А. П. Полищук

ИНФОРМАТИКА

Персональный компьютер и его программирование

(C, C++, Паскаль)

Учебно-справочное пособие

Оглавление

1. Введение.

1.1. Назначение книги.

Книга предназначена для использования в качестве учебно-справочного пособия по информатике при изучении практического прикладного программирования в старших классах средних и на младших курсах высших учебных заведений.

Изучение этого раздела как в вузах, так и в школах связано с известными трудностями - отсутствием современной техники, учебных пособий и справочников в библиотеках учебных заведений, квалифицированных преподавателей, имеющих опыт и практикующих в профессиональном программировании.

При составлении этого пособия ставилась задача собрать в одном блоке базовый набор сведений об архитектуре персонального компьютера и основных его компонентов, операционной системе, основах алгоритмизации, языках программирования и проиллюстрировать это типовыми примерами прикладных программ, использующих ресурсы компьютера и услуги операционной системы.

1.2. Состав пособия.

Автор не пытался "объять необъятное" и ограничился основами процедурного программирования для DOS, отдавая приоритет методу, а не конкретной технике исполнения.

Раздел 2 посвящен формированию в сознании будущего программирующего пользователя "программистской модели" программируемого объекта - компьютера и его основных компонентов, прежде всего - оперативной памяти. Акцентируется внимание на числовом кодировании обрабатываемой информации любого происхождения, приводятся примеры работы с числовыми объектами на модели памяти, формируются основные понятия - битов, байтов, размеров числовых объектов в памяти, их адресов и указателей, основ двоичной арифметики, команд обработки данных в памяти, принципов функционирования центрального процессора и пр.

Раздел 3 содержит описание операционной системы, ее назначения, состава и основных функций, способов предоставления услуг пользователю и прикладным программам.

В разделе 4 изложены основы алгоритмизации, формируются понятия языка программирования, основных типов трансляторов, приводятся описания ряда типовых алгоритмов, которые в последующих разделах реализуются на транслируемых языках высокого уровня.

Раздел 5 содержит краткое описание языка С с элементами С++, справочные данные по основным библиотекам, некоторые методы управления ресурсами компьютера из прикладных программ на С.

Раздел 6 содержит набор примеров прикладного программирования в среде (Турбо, Борланд) С, С++.

В разделе 7 приводится краткое описание языка программирования (Турбо, Борланд) Паскаль, справочные данные по основным библиотечным модулям и некоторые методы управления компьютерными ресурсами из прикладных Паскаль - программ.

Раздел 8 содержит Паскаль - реализацию тех же примеров, которые выполнены в разделе 6 на языках С, С++.

Последний, девятый раздел книги посвящён объектно-ориентированному программированию на языке С++.

"Двуязычие" книги, значительно осложнившее мою работу над ней, объясняется следующим. Я убедился на собственном опыте, что начальное обучение программированию необходимо начинать с языка С, как самого простого, не содержащего абстрактных типов, прямо отражающего принцип числового кодирования всех видов информации и не формирующего у обучаемого ложных понятий о представлении данных, адресов и команд в памяти компьютера (например, о несовместимости объектов типа `char` и `byte` в Паскале). У программирующих на Си Паскаль при необходимости осваивается без усилий, "сам собой", а обратный переход в большинстве случаев затруднен. Язык С++ принят базовым для программирования под Windows, он становится основой для создания новых современных языков, например Ява. Перечень этих аргументов можно продолжать, но не учитывать инерционность системы образования невозможно и для тех, кто не готов принять язык С, приведен и Паскалевский вариант.

1.3. Об общеобразовательной роли компьютерного программирования.

"Сперва хочу вам в долг вменить - на курсы логики ходить. Ваш ум, не тронутый доныне, на них приучат к дисциплине, чтоб взял он направленья ось, не разбредаясь вкривь и вкось".

Во времена Гете и доктора Фауста не было компьютерной альтернативы приведенной директиве, иначе им пришлось бы заменить логику на компьютерное программирование - лучше об общеобразовательной роли этого предмета не скажешь. А необходимость в его защите возникает в связи с периодически публикуемыми предложениями упразднить преподавание программирования в школе, легко сняв проблемы с оснащением, кадрами и пр. К сожалению кандидаты в наши президенты не скоро будут включать в свои предвыборные программы пункт "Компьютер - на каждую парту" и нам еще долго придется обучать "заочному" программированию мелом на доске и карандашом на бумаге.

