Фактически размер кучи зависит от минимального и максимального значений кучи, которые могут быть установлены директивой компилятора $M. Размер кучи никогда не будет меньше минимального значения и не превысит максимального. Если в системе нет памяти равного минимальному значению, программа не будет выполняться. Минимальное значение кучи по умолчанию равно 0 байт, максимальное – 640 К; это означает, что по умолчанию куча будет занимать всю оставшуюся память.

Управление кучей осуществляет монитор кучи (который является частью библиотеки времени выполнения Turbo Pascal).

МОНИТОР КУЧИ

Куча имеет стековую структуру, растущую от нижних адресов памяти в сегменте кучи. Нижняя граница кучи хранится в переменной HeapOrg, а вершина кучи, соответствующая нижней границе свободной памяти, хранится в переменной HeapPtr. Каждый раз, когда динамическая переменная распределяется в куче (через New или GetMem), монитор кучи передвигает HeapPtr вверх на размер этой переменной, ставя динамические переменные одну за другой.

HeapPtr нормализуется после каждой операции, устанавливая смещение в диапазоне от $0000 до $000F. Максимальный размер переменной, который может быть распределен в куче, равен 65 519 байт ($10 000 – $000F), поскольку каждая переменная должна полностью находиться в одном сегменте.

Освобождение памяти

Динамические переменные, хранящиеся в куче, удаляются одним из двух путей: (1) через Dispose или FreeMem (2) через Mark и Release. Простейший способ – это с Mark и Release, если были выполнены следующие операторы:

```
New(Ptr1);
New(Ptr2);
Mark(P);
New(Ptr3);
New(Ptr4);
New(Ptr5);
```

состояние кучи будет таким, как на рис. 13.

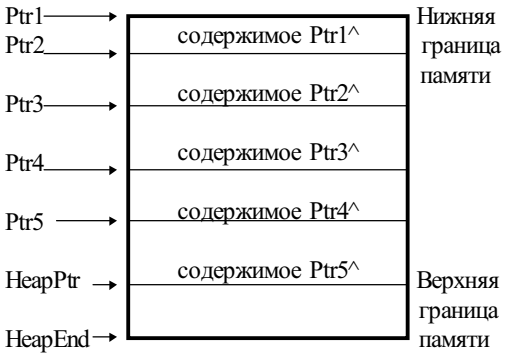


Рис. 13 Освобождение памяти с помощью Mark и Release

Оператор Mark(P) помечает состояние кучи перед распределением Ptr3 (сохранением текущего HeapPtr в P). Если выполнить оператор Release(P), то состояние кучи станет как на рис. 14, эффективно освобождая все указатели, распределенные после вызова Mark.

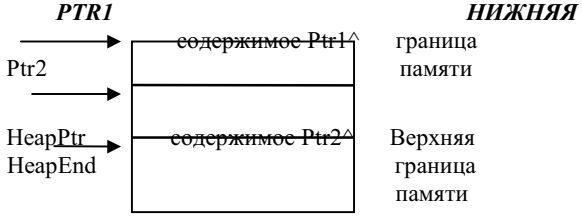


Рис. 14 Распределение кучи после выполнения Release(P)

Примечание: Выполнение оператора Release(HeapOrg) полностью очищает всю кучу, поскольку HeapOrg указывает на нижнюю границу кучи.

Для программ, которые освобождают указатели в порядке, точно обратном порядку их распределения, процедуры Mark и Release очень эффективны. Однако большинство программ распределяют и освобождают указатели случайным образом, что требует более сложной техники управления, это и реализуется процедурами Dispose и FreeMem. Эти процедуры позволяют программе освобождать любой указатель в любое время.

Когда динамическая переменная, которая не является последней (верхней) в куче, освобождается с помощью Dispose или FreeMem, куча становится фрагментированной. Если была выполнена та же последовательность операторов, а затем Dispose(Ptr3) – в середине кучи появится дырка (рис 15).

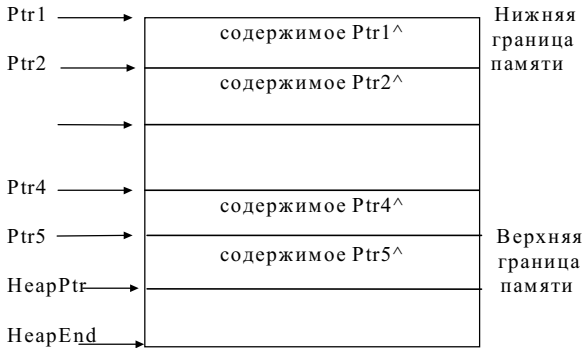


Рис. 15 "Дырка" в куче

Если сейчас выполнить New(Ptr3), то он снова займет ту же область памяти. С другой стороны, выполнение Dispose(Ptr4) увеличит свободный блок, поскольку Ptr3 и Ptr4 были соседними блоками (рис. 16).

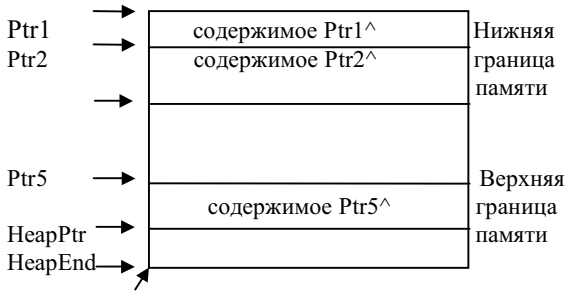


Рис. 16 Увеличение свободного блока

Наконец, выполнение Dispose(Ptr5) во-первых создаст еще больший свободный блок, а затем переместит HeapPtr вниз. Кроме того, этот свободный блок сольется со свободной памятью кучи, так как последний значащий указатель сейчас – Ptr2 (рис. 17).

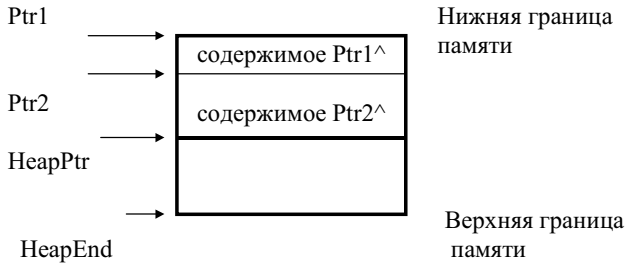


Рис.17 Удаление свободного блока

Куча сейчас в таком же состоянии, как была после выполнения Release(P) (рис. 14). Однако, свободные блоки, создаваемые и разрушаемые в этом режиме, фиксировались для последующего использования.

Список свободных блоков

Адреса и размеры свободных блоков, создаваемых Dispose и FreeMem, хранятся в списке свободных блоков, который растет сверху вниз от верхней границы сегмента кучи. Когда распределяется динамическая переменная, то до размещения ее в куче проверяется список свободных блоков. Если есть свободный блок подходящего размера (размер больше или равен размеру распределяемого блока), то он используется.

Примечание: Процедура Release всегда очищает список свободных блоков, что заставляет монитор кучи "забыть" о всех свободных блоках, которые могли быть ниже указателя кучи. Если Вы смешиваете вызовы Mark и Release с вызовами Dispose и FreeMem, то Вы должны быть уверены, что таких свободных блоков не существует. Переменная FreeList из модуля System указывает на первый свободный блок в куче. Этот блок содержит указатель на следующий свободный блок, который содержит указатель на следующий свободный блок и т.д. Последний свободный блок содержит указатель на вершину кучи (т.е. на положение, указываемое HeapPtr). Если в списке свободных блоков нет, FreeList равна HeapPtr.

Формат первых 8 байт свободного блока определяется типом TFreeRec:

```
type
```

```
  PFreeRec = ^TFreeRec;
  TFreeRec = record
    Next : PFreeRec;
    Size : Pointer;
  end;
```

Поле Next указывает на следующий свободный блок, или на то же положение, что и HeapPtr, если блок – последний свободный блок. Поле Size хранит размер свободного блока. Значение Size не – обычное 32-битовое значение; скорее это нормализованное значение указателя с числом свободных параграфов (16-байтовых блоков) в старшем слове и числом свободных байт (от 0 до 15) в младшем слове. Следующая функция BlockSize преобразует значение поля Size в нормальное значение LongInt:

```
function BlockSize(Size: Pointer): Longint;
type
  PtrRec = record Lo, Hi: Word; end;
begin
  BlockSize := Longint(PtrRec(Size).Hi) * 16 +
   PtrRec(Size).Lo;
end;
```

Чтобы гарантировать, что всегда будет место для TFreeRec в начале свободного блока, монитор кучи округляет размер КАЖДОГО блока, распределяемого New или GetMem до 8-байтовой границы. Так для блоков, размером в 1..8 байт распределяется 8 байт, для блоков, размером 9 … 16 распределяется 16 байт и т.д. Это может показаться расточительным использованием памяти и в действительности будет таким, если каждый блок будет размером в 1 байт. Однако, обычно блоки имеют больший размер и относительный размер неиспользуемого пространства невелик. Более того, и это очень важно, 8-байтный коэффициент гранулированности гарантирует, что распределение и освобождение случайных блоков небольших размеров, как, например, для строк с переменной длиной в программах обработки текста, не будет сильно фрагментировать кучу. Например, допустим 50 -байтный блок распределяется и освобождается, становясь элементом в списке свободных блоков. Этот блок будет округлен до 56 байт (7*8) и последующий запрос на распределение от 49 до 56 байт будет полностью использовать этот блок, не оставляя от 1 до 7 байт свободными, которые будут фрагментировать кучу.

Переменная Heap Error

Переменная HeapError позволяет установить функцию обработки ошибок кучи, которая вызывается, когда монитор кучи не может обработать запрос на распределение памяти. HeapError указывает на функцию со следующим заголовком:

```
function HeapFunc(Size: Word): Integer; far;
```

Директива компилятора far устанавливает дальнюю модель вызова для функции обработки ошибок.
Функция обработки устанавливается присваиванием ее адреса переменной HeapError:

```
HeapError := @HeapFunc;
```

Функция обработки ошибок кучи вызывается, когда New или GetMem не могут обработать запрос. Параметр Size содержит размер блока, который не мог быть распределен и функция обработки должна попытаться освободить блок размером не меньшим этого.

В зависимости от результата функция обработки возвращает 0, 1 или 2. В случае 0 будет немедленно возникать ошибка времени выполнения в программе. В случае 1 вместо аварийного завершения программы New или GetMem возвращают указатель, равный Nil. Наконец, 2 означает успех и повторяет запрос на распределение памяти (который может опять вызвать функцию обработки ошибок).

Стандартная функция обработки ошибок кучи всегда возвращает 0, что приводит к аварийному завершению программы, если New или GetMem не могут быть выполнены. Для многих программ будет удобна следующая функция обработки ошибок:

```
function HeapFunc(Size: Word): Integer; far;
begin
   HeapFunc := 1
end;
```

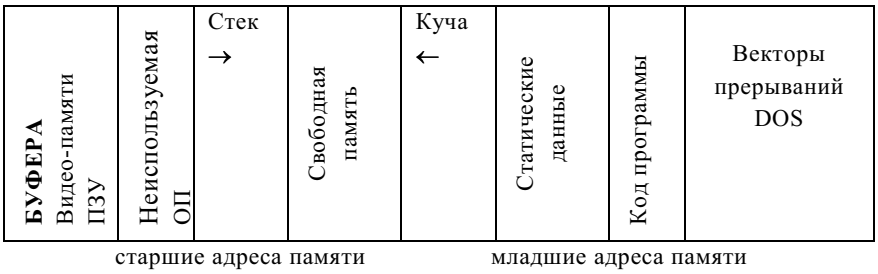
Когда эта функция установлена, New и GetMem будут возвращать nil при невозможности распределить память, не приводя к аварийному завершению программы.