Но даже в этом варианте изучение методов программирования компьютеров оказывает большое влияние на уровень общего развития ученика и прежде всего на развитие его организаторских способностей. Объясняется это тем, что компьютерное программирование требует умения получить доступ к хранилищу исходных данных и определить их размер, выделить необходимый для размещения этих данных объем оперативной памяти, загрузить исходные данные в полученное рабочее пространство, выделить место для хранения промежуточных и конечных результатов обработки данных, выполнить обработку данных, "упаковать" результат в удобную для пользователя форму и выдать (отобразить) этот результат.

Даже эта сильно упрощенная схема вызывает ассоциации со многими областями человеческой деятельности, как будто совсем не связанными с программированием. Что бы мы ни собрались произвести (написать книгу или построить дом), нам придется решить задачи получения и складирования исходных материалов, разработки пооперационного плана выполнения работ, оформления внешнего вида изделия для передачи его заказчику. Таким образом, составление компьютерных программ приучает нас планировать свою деятельность, выстраивать работы в логически обоснованную сеть операций, то есть прививает организаторские навыки, учит понимать влияние организации труда на его эффективность. Каждый грамотный человек должен понимать, что плохая организация труда в состоянии погубить результаты сколь угодно высокого индивидуального мастерства исполнения отдельных операций.

При составлении программы программист работает в некоторой инструментальной среде и при разбиении решения общей задачи на отдельные операции вынужден учитывать возможности этой среды - ассортимент услуг операционной системы, языкового транслятора, наличие необходимых процедур в библиотеках. Эта всегда присутствующая сторона

труда программиста дает навыки в разбиении процесса решения общей задачи на комплексы операций, доступных для исполнения имеющемуся в наличии набору исполнителей или в подборе исполнителей для имеющихся задач, если такая свобода выбора предоставлена.

1.4. О методологии преподавания программирования.

Настоящее пособие ориентировано не только на общеобразовательные, но и на профессионально ориентированные в "компьютерном направлении" учебные заведения или группы. Поэтому изложение материала начинается не с традиционного для большинства учебников изучения основ алгоритмизации и языков программирования, а с элементарных сведений об устройстве компьютера, его компонентов и операционной системы, то есть с изучения программируемого объекта. По опыту автора только понимание устройства и принципов работы компьютера и используемой операционной системы делает программирование полностью осознанным процессом и способствует быстрому его освоению, так как в этом случае составитель программы способен моделировать в своем сознании те преобразования, которым подвергаются находящиеся в оперативной памяти объекты вносимыми в текст программы операторами. У программиста, обученного на чисто "лингвистическом" базисе возникает слишком много "загадок" в поведении составленной им программы и "потолок" его возможностей по рациональному использованию компьютерных ресурсов остается достаточно низким.

Любая программа содержит две основные части - прикладное ядро и ресурсное обеспечение, на которое в школьных курсах программирования обращают в подавляющем большинстве случаев слишком мало внимания. Но прикладных задач в программировании - несчетное множество и методы их решения не поддаются унификации. В то же время методы управления компьютерными ресурсами являются общими для всех приложений и их осознанное использование способствует более рациональному подходу к решению всех видов прикладных задач.

2. Компьютер, который мы программируем.

2.1. Что умеет обрабатывать электронный компьютер?

Электронный компьютер - устройство (машина) для обработки чисел, одних только чисел и ничего, кроме чисел! Недаром другое его название - ЭЦВМ (Электронная цифровая вычислительная машина). Поэтому, когда вы слышите утверждения о том, что компьютер - средство для обработки информации любой природы, всегда имейте в виду опускаемое при этом дополнение - "если эта информация предварительно преобразована в числовую форму, представлена в виде последовательности чисел, закодирована в числовых кодах".

Так, например, можно вводить в компьютер музыку и другую аудиоинформацию, если предварительно преобразовать ее в изменяющийся во времени электрический ток или напряжение, а затем с помощью специального устройства - аналого-цифрового преобразователя - в последовательность чисел, отражающих величину электрического сигнала в различные моменты времени. При выводе из компьютера последовательность чисел преобразуется перед подачей на динамический громкоговоритель в непрерывный электрический сигнал с помощью другого устройства - цифро-аналогового преобразователя.