Примечание. Вызов функции обработки ошибок кучи с параметром Size = 0 указывает, что для удовлетворения запроса на распределение монитор кучи расширил кучу, передвигая HeapPtr вверх. Это происходит, когда нет свободных блоков в списке свободных блоков, или когда все свободные блоки слишком малы для запроса на распределение. Вызов с Size = 0 не указывает на ошибку, поскольку существует достаточное место для распределения между HeapPtr и HeapEnd. Скорее это указывает, что неиспользованное пространство HeapPtr было уменьшено и монитор кучи игнорирует возвращаемое значение от вызовов этого типа.

Распределение памяти в СИ

ПАМЯТЬ ДЛЯ ПРОГРАММЫ НА ТУРБО СИ ТРЕБУЕТСЯ ДЛЯ ЧЕТЫРЕХ ЦЕЛЕЙ: ДЛЯ РАЗМЕЩЕНИЯ ЕЕ ПРОГРАММНОГО КОДА, ДЛЯ РАЗМЕЩЕНИЯ ДАННЫХ, ДЛЯ ДИНАМИЧЕСКОГО ИСПОЛЬЗОВАНИЯ И ДЛЯ РЕЗЕРВИРОВАНИЯ КОМПИЛЯТОРОМ НА ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ (ТАБЛ. 23).

23 РАСПРЕДЕЛЕНИЯ ПАМЯТИ В ТУРБО Си



Область памяти под код программы в процессе работы остается неизменной. Неизменной остается и память, отводимая под статические данные. Объем памяти для кучи зависит от того, сколько памяти запрашивает программист с помощью функций alloc и malloc.

Размер использованной памяти стека изменяется при активизации автоматических (локальных) переменных в функциях, а также за счет того, что при вызовах функций в стек заносятся параметры функций.

МОДЕЛИ ПАМЯТИ Си

Крошечная: Во все четыре регистра сегментов (CS, DS, SS и ES) записывается один и тот же адрес. Под код программы, статические данные, динамически размещаемые данные и стек отводится 64 К памяти. Такая модель налагает на задачу серьезные ограничения и используется только в тех случаях, когда особенно ощущается дефицит памяти.

Переменные типа указатель в такой модели памяти занимают только 2 байта (близкие указатели). Следовательно, переменные типа указатель содержат только смещение внутри фиксированного сегмента памяти.

Маленькая: Под код программы отводится сегмент размером 64 К. Стек, куча и статические данные размещаются в одном сегменте размером 64 К. Такая модель памяти принимается по умолчанию и подходит для многих маленьких и средних задач.

Переменные типа указатель в такой памяти занимают только 2 байта (близкие указатели). Следовательно, переменные типа указатель содержат смещение внутри фиксированного сегмента памяти.

Средняя: Размер памяти под код программы ограничен 1 Мб. Это означает, что в коде программы используются далекие указатели. Стек, куча и статические данные, как и в случае маленькой модели памяти, размещаются вместе в сегменте памяти размером 64 К. Такую модель рекомендуется применять при программировании очень больших программ, не использующих большого объема данных.

Для адресации (указания) в коде программы служат: далекие указатели (сегмент и смещение), занимающие 4 байта.

Таким образом, все вызовы функций выполняются как далекие вызовы, и все возвраты из функций считаются далекими. Для адресации данных используются близкие указатели, занимающие 2 байта.

Компактная: Под код программы отводится 64 К. Под данные отводится 1 Мб. Объем статических данных ограничивается 64 К, а размер стека, как и для всех моделей, не может превысить 64 К.

Такая модель памяти должна применяться при создании малых и средних по размеру программ, требующих большого объема статических данных.

Адресация внутри программы выполняется с помощью близких указателей (их размер 2 байта).

Для адресации данных используются четырехбайтовые далекие указатели.

Большая: Размер памяти под код программы ограничен 1 Мб. Под статические данные отводится 64 К. Куча может занимать до 1 Мб памяти. Такую модель приходится использовать во многих больших задачах.

Как программа, так и данные адресуются далекими указателями, занимающими 4 байта. В большой модели памяти ни одна отдельная единица данных не может превышать 64 К.

Огромная: Аналогична большой модели, но суммарный объем статических данных может превышать 64 К.

Огромная модель памяти не предусматривает огромных указателей. Всем указателям в Турбо СИ можно присвоить безопасный адрес памяти – нуль целого типа. Гарантируется, что этот адрес не совпадает ни с одним адресом, уже использованным в системе. Такой адрес называется нулевым адресом и часто применяется как ограничитель в динамических структурах.

Выше мы рассматривали следующее описание:

```
int *x;
*x=16;
```

по адресу, задаваемому в x, записывается число 16.

Компилятор резервирует память, необходимую для хранения адреса (2 байта для маленькой модели). Компилятор не отводит двух байтов для размещения целого числа по этому адресу. Если программист желает по такому адресу поместить целое значение, то он должен сам позаботиться о выделении требуемой для этого памяти.

Начальное значение адреса x может не соответствовать допустимому адресу или адресу, по которому желательно разместить целое число. Более того, начальный адрес, содержащийся в x, может, к несчастью, совпадать с адресом таблицы описаний файлов или другой важной частью операционной системы DOS. Попытка записать целое по такому адресу может привести к тому, что будет затерта часть операционной системы. К сожалению, компилятор с Турбо СИ или любой другой компилятор с СИ не смогут "отловить" ошибку такого рода.

Ответственность за инициализацию указателей полностью лежит на программисте.

Исправить такую ситуацию можно, использовав одну из функций распределения памяти – malloc, описанной в alloc, с помощью которой можно запросить память из кучи, для целого 2 байта.

Оператор, выполняющий указанные действия

```
x=(int *) malloc(sizeof (int));
```

должен быть подставлен непосредственно перед оператором *x=16;

В программу также следует поставить с помощью include файл alloc.h, содержащий описание функций динамического распределения памяти.

В качестве аргумента функции динамического распределения памяти malloc задается число байт, которые следует зарезервировать в куче. Функция всегда возвращает указатель на тип void, поэтому такой указатель совместим с любым указателем, который может встретиться в левой части оператора присваивания. Поскольку в нашем случае мы хотим, чтобы функция malloc выдала нам адрес целого (вернуть в качестве результата указатель на целое), воспользуемся преобразованием (int *) для перевода значения, выдаваемого функцией malloc, в адрес, действительно указывающий на целое.

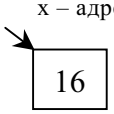
Использование функции malloc преследует две цели. Во-первых, переменной_указателю x присваивается значение адреса, которое гарантированно не совпадает с адресами, используемыми системой. Во-вторых, в памяти, отведенной для кучи, выделяются два байта, в которых можно разместить целое, и полученный адрес заносится в x.

Если в памяти под кучу нет свободного места для размещения двух байтов, то в качестве результата функции malloc будет выдан адрес 0.

Во многих ситуациях пользователь должен проверять, не выдала ли функция в качестве результат 0. Поэтому удобно использовать оператор malloc в следующей форме:

```
If (( x=(int *) malloc( sizeof(int))) !=0)
{
        /* фрагмент программы*/
}
```

Ситуация, возникающая при использовании операции malloc.



ПРОБЛЕМЫ, СВЯЗАННЫЕ С УКАЗАТЕЛЯМИ И ИХ РАЗРЕШЕНИЕ

Описание переменной_указателя в теле такой функции, как, например, main, не приводит ни к инициализации указателя, ни к резервировании в куче памяти для значения.

Существуют четыре способа задать переменной_указателю осмысленное начальное значение:

1. Описать указатель вне любой функции или снабдить его предписанием static. Начальным значением

является нулевой адрес памяти – 0. Перед тем, как начать пользоваться указателем, следует зарезервировать память под значение.

2. Присвоить указателю адрес переменной.
3. Присвоить указателю значение другого указателя, к этому моменту уже правильно инициализированному.
4. Использовать функции распределения памяти, такие, как alloc и malloc.

Первый способ основывается на важном свойстве статических переменных. Все статические переменные инициализируются значением 0.

Поскольку компилятор отводит память под переменную в момент ее описания, то присваивание указателю адреса переменной гарантирует, что нужная память отведена. После снятия ссылки значение указателя совпадает со значением переменной, адрес которой был присвоен указателю.

Когда указателю присваивается значение другого указателя, то это означает, что одному и тому же адресу памяти присваиваются разные имена (идентификаторы обеих переменных_указателей).

Если значение, на которое ссылаются два указателя, будет изменено в операторе присваивания с использованием одного из указателей, то значение, получаемое с помощью другого указателя, также будет изменено. Рассмотренная ситуация называется двойным указателем и может привести к серьезным проблемам.

Серьезная опасность возникает, если функция возвращает указатель, являющийся адресом автоматической (локальной) переменной. Автоматическая переменная описывается внутри функции. Память под нее отводится в момент активизации вызова функции. При выходе из функции память для всех автоматических переменных освобождается. Поэтому возвращенный адрес может быть позже использован системой, и информация, содержащаяся по этому адресу, может оказаться замененной новой информацией. Выход из описанного положения – никогда не возвращать адреса автоматических переменных.

Еще один источник ошибок – неосвобождение памяти, запрошенной ранее с помощью функций alloc и malloc, когда указатель уже больше не нужен. Система не способна автоматически освобождать память в куче.

Возврат (освобождение) памяти в куче выполняет функция free. В качестве аргумента функции free задается указатель, ссылающийся на освобождаемую память.

И еще одна ошибка – присваивание переменой_указателю адресного значения непосредственно.

Классы памяти в СИ

КАЖДАЯ ПЕРЕМЕННАЯ И ФУНКЦИЯ, ОПИСАННАЯ В ПРОГРАММЕ НА ТУРБО СИ, ПРИНАДЛЕЖИТ К КАКОМУ-ЛИБО КЛАССУ ПАМЯТИ. КЛАСС ПАМЯТИ ПЕРЕМЕНОЙ ОПРЕДЕЛЯЕТ ВРЕМЯ ЕЕ СУЩЕСТВОВАНИЯ И ОБЛАСТЬ ВИДИМОСТИ.

Класс памяти переменной задается либо по расположению ее описания, либо при помощи специального спецификатора класса памяти, помещаемого перед обычным описанием. Класс памяти для функции всегда external, если только перед описанием функции не стоит спецификатор static.

Все переменные Турбо СИ можно отнести к одному из следующих классов памяти:

- automatic (автоматическая, локальная);
- register (регистровая);
- extern (внешняя);
- static (статическая).

Автоматические переменные

Автоматические переменные можно описывать явно, используя спецификатор класса памяти auto. Но такой способ описания применяется редко. Обычно указание на то, что переменная является автоматической, задается неявно и следует из положения в программе точки описания такой переменной.

По умолчанию принимается, что всякая переменная, описанная внутри функции или внутри блока, ограниченного фигурными скобками, и не имеющая явного указания на класс памяти, относится к классу памяти для автоматических переменных.

Поле видимости автоматической переменной начинается от точки ее описания и заканчивается в конце блока, в котором переменная описана. Доступ к таким переменным из внешнего блока невозможен.

Память для автоматических переменных отводится динамически во время выполнения программы при входе в блок, в котором описана соответствующая переменная. При выходе из блока память, отведенная под все его автоматические переменные, автоматически освобождается. Доступ к автоматической переменной возможен только из блока, где переменная описана, так как до момента входа в блок переменная вообще не существует (т.е. под нее не отведена память).

Скалярные автоматические переменные при их описании не обнуляются. Пользователь должен сам указать начальное значение для переменных в точке их описания.

Регистровые переменные

Спецификатор памяти register может использоваться только для автоматических переменных или для формальных параметров функции.

Такой спецификатор указывает компилятору на то, что пользователь желает разместить переменную не в оперативной памяти, а на одном из быстродействующих регистров компьютера. Компилятор не обязан выполнять такое требование. На большинстве компьютеров имеется только небольшое число регистров, способных удовлетворить желание пользователя.

Спецификатор register рекомендуется использовать для переменных, доступ к которым в функции выполняется часто. Полученный в результате код будет выполняться быстрее и станет более компактным.

Существует ограничение: нельзя обращаться к адресу таких переменных. Регистровыми переменными могут быть объявлены только автоматические переменные типа short и int, а также близкие указатели.

ВНЕШНИЕ ПЕРЕМЕННЫЕ И ФУНКЦИИ

Любая переменная, описанная в файле вне какой-либо функции и не имеющая спецификатора памяти, по умолчанию относится классу памяти для внешних переменных. Такие переменные называются глобальными.

Для глобальных переменных область видимости простирается от точки их описания до конца файла, где они описаны. Если внутри блока описана автоматическая переменная, имя которой совпадает с именем глобальной переменной, то внутри блока глобальная переменная маскируется локальной. Это означает, что внутри данного локального блока будет видна именно автоматическая переменная.

Для внешних переменных память отводится один раз и остается занятой до окончания выполнения программы. Если пользователь не укажет инициализирующее значение глобальным переменным, то им будет присвоено начальное значение 0.

Структурные величины переменные – массивы, структуры и объединения – могут инициализироваться пользователем в точке их описания.

Внешние переменные видны загрузчику, осуществляющему сборку выполняемой программы из множества объектных файлов. Благодаря этому к внешним переменным возможен доступ и из других файлов. Для того, чтобы переменную можно было использовать в другом файле, для нее следует указать спецификатор памяти extern.

Пример разделенной компиляции и описание extern

```
/* файл separate.c */
        int a=6;
  /* файл test.c */
        #include <stdio.h>
        main ()
        {
        extern int a;
        printf ("a = %d\n", a);
        }
```

Вначале компилируется файл seperete.c. Затем компилируется файл test.c, и его объектный модуль объединяется с объектным модулем separate.c. Если пользователь забудет при загрузке подключить файл separate.c, то загрузчик выдаст сообщение об ошибке: неопределенный символ в модуле test.c.

Описание extern int a; показывает компилятору, что переменная a типа int описана и под нее распределена память вне данного файла. Такая переменная может быть использована так, как если бы она была описана в данном файле.

Если описание extern для переменной расположено внутри функции, то его действие распространяется на данную функцию. Если описание extern находится вне функции, то его действие распространяется от точки описания до конца файла.

Таким образом, сочетание внешних переменных и функций в одном файле и описание extern в других файлах позволяет объединить в один выполняемый файл несколько независимо скомпилированных программ.

Рекомендуется для каждого файла реализации программы (файла с расширением .c), если в них используются внешние объекты, доступ к которым будет осуществляться из других файлов, создавать интерфейсные файлы с расширением .h и помещать туда описание внешних переменных. Тогда для обеспечения доступа к внешним переменным из файлов-потребителей потребуется лишь включить в эти файлы соответствующий интерфейсный файл.

По умолчанию считается, что все функции внешние. Местом определения функции является та точка программы, где задаются параметры функции и записывается ее тело. Ко всем функциям, не имеющим спецификатора класса памяти static, обращение из других файлов оказывается возможным, если там функция описывается как внешняя. Таким образом, функция определяется один раз, но может быть описана много раз (с использованием спецификатора extern).

Статические переменные и функции

Для упрятывания функций и переменных от загрузчика используется спецификатор памяти static. Функции и переменные, для которых указан такой класс памяти, видимы лишь от точки описания до конца файла.

Если пользователь не указал инициализирующее значение, то все статические переменные, как и внешние, инициализируются значением 0.

ЕСЛИ СТАТИЧЕСКАЯ ПЕРЕМЕННАЯ ОПИСАНА ВНУТРИ ФУНКЦИИ, ТО ОНА ПЕРВЫЙ РАЗ ИНИЦИАЛИЗИРУЕТСЯ ПРИ ВХОДЕ В БЛОК ФУНКЦИИ. ЗНАЧЕНИЕ ПЕРЕМЕННОЙ СОХРАНЯЕТСЯ ОТ ОДНОГО ВЫЗОВА ФУНКЦИИ ДО ДРУГОГО. ТАКИМ ОБРАЗОМ, СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ МОЖНО ИСПОЛЬЗОВАТЬ ДЛЯ ХРАНЕНИЯ ЗНАЧЕНИЙ ВНУТРИ ФУНКЦИИ НА ПРОТЯЖЕНИИ ВРЕМЕНИ РАБОТЫ ПРОГРАММЫ, ПРИЧЕМ ТАКИЕ ПЕРЕМЕННЫЕ БУДУТ НЕВИДИМЫ ВНЕ ФАЙЛА, ГДЕ ОНИ ОПРЕДЕЛЕНЫ.

Модули Паскаля

При написании многофункциональных, объемных программ необходимо иметь понятие о модулях.

В Паскале возможен доступ к большому числу встроенных констант, типов данных, переменных, процедур и функций. Некоторые специфичны для Паскаля, другие – для IBM PC и совместимых с ним PC или для DOS. Количество различных программ велико, но почти никогда они все сразу в программах не используются. Все эти программы разделены на связанные между собой группы, называемые модулями, которые можно использовать в случае необходимости.

Модуль – это набор констант, типов данных, переменных, процедур и функций. Каждый модуль аналогичен отдельной программе. Он имеет: главное тело, которое вызывается перед стартом программы и производит необходимые действия по инициализации, когда это необходимо. Модуль – это библиотека объявлений, которую можно вставить и использовать внутри программы, что позволяет разделить программу на части и компилировать их отдельно.

Объявления внутри модуля связаны друг с другом. Например, модуль Crt содержит все объявления для программ работы с экраном PC. Паскаль предоставляет восемь стандартных модулей. Шесть из них System, Overlay, Graph, DOS, Crt, и Printer – осуществляют поддержку программ на Turbo Pascal; все они сохранены в TURBO.TPL. Две другие – Turbo3 и Craph3 – осуществляют поддержку совместимости программ, написанных в версии 3.0. Turbo Vision предоставляет целый набор модулей.

Стандартные модули

Файл TURBO.TPL содержит все стандартные модули, кроме Graph и модулей совместимости (Graph3 и Turbo3): System, Overlay, Crt, Dos и Printer. Эти модули загружаются в память вместе с Паскаль; они всегда доступны для любой программы. Файл TURBO.TPL хранится в той же директории, что и TURBO.EXE (или TPC.EXE).

Модуль System

Модуль System можно не указывать в предложении Uses, так как он автоматически присоединяется к каждой программе. Служит для поддержки большого количества процедур и функций, таких как ввод-вывод, работа с файлами и т.д.

Модуль Dos

Dos определяет многочисленные процедуры и функции Паскаль, которые эквивалентны наиболее часто используемым вызовам Dos, таким как GetTime, SetTime, DiskSize и т.д. Кроме того, здесь определяются две программы низкого уровня – MsDos и Intr, которые позволяют использовать любой вызов MS-DOS или системные прерывания.

Модуль Overlay

Модуль Overlay обеспечивает поддержку системы оверлеев

Модуль Crt

Crt обеспечивает набор специальных средств объявлений для ввода/вывода на PC: констант, переменных и программ. Их можно использовать для работы с экраном (работа с окнами, управление курсором, управление цветом). Есть возможность вводить с клавиатуры и управлять звуковым сигналом.

Модуль Printer

В модуле Printer объявляется переменная текстового файла LST, которая связывается с драйвером устройства, позволяя посылать стандартный вывод на печатающее устройство, используя Write и Writeln. Например, включив модуль Printer в программу, можно сделать следующее:

```
write (Lst,'The sum of',A:4,'and',B:4,'is');
c:=A+B;
writeln (Lst,c:8);
```

Модуль Graph

Этот файл не входит в файл TURBO.TPL, но должен находиться в том же вместе, где и вспомогательные файлы, расширения которых .BGI и .CHR. GRAPH.TPU помещается в текущий справочник, или используется справочник модулей для указания полного пути до GRAPH.TPU. (Если используется жесткий диск и программа Install, система уже установлена так, что можно использовать Graph). Файл Graph – это набор быстродействующих эффективных графических подпрограмм, которые позволяют в полной мере использовать графические возможности PC. Этот модуль реализует независимый от устройства графический драйвер, поддерживающий графические адаптеры CGA, EGA, Hercules, AT&T400, MCGA, 3270 PC, VGA и 8514.

Структура модуля

Структура модуля похожа на структуру программы, но имеет свои отличия:

```
unit <идентификатор>;
interface
uses <список модулей>;
{общие объявления}
implementation
uses <список модулей>;
  {личные объявления}
  {реализация процедур и функций}
begin
   {код инициализации}
end.
```

Заголовок модуля – слово unit, за которым следует имя модуля – идентификатор. Следующий элемент – ключевое слово interface. Это слово обозначает начало раздела интерфейса модуля, доступного для всех других модулей и программ, использующих этот модуль. В предложении uses указываются модули, которые может использовать этот модуль.

Слово uses может появляться в двух местах:

– сразу же после слова interface; в этом случае, константы или типы данных, объявленные в интерфейсах этих модулей, могут быть использованы в любых объявлениях;

– сразу же после слова implementation; в этом случае, любые объявления этого модуля могут использоваться только внутри раздела реализации.

Раздел интерфейса

Это "открытая" часть модуля, она начинается ключевым словом interface, следующим сразу за заголовком, и ограничена ключевым словом implementation. Интерфейс определяет, что является видимым (доступным) для некоторой программы (или других модулей), использующих этот модуль. Любая программа, использующая этот модуль, имеет доступ к этим видимым элементам.

В интерфейсе модуля можно объявить константы, типы данных, переменные, процедуры и функции. Как и в программе, они могут быть расположены в любом порядке, т.е разделы могут встречаться повторно

(type...var...<proc>...type...const...var)

Процедуры и функции, доступные для программы, использующей этот модуль описываются в разделе интерфейса. А их действительные тела – операторы, реализующие их, – в разделе реализации.

Объявление forward не разрешается. Тела всех обычных процедур и функций находятся в разделе реализации после раздела интерфейса, в котором перечислены их имена и заголовки. uses может появиться и в разделе implementation. Если в разделе реализации имеет место uses, то это слово следует сразу же за словом implementation.

Раздел реализации

Раздел реализации – закрытая, недоступная часть – начинается со слова implementation. Все, что объявлено в части интерфейса видимо для раздела реализации: константы, типы, переменные, процедуры и функции. Кроме того, в разделе реализации могут быть свои собственные дополнительные объявления, недоступные программам, использующим этот модуль. Программы не могут обращаться и ссылаться на них. Однако эти недоступные элементы могут использоваться (и, как правило, это делается) видимыми процедурами и функциями, заголовки которых появляются в разделе интерфейса. Предложение uses может появляться в разделе implementation. В этом случае uses следует непосредственно за ключевым словом implementation.

Если процедуры были объявлены как внешние, то в исходном файле должна быть директива {$L имя файла} в любом месте до конца модуля end. Обычные процедуры и функции, объявленные в разделе интерфейса, которые не являются встроенными, должны появляться в разделе реализации. Заголовок procedure (function) в разделе реализации должен быть такой же, как и в разделе интерфейса, или же иметь короткую форму. В краткой форме за ключевым словом (procedure или function) следует идентификатор (имя). Подпрограмма содержит свои собственные локальные объявления (метки, константы, типы, переменные, процедуры и функции). За ними следует тело главной программы. Например, в разделе интерфейса объявлены:

```
procedure  ISwap (var v1,v2: integer);
function   IMax  (v1,V2:integer);
```

Раздел реализации может быть:

```
procedure  ISwap;
var
  Temp : integer;
begin
  Temp := V1;V1:= V2;V2 := Temp;
end;  {процедуры  ISwap}

function IMax (v1,v2:integer):integer;
begin
  if V1 > V2 then
    IMax := V1
  else IMax := V2
end;  {функции IMax}
```

Подпрограммы раздела реализации (неописанные в секции интерфейса), должны иметь полный заголовок procedure/function.

Раздел инициализации

Раздел реализации модуля заключен между словами implementation и end. Но если присутствует слово begin перед end, и операторы между этими словами, то получившийся составной оператор, похожий на тело главной программы, становится разделом инициализации модуля.

Здесь инициализируются структуры данных (переменных), используемые модулем или доступные программам, использующие этот модуль. Можно использовать его для открытия файлов. Например, стандартный модуль Printer использует этот раздел для открытия на вывод текстового файла Lst. Файл Lst впоследствии можно использовать в программах в операторах Write или Writeln.