Рисунки, картины, чертежи и иная видеоинформация может быть введена в компьютер, например, с помощью специального устройства - сканера (разновидность телевизионной камеры), который преобразует цвет каждого маленького элемента изображения (пиксела) в число (код цвета). При отображении видеоинформации на экране дисплея эти числа переписываются в видеопамять, а устройство управления видеосистемой (видеоконтроллер) берет их оттуда и преобразует, например, в сигналы управления лучом, "обстреливающим" экран монитора.

Произвольные тексты могут вводиться в компьютер, если все используемые в них символы представлены в виде чисел, то есть "перенумерованы", а каждый номер является ссылкой на последовательность чисел, описывающих рисунок соответствующего символа. При выводе на экран коды символов и их атрибутов (цветов фона и рисунка) записываются в видеопамять, видеоконтроллер берет оттуда код символа, отыскивает по этому номеру последовательность чисел,

кодирующих рисунок символа, и выполняет отрисовку в соответствии с кодом атрибута.

Таким образом, в компьютере унифицирован метод представления информации любого происхождения в процессах обмена, хранения и обработки - в числах; природа информации "проявляется" только при ее отображении на внешних устройствах. Одно и то же число может быть отображено в виде последовательности цифр, буквы, цвета точки на экране или бумаги, частоты или продолжительности звука. Сам способ отображения тоже кодируется при этом в числовой форме.

2.2. Кое-что из компьютерной истории.

Итак, компьютер умеет обрабатывать только числа. Так уж сложилось исторически. По-видимому, количественная мера вещей играла важнейшую роль в жизни и самоутверждении древнего человека, так как считать и изобретать инструменты для облегчения счета он начал значительно раньше, чем изобрел алфавит. При этом инструментальный счет опережал в развитии письменный - пергамент предстояло изобрести только в 5-м веке до н.э., бумагу-в 11-м нашей, а изобретение первого компьютера - Абака, представлявшего собой размеченную на колонки доску с размещенными по позиционному принципу морскими камешками, относят к 4-му веку до н.э. Развитие торговли, строительства, ремесел, промышленного производства, военного дела непрерывно увеличивало потребность людей в вычислениях, сами вычислительные задачи становились все более сложными. Люди, которым в силу различных причин приходилось выполнять нелегкую вычислительную работу, стремились облегчить ее созданием вычислительных устройств. Сегодня мы называем эти устройства калькуляторами, вычислителями, компьютерами (computer - вычислитель). Где-то на рубеже 16-17 веков появился русский абак -счеты.

Появление и распространение в 12-13 веках бумаги вызвало развитие письменного счета, основанного на использовании арабско-индийских цифр.

Вычислительные трудности были существенно снижены изобретением в начале 17 века логарифмов (Джон Непер опубликовал первую таблицу логарифмов в 1614 г). Это позволило решать задачи на умножение и сложение с помощью сложения и вычитания. Изобретение логарифмов стимулировало работы по механизации вычислений - вначале появились логарифмические шкалы, облегчавшие вычисления, а затем различные конструкции логарифмических линеек, в

усовершенствовании которых идеей бегунка принял участие и великий Ньютон.

Первую механическую "арифметическую машину", осуществлявшую 4 арифметических действия в 10-тичной системе счисления, изготовил в 1645 году известный физик и математик Блез Паскаль. В России первая счетная машина создана около 1770 года литовским часовым мастером Якобсоном, затем в 1843 г Слонимским. Среди последующих вариантов следует упомянуть первый в мире арифмометр - арифметическую машину Лейбница, выполнявшую уже все арифметические действия в отличие от предшествующих суммирующих машин.

Пионером автоматических вычислений заслуженно считают Чарльза Бэббиджа, создателя первой универсальной вычислительной машины с автоматизированным процессом вычислений - прообраз современных компьютеров. Предложенная им в 1819 году машина предназначалась для вычисления значений многочленов по способу разностей. Вмешательство человека в процесс вычислений в машине Бэббиджа не требовалось. По-видимому, первым программистом следует считать Аду Байрон-Лавлейс (дочь знаменитого поэта), составившую для машины Бэббиджа алгоритм вычисления чисел Бернулли. Все упоминавшиеся нами машины были чисто механическими.