При выполнении программы, использующей некоторый модуль, раздел инициализации вызывается перед выполнением тела главной программы. Если в программе используется несколько модулей, раздел инициализации каждого модуля вызывается (в порядке, указанном в операторе uses программы) до выполнения тела главной программы.

Использование модулей

Модули, которые использует программа, уже откомпилированы и хранятся в специальном машинном коде; это не файлы типа Include.

Даже раздел интерфейса хранится в специальном двоичном формате, который использует Паскаль. Более того, стандартные модули хранятся в специальном файле TURBO.TPL и автоматически загружаются в память с Паскаль.

В результате подключения модулей к программе время компиляции программы увеличивается незначительно, приблизительно на одну секунду. Если модули загружаются из отдельных дисковых файлов, то может потребоваться дополнительное время из-за чтения с диска.

Для использования модулей необходимо, чтобы в начале присутствовало предложение uses, за которым следует список имен всех модулей, разделенных запятыми.

```
program  MyProg;
uses  thisUnit,thatUnit,theotherUnit;
```

При компиляции этой информации к таблице символов прибавляется информации из раздела интерфейса, а из раздела реализации к самой программе машинный код. Порядок описания модулей в предложении uses не имеет большого значения. Если thisUnit использует thatUnit, то можно объявить их в любом порядке. Компилятор сам определит, который из них должен следовать первым. Иначе говоря, если thisUnit использует thatUnit, а программа MyProg не вызывает какие-либо программы в подпрограмме thatUnit, то можно "спрятать" подпрограммы в программу thatUnit, опуская их в операторе uses:

```
unit   thisUnit
uses   thatUnit
...
program   MyProg;
uses   thisUnit,theotherUnit;
...
```

В этом примере thisUnit может вызвать подпрограмму thatUnit, а MyProg – подпрограммы thisUnit и theotherUnit. MyProg не может вызвать thatUnit, так как эта подпрограмма не описана в его предложении uses.

Если предложение uses отсутствует, Turbo Pascal подсоединяет стандартный модуль System.

Ссылки на описание модуля

Если модуль включен в программу, то все константы, типы данных, переменные, процедуры и функции, объявленные в интерфейсе этого модуля, становятся доступными для этой программы.

Допустим, есть модуль:

```
unit MyStuff;
interface
  const
    MyValue := 915;
    type
      MyStars=(Deneb,Antares,Betelgeuse);
  var
    MyWord : string[20];
  procedure SetMyWord(Star : MyStars);
  function  TheAnswer : integer;
  implementation
  ...
end.
```

Часть модуля, которая описана в интерфейсе, доступна и может быть использована в программе. Поэтому, можно написать следующую программу:

```
program TestStuff;
uses MyStuff;
var
  I : integer;
  AStar : MyStars;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I);
end.
```

После включения предложения uses MyStuff в программу, появилась возможность ссылаться на все объявления и описания в секции интерфейса модуля MyStuff (MyWord, MyValue,и т.д). Рассмотрим следующую ситуацию:

```
program TestStuff;
```

```
uses MyStuff;
const
  MyValue := 22;
var
  I : integer;
  AStar : MyStars;
function TheAnswer : integer;
begin
  TheAnswer := -1;
end;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I);
end.
```

В этой программе некоторые идентификаторы, объявленные в MyStuff, переопределяются. При выполнении эта программа будет использовать собственные описания для MyValue и TheAnswer, так как они были описаны позже, чем в MyStuff.

Если нужно использовать идентификаторы из MyStuff, то в этом случае при описании перед каждым идентификатором помещается MyStuff с точкой (.). Например:

```
program  TestStuff;
uses MyStuff;
const
  MyValue = 22;
var
  I : integer;
  Astar : MyStars;
function TheAnswer : integer;
begin
  TheAnswer := -1;
end;
begin
  Writeln(MyStuff.MyValue);
  Astar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I:= MyStuff.TheAnswer;
  Writeln(I);
end.
```

Эта программа отрабатывает так же, как и первая, даже если MyValue и TheAnswer были переопределены. В действительности, первую программу можно было написать:

```
program  TestStuff;
uses MyStuff;
var
  I : integer;
  AStar : MyStuff.MyStars;
begin
  Writeln(MyStuff.MyValue);
  AStar := MyStuff.Deneb;
  MyStuff.SetMyWord(AStar);
  Writeln(MyStuff.MyWord);
  I := MyStuff.TheAnswer;
  Writeln(I);
end.
```

Заметим, что идентификаторы – константы, типы данных, переменные или подпрограммы – могут быть предварены именем модуля.

Предложение uses раздела реализации

Паскаль 7.0 дает возможность использовать предложение uses в разделе реализации. Это предложение должно немедленно следовать за ключевым словом implementation так же, как и предложение uses в разделе интерфейса появляется сразу же за ключевым словом interface.

Предложение uses в разделе реализации позволяет сделать недоступными некоторые детали модуля, поскольку модули, используемые в разделе реализации, невидимы пользователям этого модуля. Однако более важно, что это так же позволяет конструировать взаимно-зависимые модули.

Поскольку в Turbo Pascal модули необязательно должны быть строго иерархическими, можно задавать циклические ссылки модулей. Следующий раздел показывает пример, который демонстрирует циклические ссылки модулей.

В предложении Uses главной программы можно не указывать имена всех модулей, используемых программой, так как программа имеет четко выраженную иерархическую структуру (рис. 18).

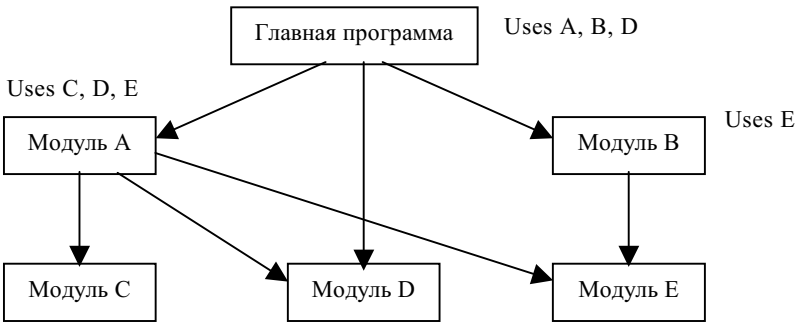


Рис. 18 Иерархическая структура программ

Циклические ссылки модулей

Следующая программа показывает, как два модуля могут использовать друг друга. Главная программа Circular вызывает модуль Display. Модуль Display содержит одну программу в разделе интерфейса, WriteXY, которая имеет три параметра: (X, Y) – координаты точки и выводимое на экран сообщение. Если значение координат (X, Y) находится в пределах видимости на экране, то подпрограмма WriteXY устанавливает курсор в точку с координатами (X, Y) и выводит сообщение. В противном случае вызывается подпрограмма выдачи ошибок.

Итак, WriteXY вычисляет координаты для Write. Как подпрограмма выдачи ошибочных сообщений выдает сообщение? Используя опять же подпрограмму WriteXY. Итак, программа WriteXY вызывает подпрограмму ShowError, которая в свою очередь вызывает WriteXY для выдачи сообщения на экран, т.е. для расчета координат. Рассмотрим программу Circular. Она чистит экран и три раза обращается к программе WriteXY:

```
program  Circular;
{вывод текста программой WriteXY}
uses
  Crt,Display;
begin
  ClrScr;
  writeXY(1,1,верхний левый угол экрана);
  writeXY(100,100,вне экрана);
end.
```

Координаты (X, Y) при втором вызове WriteXY установлены жестко как (100, 100) при экране 80 × 25. Посмотрим, как работает программа WriteXY. Ниже приведен исходный код модуля Display, который содержит процедуру WriteXY. Если координаты (X, Y) действительно находятся в пределах экрана, она выводит сообщение, в

противном случае выводится сообщение об ошибке.

```
unit  Display;
{содержит подпрограмму выдачи сообщений}

interface

procedure WriteXY(X,Y   :    integer;
       Message  : string);

implementation

 uses
    Crt, Error;
 Procedure WriteXY;
 begin
   if (X in[1..80]) and (Y in[1..25]) then
   begin
      GotoXY(X,Y);
      Write(Message);
  end
  else
     ShowError('неверны координаты для writeXY');
end;

end.
```

Процедура ShowError, вызываемая WriteXY, объявлена в следующем ниже тексте модуля Error. Программа ShowError всегда выводит свои сообщения на 25 строке экрана.

```
unit Error;
{содержит подпрограмму выдачи ошибок}

interface

procedure ShowError (ErrMsg : string);
implementation
uses
   Display;

procedure ShowError (ErrMess : string);
begin
   WriteXY (1, 25, ErrMsg); {Сообщение об ошибке}
end;
end.
```

Отметим, что в модулях Display и Error, в их предложениях uses в разделах реализации есть ссылки этих модулей друг на друга. Эти два модуля могут ссылаться друг на друга в разделах реализации, потому что Паскаль может компилировать полностью интерфейсные разделы для обоих модулей. Другими словами, компилятор Паскаль допускает ссылку на частично откомпилированный модуль A из раздела реализации модуля B, поскольку интерфейсные разделы модулей A и B не зависят друг от друга (и следовательно, соответствуют строгим правилам Паскаля для порядка объявлений).

Разделение других объявлений

Предположим, необходимо модифицировать программы WriteXY и ShowError так, чтобы добавить дополнительный параметр, который задает прямоугольное окно на экране:

```
procedure writeXY(SomeWindow:Wind Rec;
                  X, Y : integer;
                  Message : string);
```

```
procedure ShowError (SomeWindow : WindRec;
                     ErrMsg     : string);
```

Вспомним, что эти процедуры находятся в разных модулях. Даже если объявить модуль WindData в разделе интерфейса одного модуля, то это объявление не будет доступно в разделе интерфейса другого модуля.

Лучше объявить третий модуль, который состоит из одних объявлений записи окна:

```
unit WindData;
interface
type
  WinDRec = record
       X1, Y1, X2, Y2 : integer;
       TextColor,
       BackColor      : byte;
       Active         : boolean;
  end;
implementation  {Здесь отсутствует секция реализации}
end;
```

Сейчас интерфейсные разделы обоих модулей Display и Error могут видеть WindData. Это допустимо, потому что в модуле WindData нет предложения uses, а модули Display и Error имеют ссылки друг на друга в разделе implementation (реализации). Все глобальные объявления помещают в отдельный модуль.

Создание собственных модулей

Пусть модуль IntLib, помещен в файл INTLIB.PAS и откомпилирован; результат компиляции – файл INTLIB.TPU. Для того, чтобы можно было его использовать в программе, его необходимо описать в операторе uses. Таким образом программа будет выглядеть:

```
program MyProg;
uses IntLib;
```

Паскаль предполагает, что файл, в котором находится модуль, имеет такое же имя (до 8 символов), что и имя модуля. Если модуль – MyUtilities, то Паскаль ищет файл с именем MYUTILIT.PAS.

Компиляция модулей

Модуль компилируется также как и программа: создается при помощи редактора, командой Alt-F9.

Но, вместо файла с расширением .EXE, создается файл с расширением .TPU (модуль Паскаль). Можно оставить этот файл как одиночный файл. Можно поместить его в TURBO.TPL при помощи TPUMOVER.EXE.

В любом случае, можно поместить файл .TPU в справочник модулей, который задается в окне ввода Unit Directories (Options/Directories). Таким образом, можно ссылаться на эти файлы, когда они не находятся в текущем справочнике или в TURBO.TPL.

Можно иметь только один модуль для исходного файла; компиляция останавливается, когда достигается последний оператор end.

Чтобы найти модуль, указанный в операторе uses, компилятор вначале просматривает резидентные модули – модули, загруженные в память во время запуска компилятора Паскаль из TURBO.TPL. Если этого модуля нет среди резидентных, компилятор считает, что он должен быть на диске. Он вначале ищет в текущей директории, а затем в директории, заданной командой O/D/Unit Directories или директивой /U в командной строке TPC. Например, конструкция

```
uses Memory;
```

где Memory не резидентный модуль, заставляет компилятору искать MEMORY.TPU в текущей директории, а затем в каждой из директорий модулей.

Когда команды Compile/Make и Compile/Build компилируют модули, заданные в операторе uses, исходные файлы ищутся так же, как и .TPU файлы и имя исходного файла модуля принимается то же, что и имя модуля с расширением .PAS.

Пример:

Рассмотрим небольшой модуль под названием IntLib, состоящий из двух простых программ: процедура ISwap и функция IMax:

```
unit IntLib;
interface
```

```
procedure ISwap (var I, J : integer);
function IMax (I, J : integer) : integer;
implementation
procedure ISwap;
var
  Temp : integer;
begin
  Temp := I; I := J; J := Temp;
end; {of proc ISwap}
procedure IMax;
begin
  if I > J then
    IMax := I
  else
    IMax := J;
end;{функции IMax}
end.{модуля IntLib}
```

Текст модуля находится в файле INTLIB.PAS. После компиляции результирующий код помещается в файл INIT.TPU, а затем в директорию модулей, если он есть, или остается в той же директории, где находится программа. Следующая программа использует модуль IntLib:

```
program IntTest;
uses IntLib;
var
  A, B : integer;
begin
  Write ('Введите две переменные целого типа : ');
  Readln (A, B);
  ISwap (A, B);
  Writeln ('A= ', A, 'B= ', B);
  Writeln ('Максимальная из них ', IMax (A, B);
end. {программы IntTest}
```

В примере показано как создать модуль и использующую его программу.

Модули и большие программы

Обычно большая программа делится на модули, которые состоят из процедур и функций, сгруппированных по назначению. Например, программы редактора можно разделить на инициализацию, вывод, чтение и запись файла, форматирование и т.д. Так же может быть глобальный модуль – модуль используемый всеми другими модулями, а так же главной программой – он определяет глобальные константы, типы данных, переменные, процедуры и функции. Схема большой программы:

```
program Editor
uses
  DOS,Crt,Printer {стандартные модули из TURBO.TPL}
  EditGlobals,    {модули, написанные пользователем}
  EditInit,
  EditPrint,
  EditRead,
  EditWrite,
  EditFormat;
 {объявления программы, процедуры и функции}
begin {главная программа}
 end.{программы Editor}
```

Модули этой программы могут находиться в TURBO.TPL или существовать как отдельные .TPU файлы. В последнем случае Turbo Pascal будет управлять проектом. Это значит, что при перекомпиляции программы Editor, Turbo Pascal, проверив дату файлов .PAS и .TPU, перекомпилирует только те модули и файлы, которые были модифицированы.

Организация программ

Turbo Pascal версии 7.0 позволяет разделить программу на кодовые сегменты. Главная программа после

компиляции занимает сегмент. Это значит, что она не может занимать памяти больше 64 Кб.

Однако, имеется возможность увеличить этот верхний предел, разбив Вашу программу на модули. Каждый модуль может содержать до 64 Кб машинных кодов при компиляции.

Для этого можно объединить все глобальные определения – константы, типы данных, переменные – в один модуль. Его можно назвать MyGlobals. В отличие от включаемых файлов, модули не могут "видеть" любые определения, сделанные в главной программе; они "видят" только то, что определено в интерфейсной части их собственного модуля и в других модулях, используемых ими. Поэтому модуль может использовать MyGlobals и обращаться ко всем глобальным объявлениям.

Второй возможный модуль – MyUtils. В котором можно собрать подпрограммы, используемые программе. Здесь должны быть собраны подпрограммы , которые не зависят от каких-либо других подпрограмм.

Кроме этого, можно объединить процедуры и функции в логические группы. В каждой группе можно определить несколько процедур и функций, которые наиболее часто используются программой и затем процедуры и функции, которые используются несколько реже. Разбиение большой программы на модули может производиться в приведенной последовательности:

1 Скопируйте все эти процедуры и функции в отдельный файл и удалите их из главной программы.

2 Откройте этот файл для редактирования.

3 Наберите следующие строки перед процедурами и функциями:

```
unit имя модуля;
  intarface
uses MyGlobals
implementation,
```

ГДЕ ИМЯ МОДУЛЯ – ИМЯ ВАШЕГО МОДУЛЯ (И ТАК ЖЕ ИМЯ РЕДАКТИРУЕМОГО ФАЙЛА).

4 Наберите оператор end, в конце файла.

5 Между interface и implementation скопируйте заголовки процедур и функций, вызываемых из главной программы. Заголовок – это первая строка подпрограммы вместе со словами procedure или function.

6 Если этот модуль использует другие модули, приведите их имена через запятую в предложении Uses.

7 Откомпилируйте этот файл.

8 Вернитесь в Вашу главную программу и добавьте имя этого модуля в предложение Uses.

Средства Build и Make

Turbo Pascal включает в себя очень важное и очень нужное средство управления проектом – встроенную утилиту Make. Рассмотрим ее назначение

Допустим имеется программа MAIN.PAS, которая использует четыре модуля: A, B, C, D. Эти четыре модуля – четыре текстовых файла A.PAS, B.PAS, C. PAS, D.PAS. Далее B использует A, а C и D используют и A и D (рис. 19).

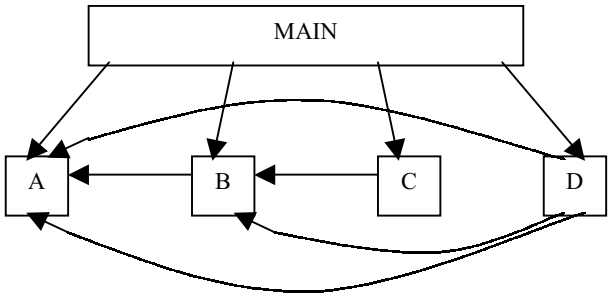


Рис. 19 Схема взаимодействия модулей

При компиляции MAIN.PAS компилятор ищет файлы A.TPU, B.TPU, C.TPU и D.TPU, загружает их в память, собирает их коды в файл MAIN.PAS, компилирует и записывает его в файл MAIN.EXE (если компилируется на диск).

Допустим, что мы вносятся изменения в C.PAS. Теперь для создания MAIN.EXE необходимо перекомпилировать и C.PAS и MAIN.PAS. Это немного скучная, но не сложная задача.

Допустим, изменения внесены в секцию интерфейса A.PAS. Для создания новой версии MAIN.EXE необходимо перекомпилировать уже все четыре модуля и сам MAIN.PAS. Это означает, что при каждом изменении в модуле A.PAS, требуется перекомпиляция всех пяти модулей.

Make

Turbo Pascal предлагает решение этой проблемы: можно использовать опцию Make в меню Compile и Turbo Pascal выполнит всю работу. Процесс очень простой: после внесения изменений в какой-либо модуль или в главную программу перекомпилировать надо только главную программу.

Turbo Pascal осуществляет три вида проверок:

1 Во-первых, проверка даты и времени для каждого модуля, используемого программой. Дата сверяется у

файлов с расширениями .PAS и .TPU. Если в файлы (.PAS) вносились изменения с тех пор, как был создан соответствующий .TPU, то этот файл .PAS перекомпилируется заново, создавая обновленный файл .TPU. Поэтому в первом примере, когда изменения вносятся в C.PAS, Turbo Pascal автоматически откомпилирует C.PAS перед компиляцией MAIN.PAS (при условии использования опции Make).

2 Вторая проверка: были ли внесены изменения в секцию интерфейса модифицируемого модуля. Если это имело место, то Turbo Pascal заново от компилирует все модули, использующие данный модуль.

Во втором примере изменения внесены в раздел интерфейса A.PAS, модуль MAIN.PAS компилируется заново. Turbo Pascal перед компиляцией MAIN.PAS автоматически перекомпилирует A. PAS, B.PAS, C.PAS, D.PAS (в описанном в uses порядке). Однако, если был модифицирован только раздел реализации, то перекомпиляция других зависимых модулей не требуется, поскольку (с их точки зрения) не изменился этот модуль.

3 Третья проверка касается включаемых и .OBJ файлов, содержащих подпрограммы на Ассемблере, используемых каким-либо модулем. Если данный файл .TPU создан раньше, чем какой-нибудь файл включаемый или .OBJ, с которым он собирается, то соответствующий модуль компилируется заново. Таким образом, если были внесены изменения в подпрограммах, написанных на Ассемблере, используемые модулем, этот модуль автоматически перекомпилируется, когда компилируется программа, использующая этот модуль.

Для использования опции Make надо выбрать команду Make в меню Compile или нажать F9. При работе с компилятором командной строки указать опцию /M. Опция Make не воздействует на модули, находящиеся в TURBO.TPL.

B u i l d

Опция Build – это частный случай Make. При использовании Build перекомпилируются все модули, используемые данной программой, исключая модули из библиотеки TURBO.TPL. Это более простой и надежный способ, что все будет обновлено.

Для вызова Build из командной строки используется опция /B.

Автономная утилита Make – это программа, поставляемая с Turbo Pascal.

Turbo Pascal предлагает большой набор мощных средств для управления и создания больших и сложных программ, построенных из многочисленных модулей, исходных и объектных файлов. Система предлагает автоматическое выполнение операций Make и Build, перекомпилируя модули в случае необходимости. В то же время, Turbo Pascal не имеет средств для получения обновленных .OBJ файлов (файлов объектных кодов) подпрограмм, написанных на языке Ассемблера (.ASM файлов), в случае модификации последних. Для этого необходимо использовать отдельный Ассемблер. Вопрос: Как получить последние версии файлов с расширениями .ASM и .OBJ и как их подключать к программам? Ответ прост. Используется автономная утилита Make, которая поставляется вместе с системой Turbo Pascal .

Make – интеллектуальный программный администратор, который при задании определенных команд, выполнит необходимую работу по сохранению и обновлению программ. На самом деле возможности утилиты Make значительно шире. В ее функциональные возможности входит:

– создание резервных копий;

– удаление файлов из различных поддиректорий;

– автоматический запуск программ с внесением изменений в используемые файлы данных .

По мере изучения использования утилиты Make, можно увидеть и другие возможности и способы применения этой утилиты для разработки программного обеспечения.

Make – автономная утилита; она отличается от опций Make Build, которые включены в IDE и компилятор командной строки

Небольшой пример:

Допустим, надо написать несколько программ, выводящих на дисплей некоторую информацию о ближайших звездных системах. Одна программа GETSTAR.PAS считывает в текстовый файл список звездных систем, обрабатывает его и создает двоичный файл этой информации.

GETSTAR.PAS использует три модуля: STARDEFS.TPU, который содержит глобальные определения; STARLIB.TPU, содержащий некоторые утилиты (вспомогательные подпрограммы); STARPROC.TPU, который делает основную обработку информации. Исходные коды их находятся, соответственно, в файлах: STARDEFS.PAS, STARLIB.PAS, STARPROC.PAS. Определим следующие зависимости:

– STARDEFS.PAS не использует никаких других модулей;

– STARLIB.PAS использует STARDEFS;

– STARPROC.PAS использует STARDEFS и STARLIB;

– GETSTAR.PAS использует STARDEFS, STARLIB и STARPROC.

Для получения GETSTAR.EXE необходимо просто "сделать" (откомпилировать) GETSTAR.PAS. Turbo Pascal будет перекомпилировать модули по необходимости.

Допустим, есть несколько подпрограмм из STARLIB.PAS – написанных на Ассемблере файлов SLIB1.ASM и SLIB2.ASM. После работы Turbo Assembler получаем файлы SLIB1.OBJ и SLIB2.OBJ. Каждый раз STARLIB.PAS при компиляции компонуется с обоими .OBJ файлами. Фактически, Turbo Pascal заново перекомпилирует STARLIB.PAS, если файл STARLIB.TPU создан раньше, чем какой-либо из этих .OBJ файлов.

Что будет в случае, если какой-либо из .OBJ файлов окажется созданным раньше, чем файлы .ASM, от которых они зависят? Это значит, что соответствующий файл с расширением .ASM должен быть реассемблирован. Turbo Pascal не может ассемблировать такие файлы. Что же делать?