Первой машиной, в которой для управления процессом вычислений использовались электромеханические реле, была "Вычислительная машина с автоматическим управлением последовательностью операций" по имени "Марк-1". Ее создание финансировала известная теперь фирма-производитель компьютеров IBM (1944 год, США). Для представления чисел в ней по-прежнему использовались механические элементы, а управляющие работой машины команды вводились с перфорированной ленты с помощью контактных щеток. Автор разработки - математик и инженер Айкен. Примерно в это же время были созданы ряд "чисто релейных" машин - уже с двоично-десятичным (1941 г., Цузе, Германия) или с двоично-пятеричным (Штибитц), или с чисто двоичным (1954, Бессонов, СССР) представлением чисел.

Век релейных машин был очень недолгим - в 1943 году в США под руководством ученых Мочли и Эккерта началась работа над созданием первой электронной вычислительной машины на электронных лампах. В 1945 г. ими была создана предназначенная для военных целей ламповая машина ЭНИАК - огромное сооружение на 40 панелях, содержащих 18000 ламп и 1500 реле, потреблявшее 150 квт электроэнергии. ЭНИАК

тратил на сложение 0.0002 сек, а на умножение 0.0028 сек - скорость, недоступная его механическим и электромеханическим предшественникам.

Пионером в программировании на ЭНИАКе по праву считается Грейс Хоппер; именно ей мы обязаны таким термином, как отладка - debugging (при работе одной из программ виновником сбоя оказалась бабочка, попавшая во внутренности ЭНИАКа, процесс извлечения которой и был первым debug'ом - удалением жучка).

После того, как известный ученый фон Нейман с коллегами предложили принцип "хранимой программы", в Англии в 1950г была создана первая в мире машина с хранимой программой - машина совершенно нового типа. После нее настроили много ламповых машин, в том числе МЭСМ (Лебедев, Киев), БЭСМ (Лебедев, Москва) и, наконец, семейство первых в СССР серийных машин "Урал".

В середине 50-х годов на смену ламповым машинам пришли ЭВМ на полупроводниковых транзисторах, более быстродействующие, надежные, экономичные машины второго поколения.

Развитие третьего поколения компьютеров тесно связано с разработкой интегральной технологии, позволившей объединить в одном элементе несколько транзисторов - это были машины на малых интегральных схемах (МИС).

Затем развитие интегральной технологии привело к созданию микросхем средней интеграции, содержащих уже десятки транзисторов, и к разработке вычислительных машин 4-го поколения.

5-е поколение - это использование больших интегральных схем, приведших компьютерную технику и связанные с ней технологии обработки информации на столе у каждого желающего.

Первая электронная машина использовалась в метеорологии, военном деле и для некоторых научно-технических расчетов. 20 лет назад насчитывалось уже 2500 областей применения ЭВМ, а сегодня можно сказать, что области деятельности, обходящиеся без машин, остались в меньшинстве - это в основном ручной труд. Но так как сегодня только 1% всей полезной работы на Земле выполняется вручную, то сферы применения ЭВМ уже неперечислимы.

Таким образом, к тому периоду, когда общество ощутило потребность в быстродействующих средствах приема, передачи, хранения, обработки и отображения больших объемов текстовой, графической, звуковой и других видов информации, в его распоряжении уже были такие средства для

числовой информации. Оставалось использовать их возможности с предварительным преобразованием (кодированием) нечисловых типов в числовую форму.

2.3. Электронная память компьютера, центральный процессор и операционная система.

2.3.1. Электронная память - "технологическое пространство" для обработки представленных в числовой форме данных.

Для выполнения над числами любых операций их необходимо где-то записать (запомнить), а затем извлекать, выполнять над ними некоторые операции, куда-то записывать результат - другими словами нужна оперативная быстродействующая память. Электрический способ передачи сигналов по скорости уступает только световому, поэтому для размещения чисел на период их обработки в электронных компьютерах используют электронную память.

2.3.2. Биты памяти как отражение цифр двоичных чисел.

Так как числа в позиционных системах счисления принято записывать в виде последовательности цифр, то элементарной, самой маленькой единицей памяти должно быть техническое устройство для записи всех возможных значений числа, состоящего из одной цифры. В десятичной системе счисления таких значений 10 (0, 1,..., 9). При электрическом способе реализации пришлось бы различать 10 уровней тока или напряжения, что не только громоздко, но и ненадежно, так как колебания питающего напряжения, температурные изменения параметров элементов электрических цепей и другие случайные факторы могут приводить к "взаимному перекрытию" уровней напряжения и искажать записываемые в память значения чисел в каждой из цифр.

Поэтому в компьютерах используют для представления чисел самую простую из позиционных систем счисления - двоичную, в которой каждая позиция числа может принимать не 10, а только 2 значения - 0 или 1. Техническое представление одноразрядного 2-ичного числа значительно проще - оно может

моделироваться на качественном уровне - "высокий" или "низкий" уровень напряжения.

Каждую позицию 2-ичного числа называют **БИТ ЧИСЛА** (от binary digit - двоичная цифра), а ее техническую реализацию - **БИТ ПАМЯТИ**. Бит электронной памяти представлен **ТРИГГЕРОМ** - транзисторной схемой, в которой при подаче сигнала на управляющий электрод транзистор переводится из проводящего состояния в непроводящее или наоборот, а на другом его электроде - коллекторе - напряжение меняется с высокого уровня на низкий или наоборот. Точное значение напряжения при этом не играет роли и изменение значения позиции числа осуществляется достаточно просто. Договорившись считать высокий уровень значением 1, а низкий - значением 0, мы получаем электронную модель цифры 2-ичного числа.

Таким образом, память компьютера - это цепочка битов памяти со значениями 0 или 1; при подаче питания эти биты устанавливаются в случайные значения:

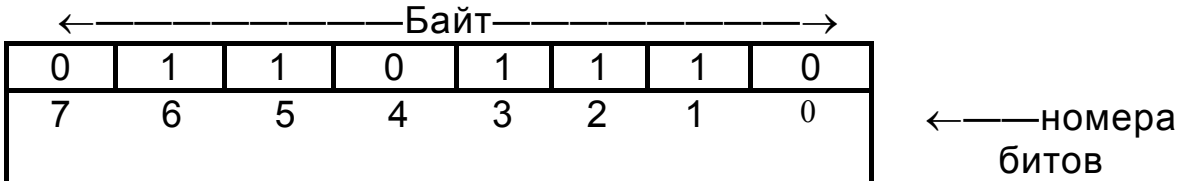
Цепочка битов...

0	1	1	1	0	0	1	0	1	1	0	....	1	1	0	0	0	0	1	1	0	1	0	1	1

2.3.3. Байт - двоичное число минимальной длины в памяти компьютера, для которого можно определить его местонахождение (адрес).

В письменном счете мы не озабочены количеством цифр в числах, которые складываем или перемножаем, а в технических устройствах приходится иметь дело с числами ограниченной длины, иначе будет трудно определять в цепочке битов, где начинается и заканчивается то или иное число. Место в памяти для размещения числа считается определенным, если известно его расстояние от начала памяти (адрес) и длина размещаемого числа. Для обеспечения возможности адресации размещаемых в памяти чисел пришлось определить размер самого короткого числа, к которому будет возможно адресоваться. Длину минимального адресуемого в памяти числа определили так, чтобы можно было адресоваться в памяти к числовым кодам каждой буквы размещаемого в памяти текста. Так как для числового кодирования печатных символов и кодов управления печатью (возврат каретки, перевод строки, конец страницы и пр.) оказалось достаточно 256 комбинаций единиц и нулей и это количество кодов обеспечивается 8-битовым 2-ичным числом ($2^8 = 256$), то и наименьшее по длине

адресуемое в памяти число определили как 8-мибитовое и назвали эту 8-битовую группу **БАЙТОМ** (`byte`).



Как и в десятичной системе счисления, позиции цифр (битов) в двоичном числе нумеруются справа налево и числовое значение цифры определяется значением цифры, умноженном на его "вес" (основание системы счисления в степени, равной номеру ее позиции). Введя для краткости записи значок '*' для операции умножения и '^' для возведения в степень, получим для нашего примера:

бит с номером 3 имеет значение 1*(2 ^ 3) т.е. 8.

бит с номером 4 имеет значение 0*(2 ^ 4) т.е. 0.