Необходимо создать командный файл для Make, вызвать утилиту Make, подав ей этот командный файл. Этот файл состоит из зависимостей и команд. Зависимости определяют зависимость файлов друг от друга. Команды указывают Make, как создать данный файл из других файлов.

Создание командного файла для Make

Файл для Make в этом случае должен выглядеть:

```
GETSTARS.EXE : GETSTARS.PAS STARDEFS.PAS STARLIB.PAS SLIB1.ASM\SLIB2.ASM SLIB.OBJ SLIB2.OBJ TPC GETSTARS /M
SLIB1.OBJ : SLIB1.ASM
   TASM SLIB1.ASM SLIB1.OBJ
SLIB2.OBJ : SLIB2.ASM
   TASM SLIB2.ASM SLIB2.OBJ
```

– Первые две строки поясняют Make, что GETSTARS.EXE зависит от трех файлов написанных на Turbo Pascal; двух файлов, написанных на ассемблере; двух .OBJ файлов. Обратный слэш (\) в конце первой строки – символ строки продолжения, т.е. строка оператора продолжается далее.

– Третья строка – указание Make, как создается новая версия GETSTARS.EXE. Заметим, что для обработки GETSTARS.PAS используется компилятор командной строки и использует встроенное в Turbo Pascal средство Make (параметр /M).

– Следующие две строки (пустые строки игнорируются) говорят о том, что SLIB1.OBJ зависит от SLIB1.ASM и о том, как получить SLIB1.OBJ.

– Последние две строки определяют зависимости (для одного файла) и действие Make и SLIB.OBJ.

Использование Make

Допустим, создан этот командный файл при помощи встроенного в интегрированную среду редактора (или же каким-либо другим редактором), записан и сохранен в файле STARS.MAK. Его можно задать командой:

```
make –f STARS.MAK
```

где –f – опция, указывающая Make, какой файл использовать. Make обрабатывает этот файл с конца и до вершины файла. Во-первых, она проверяет даты SLIB2.OBJ и SLIB2.ASM, если SLIB2.OBJ старее, то Make вызывает команду:

```
TAMS SLIB2.ASM SLIB2.OBJ,
```

которая ассемблирует SLIB2.ASM и создает новую версию SLIB2.OBJ. Такие же действия (проверка даты и ассемблирование) производятся с SLIB1.OBJ. В заключение, проверяются все зависимости GETSTARS.EXE и, если необходимо, выполняется команда:

```
tpc GETSTARS/M.
```

Опция /M указывает Turbo Pascal использовать внутреннюю программу Make, которая просмотрит все связи, включая перекомпиляцию файла STARLIB.PAS, если дата создания SLIB1.OBJ или SLIB2.OBJ окажется более новой, чем дата создания STARLIB.TPU.

Оптимизация кода программы

Некоторые опции компилятора оказывают влияние, как на размер, так и на скорость выполнения программы. Происходит это потому, что в зависимости от этих опций в программу вставляется код проверки ошибок и их обработки. Эти опции лучше использовать при разработке программ. В окончательной версии их рекомендуется исключать (уменьшается размер и сокращается время выполнения). Ниже приведен список опций и их значений, которые используются для оптимизации программы.

{$A+} позволяет выравнивать переменные и типизированные константы на границу слова. Это обеспечивает более быстрый доступ к памяти для систем 80 × 86. По умолчанию эта опция включена.

{$I-} отключает проверку ошибок ввода/вывода. С помощью вызова встроенной функции IOResult в программе, можно обрабатывать ошибки ввода/вывода самим. По умолчанию используется {$I+}.

{$N-} генерирует код 8087, выполняющий все операции с плавающей точкой, используя встроенный 6-байтовый тип Real. Когда эта опция включена, будет использоваться аппаратное обеспечение 8087 или программная эмуляция. Если Вы компилируете программу и все модули, используя {$N-}, библиотека времени выполнения 8087 не требуется и Turbo Pascal игнорирует директиву $E. По умолчанию – {$N-}.

{$R-} выключает проверку диапазона. При генерации кода не осуществляется проверка ошибок в индексировании массивов и не проверяется принадлежность присваиваемых значений допустимому диапазону. По умолчанию эта опция отключена.

{$S-} выключает проверку стека. При генерации кода не проверяется наличие достаточной памяти для стека каждого вызова процедуры или функции. По умолчанию эта опция отключена.

{$V-} отменяет проверку параметров var, которые являются строками. Это позволяет передавать в качестве фактических параметров строки, длина которых отлична от типа, определенного для формального параметра var. По умолчанию эта опция включена.

{$X+} разрешает использовать вызовы функций как операторы; результат вызова функции может быть отброшен.

Использование этих опций означает определенный риск. Пользоваться ими надо осторожно. Если же программа поведет себя непонятно и непредсказуемо, то их следует включить вновь.

Помимо возможности включения этих средств в исходный код, имеется возможность установить их с помощью меню Options/Compiler в интегрированной среде или с помощью опции $X в командной строке, (где X – это соответствующая буква директивы компилятора).

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ СРЕДСТВА СИСТЕМЫ ПРОГРАММИРОВАНИЯ

ОБЪЕКТЫ ПАСКАЛЯ

ПОД ОБЪЕКТОМ ПОНИМАЮТ ТИП ДАННЫХ, В КОТОРОМ ОБЪЕДИНЕНЫ ДАННЫЕ И ФУНКЦИИ ДЛЯ РАБОТЫ С ЭТИМИ ДАННЫМИ. С ОБЪЕКТНО-ОРИЕНТИРОВАННЫМ ПРОГРАММИРОВАНИЕМ СВЯЗАНО ТРИ ПОНЯТИЯ: ИНКАПСУЛЯЦИЯ (ОБЪЕДИНЕНИЕ ДАННЫХ РАЗЛИЧНЫХ ТИПОВ И МЕТОДОВ ИХ ОБРАБОТКИ), НАСЛЕДОВАНИЕ (ДАННЫХ И МЕТОДОВ ОБЪЕКТОМ НАСЛЕДНИКОМ ОТ ОБЪЕКТА ПРЕДКА) И ПОЛИМОРФИЗМ (ИМЕЕТ МЕСТО КОГДА ОДНА И ТА ЖЕ ФУНКЦИЯ ПРИМЕНЯЕТСЯ К АРГУМЕНТАМ РАЗЛИЧНЫХ ТИПОВ, ИЛИ ФУНКЦИЯ С ОДНИМ ИМЕНЕМ ИМЕЕТ НЕСКОЛЬКО ИНТЕРПРИТАЦИЙ ИЛИ ОБЪЕДИНЕНИЕ ДАННЫХ РАЗЛИЧНЫХ ТИПОВ В ЕДИНУЮ ПЕРЕМЕННУЮ).

Объявление объекта

В операторе Type задается имя объекта и указывается что это объект. После чего задается поля объекта.

```
Type
  TData = object
     Name: string[30];
     Date: string[30];
end;
```

Наследование объектов

Объект может быть определен, как потомок уже существующего объекта, т.е. он может наследовать поля и методы предыдущего объекта.

```
Type
  TData = object
     Name: string[30];
      Date: string[30];
end;
     TStudent = object (TDate)
       Ball: real;
end;
```

В данном случае TDate является родительским типом, а TStudent является дочерним.

Экземпляры типа Object

Объект может описываться, как любая статическая или динамическая переменная, ссылающаяся на размещенную в динамической памяти переменную

```
Type
   PStudent = ^Tstudent;
Var
    Stat_st: TStudent;
    Din_st: PStudent;
```

Инициалиизация полей объектов

Для удобства в объектно-ориентированном программировании для инициализации полей объекта используют процедуру Init

```
Type
```

```
TDate = object
    Name: string[30];
    Date: string[30];
    Procedure Init (Nm,Dt: string);
end;
Procedure TDate.Init (Nm,Dt string);
  Begin
    Name := Nm; {инициализация поля name}
    Date := Dt; {инициализация поля date}
end;
```

Определение методов

Внутри объекта метод определяется заголовком процедуры или функции, действующей как метод. Причем сами методы описываются вне определения объекта как отдельная процедура или функция.

```
Type
  Tdate = object
    Name: string[30];
          Date: string[30];
          Procedure Init (Nm,Dt: string);
          Function GetName: strring;
          Function GetData: string;
   end;
   Procedure Tdate.Init (Nm,Dt: string);
   Begin
          Name := Nm ;
          Date :=Dt;
   end;
   Function Tdate.GetName: string;
   Begin
          GetName := Name;
   end;
   Function Tdate.GetDate: string;
   Begin
          GetDate := Date;
   end;
```

Поля данных объектов и формальные параметры методов

Поскольку методы и их объекты имеют общую область действия, формальные параметры метода не могу быть идентичными любому из полей данных объекта (так же как и в процедурах запрет на идентичные локальные параметры). Выдается сообщение Duplicate identifier

Включение объектов в модули

включении объекта в модуль в модуле объявляем объект со всеми его полями и методами. В программе указывается модуль, в котором находится этот объект и описывается экземпляр типом этого объекта в разделе объявления переменных:

```
Program Use object;
uses Date; {Модуль в котором определен объект}
Var date:Tdate;
```

Переопределение методов

Для переопределения методов используют наследование (при определении порожденного типа им наследуются методы порождающего типа, однако они могут переопределяться). Для переопределения наследуемого метода описывается новый метод с тем же именем, что и наследуемый метод. но с другим телом и параметрами (при необходимости).

```
Tstudent = object (Tdate)
        Ball: real;
        Procedure Init (Nm,Dt: string; bl: real);
        Function GetBall: real;
        Function GetSum: real;
        Procedure ShowSum;
        Procedure ShowBall;
```

```
end;
Procedure Tstudent.Init (Nm,Dt: string; bl: real);
 Begin
        Tdate.Init (Nm,Dt); {Здесь можно выполнить собственную инициализацию}
        Ball := bl;
end;
```

C o n s t r u c t o r

Процедура, отвечающая за создание объекта. В этой процедуре можно выполнить инициализацию полей

```
Type
        .....
        Constructor Init (Nm,Dt: string);
        .....
end:
Constructor Tdate.Init (Nm,Dt: string);
 begin
        Name := Nm;
        Date := Dt;
 end;
```

D e s t r u c t o r

Процедура, отвечающая за уничтожение объекта

```
Type
        .....
        Constructor Init (Nm,Dt: string);
        .....
        Destructor Done;
end:
Constructor Tdate.Init (Nm,Dt: string);
 begin
        Name := Nm;
        Date := Dt;
 end;
.....................
Destructor Tdate.Done;
 begin
        clrscr;
 end;
```

Р а с ш и р е н н о е и с п о л ь з о в а н и е о п е р а т о р а New

В объектно-ориентированном программировании New выполняет выделение памяти для объекта в динамической области и инициализацию самого объекта с помощью вызова его конструктора.

```
Type
    Pperson = ^Tdate;
Var person: Pperson;
.............
person := New (Pperson);
```

или

```
person := New (Pperson, Init('Alex','05-02-1999'));
```

или

```
New (person, Init(Nm,Dt));
```

Р а с ш и р е н н о е и с п о л ь з о в а н и е о п е р а т о р а Dispose

Очищает выделенную память в динамической области и уничтожает объект по средством вызова его Destructor'a

```
Dispose (person,Done);
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ ОБЪЕКТОВ В ПАСКАЛЕ: ВЫЧИСЛЕНИЕ ЗНАЧЕНИЯ ОПРЕДЕЛЕННОГО ИНТЕГРАЛА МЕТОДАМИ СИМПСОНА И ПРЯМОУГОЛЬНИКОВ **{В МЕТОДЕ СИМПСОНА ИСПОЛЬЗУЕТСЯ ФОРМУЛА**

$$Y(X + H) = Y(X) + H * (F(X) + 4*F(X + H/2) + F(X + H)) / 6\}$$

```
uses crt;
const
 xn = 0.6;
 xk = 1.4;
Type Tf=object {Объявление объекта}
     h1, x, y1, y2 : real; {Поля объекта}
     Constructor Init(h:real); {Определение методов}
     function f(x1:real):real;
     function square:real;
     function simpson:real;
     Destructor Done;
End;

constructor tf.init(h:real); {создание объекта и инициализация поля h1}
begin
h1:=h;
end;

function tf.f(x1:real):real;
     begin
     f:= sqrt( x1 ) * cos( x1 )
     end;
function tf.square:real;
begin {расчет по методу прямоугольников}
 x:= xn; y1:= f(x); y2:= y1 + h1*f(x);
 while x < xk do
     begin
     x:= x + h1;
     y1:= y2;
     y2:= y1 + h1*f(x);
     end;
     square:=y2;
end;

function tf.simpson:real;
begin {расчет по методу Симпсона}
 x:= xn; y1:= f(x); y2:= y1 + (f(x) + 4*f(x + h1/2) + f(x + h1) ) / 6;
 while x < xk do
     begin
     x:= x + h1;
     y1:= y2;
     y2:= y1 + h1 * ( f(x) + 4*f(x + h1/2) + f(x + h1) ) / 6;
     end;
     simpson:=y2;
end;
Destructor tf.Done; {уничтожение объекта}
begin
clrscr;
end;

var smp:^tf; {указатель на динамический объект}
     h,y : real;
BEGIN
 clrscr;
 write (' введите h (приращение аргумента) '); read (h);
 new(smp,init(h)); {Выделение памяти для объекта в динамической области и инициализация самого объекта с помощью вызова его конструктора}
writeln (' значение интеграла, рассчитанное по методу прямоугольников, равно ');
 writeln (smp^.square);
 writeln (' значение интеграла, рассчитанное по методу Симпсона, равно ');
 writeln (smp^.simpson);
```

```
    readkey;
    dispose(smp,done);
END.
```

6 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

И н т е г р и р о в а н н а я с р е д а р а з р а б о т к и п р о г р а м м

Интегрированная среда является частью системы разработки программ на языках СИ и Паскаль. Borland C++ и Borland Pascal имеют разные, но очень похожие среды программирования.

Основным достоинством среды является интеграция необходимых средств разработки программ в единую оболочку. Не выходя из которой, программист имеет возможность создавать, компилировать, выполнять, отлаживать и корректировать программы.

Работа пользователя базируется на использовании техники меню и окон. Здесь также имеется контекстно-зависимая помощь. Основные оперативные клавиши среды программирования приведены в табл. 24.

Экран состоит из четырех частей: главное меню, окно редактора, окно сообщения и строки оперативной подсказки.

24 ОПЕРАТИВНЫЕ КЛАВИШИ СРЕДЫ ПРОГРАММИРОВАНИЯ

Клавиши	Назначение
F2	Сохраняет файл, находящийся в данный момент в редакторе.
F3	Загрузка файла (появляется запрос имени файла).
F5	Наложение активных окон друг на друга.
F6	Смена активного окна.
F7	Выполнение очередного оператора программы.
F8	Переход к следующей ошибке.
F9	Создает выполняемый файл, но не загружает его.
F10	Вызов главного меню.
ALT F1	Вызывает предыдущий экран подсказки.
ALT F2	Закрытие активного окна .
ALT F9	Транслирует с расширением *.OBJ в СИ и с расширением *.TPU в Паскале файл, загруженный в редактор.
ALT X	Выход из среды.
CTRL F 1	Вызывает подсказку о текущей позиции курсора
CTRL F 2	Завершение отладки программы.
CTRL F 3	Вывод списка активных блоков.
CTRL F4	Просмотр значения выражения.
CTRL F5	Изменение размера и положения активного окна
CTRL F6	Добавление выражения в окно наблюдений.
CTRL F8	Установка или отмена точки останова.
CTRL F9	Запуск программы.
Shift Del	Удаление выделенного текста из файла в карман
Shift Ins	Помещение выделенного текста из кармана в файл.

Г Л А В Н О Е И О С Н О В Н Ы Е В Ы П А Д А Ю Щ И Е М Е Н Ю

FILE – служит для работы с файлами (загрузка, сохранение, выбор, создание, запись на диск), а так же работа с директориями (изменение директории, просмотр директории, выход из программы, и временный выход в DOS).

EDIT – это создание и редактирование исходных файлов.

SEARCH – поиск текста, процедуры, функции или места ошибки.

RUN – автоматическая трансляция, компоновка, запуск программы.

COMPILE – трансляция программы в объектный и исполняемый файлы.

DEBUG – позволяет отслеживать ошибки и работать с сообщениями об ошибках.

TOOLS – вызов вспомогательных программ (утилит)

OPTIONS – позволяет выбрать параметры компилятора, такие как модель памяти, параметры компиляции, диагностику программы, параметры компоновщика, определить макросы, записать директории: выходных, библиотечных и включаемых файлов; сохранять параметр компилятора, загружать параметры из файла конфигурации.

WINDOW – работа с окнами.

HELP – обращение к справочной службе.

DIRECTORIES – коротко его назначение можно описать фразой '' что, где, искать и куда записывать''. Для задания требуемой директории требуется подвести курсор к нужной позиции в меню и нажать Enter. Среда выдает окно ввода нового значения, старое значение будет написано в этом же окне и его можно отредактировать.

Для СИ:

PROJECT – позволяет указать файлы, составляющие программу и управлять проектом ее создания.

INCLUDE DIRECTORIES (C:\TC\INCLUDE)

Для include файлов, чьи имена заключены в двойные кавычки, просмотр осуществляется в текущей директории.

LIBRARY – это библиотечные директории. В указанных здесь директориях редактор внешних связей ищет библиотеки объектных программ.

OUTPUT – это выходной директорий, в который записывается создаваемые компилятором и редактором связей объектные файлы с расширением *.OBJ и *.EXE.

ТС ДИРЕКТОРИИ

TCHELP.TCH – файл с комплексно-зависимой подсказкой.

PICK FILE NAME – здесь указывается имя PICK-файла (TCPICK.TCP). В этом файле указывается (находится) информация о файле, редактировавшегося в момент выхода из среды.

ARGUMENTS – средства задания параметров программы, которые будут переданы в качестве аргументов функции main из командной строки.

SAVE OPTIONS – сохранение опций.

КОМАНДЫ РЕДАКТОРА СРЕДЫ

Максимальная длина строки в редакторе – 248 (видимых – 77).

УДАЛЕНИЕ СТРОКИ В РЕДАКТОРЕ: CTRL Y.

УДАЛЕНИЕ СЛОВА: CTRL T.

Работа с блоками

Ctrl K B – начало блока.
Ctrl K K – конец блока.
Ctrl K H – отмена выделения блока.
Ctrl K Y – удалить блок.
Ctrl Del – удалить блок.
Ctrl K W – записать блок на диск.
Ctrl K R – считать с диска.
Ctrl K C – скопировать блок.
Ctrl K V – переместить блок.
Shift Del – взять блок в карман.
Shift Insert – взять блок из кармана.

Методы отладки программ

Иногда, когда программы делают что-то непредвиденное и нельзя достаточно быстро определить причину этого, то в этом случае лучше всего остановить программу в заданной точке и просмотреть ее работу по шагам, при этом просматривая значения переменных.

Выполнение программы по шагам и трассировка

1 STEP OVER – команда выполнения по шагам
2 TRACE INFO – команда трассировки.
3 Остановка выполнения программы – GO TO курсор.

Эти команды находятся в меню RUN, однако, можно обойтись и без них используя следующие комбинации клавиш:

Ctrl F8 – отметить строку программы с которой необходимо трассировать;
Ctrl F7 – просмотреть, задаваемые значения переменных в окне watch;
Ctrl F9 – запустить программу на исполнение;
F7 – выполнение трассировки;
Ctrl F2 – прервать трассировку;
Ctrl F4 – просмотреть значение переменной.

Разница между трассировкой и просмотра программы по шагам заключается в следующем: при выполнении программы по шагам вызовы функции интерпретируются, как простой оператор, и после завершения вашей подпрограммы возвращается следующий оператор, а трассировка загружает код этой подпрограммы и продолжает ее построчное выполнение.

Технология программирования

Создание программы для ЭВМ – сложный и трудоемкий процесс. Он включает в себя следующие основные этапы: осознание проблемы и представление задачи на естественном языке; формулировку задачи на языке математических соотношений; разработку численного метода и алгоритма решения задачи на языке математики; разработку структуры данных; проектирование схемы алгоритма обработки данных; запись программы на языке программирования и подготовку исходных данных; разработку системы тестов и отладку программы; решение задачи на ЭВМ; получение и интерпретацию результатов.

Совокупность приемов, направленных на создание безошибочной программы за приемлемое время, составляет технологию программирования. К их числу относятся приемы разработки структуры программы, приемы включения в программу дополнительных операторов, обеспечивающих ускоренное создание программы и как можно более раннее обнаружение ошибок, и приемы, обеспечивающие возможность использования созданной программы как промышленного изделия.

Для каждого из этапов создания и использования программы существуют определенные приемы обеспечения качества программы.

Обеспечение качества программы на этапах разработки алгоритма и записи программы

Высокое качество программы достигается, в первую очередь, за счет глубокой проработки схемы алгоритма на этапе проектирования. Это прежде всего безошибочность программы, уверенность программиста в том, что она не содержит ошибок, и уверенность пользователя в том, что она правильна.

Уверенность в безошибочности программы определяется ясностью и простотой, читаемостью и легкостью интерпретации ее автором и пользователями, поскольку ошибки в программе могут выявляться в процессе ее создания и эксплуатации. Шансы сделать ошибки уменьшаются, если при разработке создатели программы будут стремиться к тому, чтобы она была понятной другим людям.

Хотя при программировании невозможно избежать ошибок, рекомендуется придерживаться некоторых правил составления программы, которые позволяют быстро обнаружить и устранить ошибки. К числу правил хорошего стиля программирования относят следующие.

1) Необходимо стремиться к наиболее полному изучению поставленной задачи при формулировке задачи, т.е. разработке математической модели. Такое изучение позволяет добиться ясной, предусматривающей множество логических взаимодействий объектов программы. Недостаточно глубокая проработка математической модели приводит к неправильным результатам решения задачи, если модель неприменима к тем классам объектов, для расчета параметров которых она используется в данной программе, или если исходные данные не учитывают особенностей данной модели, а модель, в свою очередь, не учитывает всех взаимодействий между данными.

2) Проработка алгоритма решения задачи связана с возможно более полным учетом общих особенностей процесса вычислений на ЭВМ. Так как в ЭВМ не существует никакого другого внутреннего способа представления комплексного числа, кроме представления в виде пары вещественных чисел, то попытка вычислить на ЭВМ корень четной степени из вещественного отрицательного числа вызывает прекращение вычислений и сообщение об ошибке. Алгоритм в подобных случаях должен предусматривать анализ знака значения подкоренного выражения и содержать две возможных ветви вычислений – для вещественного и мнимого значений корня. Другой пример связан с приближенным представлением вещественных чисел в ЭВМ. Нецелесообразно сравнивать значения двух вещественных выражений. Сравнение следует заменить проверкой соотношения

$$|A - B| < \mathrm{e},$$

где e – малое число; если это неравенство выполняется, то следует считать, что $A = B$.

Приближенное представление вещественных чисел в ЭВМ может привести и к тому, что при вычислениях с числами разного порядка могут получиться неправильные результаты. Например, при вычислениях с семью значащими цифрами для $A = 199$ и $B = 0{,}0001$ получено значение $(A + B)^3 - A^3 = 0{,}1*10^2$ вместо числа $0{,}1188*10^2$; более точный результат получен по другой формуле: $B^3 + 3B^2A + 3A^2B$.

Погрешность результата будет меньшей, если начинать суммировать с меньших по величине чисел, а не с больших, как в первом случае.

3) При разработке алгоритма необходимо стремиться к максимальной простоте и понятности. Это относится как к содержательной стороне, так и к форме записи программы на языке программирования. Применение стандартных приемов структурного программирования делает программу более ясной, хотя в некоторых случаях более громоздкой и менее эффективной. Ясность и простота программы важнее, чем выигрыш в эффективности.

Хороший набор стандартов поможет сконцентрировать внимание на новых задачах. Рекомендуется программу сначала записать на каком-либо легко воспринимаемом языке, например на языке схем алгоритмов, и лишь после анализа

Тестирование и отладка программ

При разработке программ наиболее трудоемким является этап отладки и тестирования программ. Цель тестирования, т.е. испытания программы, заключается в выявлении имеющихся в программе ошибок. Цель отладки состоит в выявлении и устранении причин ошибок.

Отладку программы начинают с составления плана тестирования. Такой план должен представлять себе любой программист. Составление плана опирается на понятие об источниках и характере ошибок. Основными источниками ошибок являются недостаточно глубокая проработка математической модели или алгоритма решения задачи; нарушение соответствия между схемой алгоритма или записью его на алгоритмическом языке и программой, записанной на языке программирования; неверное представление исходных данных на программном бланке; невнимательность при наборе программы и исходных данных на клавиатуре устройства ввода.

Нарушение соответствия между детально разработанной записью алгоритма в процессе кодирования программы относится к ошибкам, происходящим вследствие невнимательности программиста. Отключение внимания приводит и ко всем остальным ошибкам, возникающим в процессе подготовки исходных данных и ввода программы в ЭВМ. Ошибки, возникающие вследствие невнимательности, могут иметь непредсказуемые последствия, так как наряду с потерей меток и описаний массивов, дублированием меток, нарушением баланса скобок возможны и такие ошибки, как потеря операторов, замена букв в обозначениях переменных, отсутствие определений начальных значений переменных, нарушения адресации в массивах, сдвиг исходных данных относительно полей значений, определенных спецификациями формата.

Учитывая разнообразие источников ошибок, при составлении плана тестирования классифицируют ошибки на два типа: 1 – синтаксические; 2 – семантические (смысловые).

Синтаксические ошибки – это ошибки в записи конструкций языка программирования (чисел, переменных, функций, выражений, операторов, меток, подпрограмм).

Семантические ошибки – это ошибки, связанные с неправильным содержанием действий и использованием недопустимых значений величин.

Обнаружение большинства синтаксических ошибок автоматизировано в основных системах программирования. Поиск же семантических ошибок гораздо менее формализован; часть их проявляется при исполнении программы в нарушениях процесса автоматических вычислений и индицируется либо выдачей диагностических сообщений рабочей программы, либо отсутствием печати результатов из-за бесконечного повторения одной и той же части программы (зацикливания), либо появлением непредусмотренной формы или содержания печати результатов.

В план тестирования обычно входят следующие этапы:

1 Сравнение программы со схемой алгоритма.

2 Визуальный контроль программы на экране дисплея или визуальное изучение распечатки программы и сравнение ее с оригиналом на программном бланке.

Эти два этапа тестирования способны устранить большое количество ошибок, как синтаксических (что не так важно), так и семантических (что очень важно, так как позволяет исключить их трудоемкий поиск в процессе дальнейшей отладки).

3 Трансляция программы на машинный язык. На этом этапе выявляются синтаксические ошибки. Компиляторы с языка СИ, ПАСКАЛЬ выдают диагностические сообщения о синтаксических ошибках в листинге программы *(листингом* называется выходной документ транслятора, сопровождающий оттранслированную программу на машинном языке – объектный модуль).

4 Редактирование внешних связей и компоновка программы. На этапе редактирования внешних связей программных модулей программа-редактор внешних связей, или компоновщик задач, обнаруживает такие синтаксические ошибки, как несоответствие числа параметров в описании подпрограммы и обращении к ней, вызов несуществующей стандартной программы, например 51 H вместо 51 N. различные длины общего блока памяти в вызывающем и вызываемом модуле и ряд других ошибок.

5 Выполнение программы. После устранения обнаруженных транслятором и редактором внешних связей (компоновщиком задач) синтаксических ошибок переходят к следующему этапу – выполнению программы на ЭВМ на машинном языке: программа загружается в оперативную память, в соответствии с программой вводятся исходные данные и начинается счет. Проявление ошибки в процессе ввода исходных данных или в процессе счета приводит к прерыванию счета и выдаче диагностического сообщения рабочей программы. Проявление ошибки дает повод для выполнения отладочных действий; отсутствие же сообщений об ошибках не означает их отсутствия в программе. План тестирования включает при этом проверку правильности полученных результатов для каких-либо допустимых значений исходных данных.

6 Тестирование программы. Если программа выполняется успешно, желательно завершить ее испытания тестированием при задании исходных данных, принимающих предельные для программы значения, а также выходящие за допустимые пределы значения на входе.

Контрольные примеры (тесты) – это специально подобранные задачи, результаты которых заранее известны или могут быть определены без существенных затрат.

Наиболее простые способы получения тестов:

а) подбор исходных данных, для которых несложно определить результат вычислений вручную или расчетом на микрокалькуляторе;

б) использование результатов, полученных на других ЭВМ или по другим программам;

в) использование знаний о физической природе процесса, параметры которого определяются, о требуемых и возможных свойствах рассчитываемой конструкции. Хотя точное решение задачи заранее неизвестно, суждение о порядке величин позволяет с большой вероятностью оценить достоверность результатов.

П р о е к т и р о в а н и е п р о г р а м м

Разработка программ и программных комплексов должна обеспечивать создание в кратчайшие сроки программных изделий, которые могут использоваться без участия разработчиков. Понятие программного изделия включает в себя как программу, ее текст, представляемый на машинном носителе (магнитных дисках), так и сопровождающую ее документацию. Программное изделие регистрируется в фондах алгоритмов и программ, в функции которых входит размножение копий программ и документации к ним для пользователей.

Основные требования, предъявляемые к качеству программного изделия, – функциональность, надежность, удобство эксплуатации – обеспечиваются за счет правильного проектирования программного комплекса и создания необходимой документации.

Для сложных программ и программных комплексов этап проектирования программы выполняется параллельно с этапом разработки алгоритма и структуры данных. Проектирование предполагает разбиение задачи на несколько подзадач, для решения каждой из которых создается программный модуль. При этом программа проектируется как многомодульная структура. Существует два основных способа проектирования многомодульных программ – восходящее и нисходящее проектирование.

Восходящее проектирование (или проектирование "снизу вверх") основано на выделении нескольких достаточно крупных модулей, реализующих некоторые функции в общей программе. При выделении модулей опираются на доступность реализуемых функций для понимания, простоту структурирования данных, существование готовых программ и модулей для реализации заданных функций, возможности переделки существующих программ для новых целей; имеет значение и размер будущего модуля. Каждый модуль при восходящем проектировании автономно программируется, тестируется и отлаживается. После этого отдельные модули объединяются в подсистемы с помощью управляющего модуля, в котором определяется последовательность вызовов модулей, ввод-вывод и контроль данных и результатов. В свою очередь, подсистемы затем объединяются в более сложные системы и в общий программный комплекс, который подвергается комплексной отладке с проверкой правильности межмодульных связей.

Рассмотренный подход можно рекомендовать при разработке не очень сложных программ. Если размеры подпрограмм невелики, то целесообразно выделить подпрограммы и начать программирование с их составления.

Основные недостатки восходящего проектирования программы проявляются в сложности объединения модулей в единую систему, в трудности выявления и исправления ошибок, допущенных на ранних стадиях разработки модулей. Кроме того, отдельные модули могут создаваться без общего представления о структуре всей системы, что затрудняет их объединение.

Для создания сложных программ можно рекомендовать нисходящее проектирование, основанное на выделении в решаемой задаче иерархии уровней обобщения. Схема иерархии уровней обобщения позволяет программисту сначала сконцентрировать внимание на том, что нужно сделать, и лишь затем – на том, как это сделать.

Ведущая программа записывается как программа верхнего уровня, управляющая вызовами модулей более низкого уровня. Каждый из модулей более низкого уровня, в свою очередь, управляет вызовами модулей еще более низкого уровня. Такой способ проектирования позволяет создавать сложные и громоздкие программы из небольших простых модулей: размер задачи отражается только в числе модулей и уровней обобщений.

При нисходящем проектировании появляется возможность использовать вертикальное управление в схеме иерархии с использованием таких правил: модуль возвращает управление вызвавшему; модуль вызывает только модули более низкого уровня; принятие основных решений возлагается на модули максимально высокого уровня.

После разработки некоторой верхней части схемы иерархии модулей можно составлять, тестировать и отлаживать соответствующие программные модули, причем вместо каждого из модулей при тестировании могут использоваться так называемые "заглушки", т.е. фиктивные модули, содержащие лишь заголовки и операторы возврата. Нисходящий способ проектирования позволяет начать комплексную отладку и тестирование написанной части программной системы, не дожидаясь окончания написания всех модулей. Вставляя в фиктивные модули операторы печати сообщений о входе в имитируемый заглушкой модуль, получают трассировку программы; в заглушку можно поместить операторы, позволяющие выполнять оценку общих затрат машинного времени и памяти ЭВМ.

Основные достоинства нисходящего проектирования:

1) проявление логики программы возникает уже при чтении головного модуля, что делает программу более простой;

2) возможность контроля хода работы над программой в процессе последовательной детализации программы обеспечивает ее непрерывную корректировку; отсутствие комплексной отладки благодаря сквозному контролю позволяет сэкономить до 30 % общего времени разработки программ;

3) одновременная параллельная работа нескольких программистов может оказаться эффективной.

При нисходящем проектировании, однако, возможны и такие ситуации, когда после значительных затрат на программирование выясняется необходимость объединения нескольких подзадач в один модуль, либо обнаруживается невозможность выполнения модулями нижних уровней **своих** функций при заданных временных ограничениях.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИЕТРАТУРЫ

Основная

1 Вычислительная техника в инженерных и экономических расчетах: Учеб. для вузов / Под ред. А. В. Петрова– М.: Высш. шк., 1984. 320 с.

2 Брябрин В. М. Програмное обеспечение персональных ЭВМ. М.: Наука, 1988.

3 Адамс Г., Ли Т., Гейнз У. Управление процессами с помощью вычислительных машин. Моделирование и оптимизация. М.: Советское радио, 1972.

4 Уинер Р. Язык Турбо СИ. М.: Мир, 1991.

5 Егоров Г. А., Кароль В. Л. / Операционная система ОСРВМ СМ ЭВМ М.: Финансы и статистика, 1990.

6 Фигурнов В.Э. IBM PC для пользователя / 6-е изд., перераб. и доп. М.: Финансы и статистика, 1996.

7 Романовская Л. М. Программирование в среде СИ для ПЭВМ. М.: Финансы и статистика, 1991.

8 Страуструп Б. Язык программирования СИ ++: Пер. с англ. М.: Радио и связь, 1991.

9 Епанешников А., Епанешнеков В. Программирование в среде Turbo Pascal 7.0.-М.: Диалог-МИФИ, 1996.

10 Ален И., Голуб Т. СИ и СИ ++ Правила программирования. М.: Бином, 1996.

11 Артемова С. В., Орлова Л. П., Трейгер В. В. Алгоритмические языки и программирование. Тамбов: ТГТУ, 1995.

Дополнительная

1 Брич З. С., Капилевич Д. В. Фортран ЕС ЭВМ. 2-е изд., перераб. и доп. М.: Финансы и статистика, 1985.

2 Гончаров А. Exel-7.0 в примерах. СПб.: Питер, 1996.

3 Рассохин Д. От СИ к СИ ++. М.: Эдель, 1993.

4 Графические средства Турбо СИ и Турбо СИ ++ / Б. П. Прокопьев и др. М.: Финансы и статистика, 1992.

5 Бошкин А. В. Работа в Турбо СИ. М.: Финансы и статистика, 1991.

6 Тондо К., Гимпел С. Язык СИ. Книга ответов. М.: Финансы и статистика, 1994.

7 Семенов Ю. А. Протоколы и ресурсы Internet. М.: Радио и связь, 1996.

8 Джамса К., Коуп К, Программирование для Internet в среде Windows: Пер. с англ. СПб.: Питер, 1996.

СОДЕРЖАНИЕ