Значение всего вышеприведенного 1-байтового числа:

0*1+1*2+1*4+1*8+0*16+1*32+1*64+0*128=110.

Множество целых чисел делится на подмножества знаковых (`signed`) и беззнаковых (`unsigned`, натуральных). Если в памяти отводится место для числа, значение которого может быть как положительным, так и отрицательным, то знак числа тоже надо закодировать и для кода знака отводят бит с самым старшим номером (в нашем примере - 7-й). Число считают положительным, если значение этого бита 0, и отрицательным, если его значение 1.

Другой особенностью представления отрицательных чисел является их хранение в так называемом дополнительном коде - это связано с желанием заменить операцию вычитания операцией сложения (из-за чисто технических трудностей "заёма" из старших разрядов при вычитании). Каким числом мы должны заменить вычитаемое, чтобы результат оказался верным после сложения? Это число равно дополнению вычитаемого до переполнения разрядной сетки (будем помнить об ограниченной длине чисел в памяти компьютера) и называется его

АРИФМЕТИЧЕСКИМ ДОПОЛНЕНИЕМ.

Пусть нужно выполнить 5643 - 341 = ? (Должно быть 5302) в устройстве с 5-разрядной числовой сеткой. Задача перепишется в виде 05643 - 00341 = ?. Дополнением числа 341 до переполнения будет 100000 - 341 = 99659. Теперь сложим 05643 + 99659 = (1) 05302. Взятая в скобки 1 "выпала" в отсутствующий старший разряд и мы получили правильный

результат вычитания, выполненного сложением с арифметическим дополнением вычитаемого.

В двоичной арифметике задача определения арифметического дополнения решается чрезвычайно просто - для представления отрицательного числа необходимо инвертировать все позиции (биты) двоичного числа (заменить единицы нулями, а нули единицами) и прибавить единицу.

В самом деле, рассмотрим пример:

Число 65 в двоичном формате при 8-битовой сетке: 01000001
Инвертируем: 10111110
Добавляем 1: результат минус 65 10111111

Если сложить весовые значения единичных битов в последнем числе, то 65, конечно, не получится. Для определения абсолютного значения отрицательного числа необходимо все повторить: инвертировать все биты и прибавить 1:

Двоичное значение: 10111111
Инвертируем: 01000000
Плюс 1 для получения 65: 01000001

Сумма +65 и -65 должна давать 0:

```
  01000001
  10111111
  ————————
(1) 00000000
```

В скобках показана выпавшая в несуществующий разряд единица.

Для лучшего понимания обсудите вопрос: какое число надо прибавить к 00000001, чтобы получить 00000000? (помните о фиксированном количестве числовых позиций в технических устройствах).

```
  00000001
  11111111
  ————————
(1) 00000000
```

ЗАПОМНИТЕ ПРОСТОЙ ФАКТ: ВЕЛИЧИНУ ОТРИЦАТЕЛЬНЫХ ЧИСЕЛ ОПРЕДЕЛЯЮТ НУЛИ ТАК ЖЕ КАК ВЕЛИЧИНУ ПОЛОЖИТЕЛЬНЫХ - ЕДИНИЦЫ! СЛОЖИТЕ НУЛЕВЫЕ БИТЫ И НЕ ЗАБУДЬТЕ ПРИБАВИТЬ 1!

Очевидно, что наибольшие значения положительных и абсолютных значений отрицательных чисел одной и той же

длины различны из-за "расходуемого" на хранение кода знака одного бита для отрицательных чисел. Таким образом, однобайтовое знаковое может иметь наибольшее абсолютное значение 2*(2 ^ 7)-1 т.е. 127, а беззнаковое 2*(2 ^ 8)-1 т.е. 255.

Длины чисел больше байта определены кратными байту, они могут быть 2-хбайтовыми, 4-хбайтовыми и т.д. Вещественные числа всегда знаковые и они представляются в памяти компьютера как отражение так называемого экспоненциального формата записи, при котором значение числа записывается в виде произведения числа-мантиссы, умноженной на основание системы счисления в степени числа-порядка. Например, в 10-тичной системе число 123.456 можно записать как 1.23456*10^2. Здесь 1.23456 - мантисса, 10 - основание системы счисления, 2 - порядок числа.

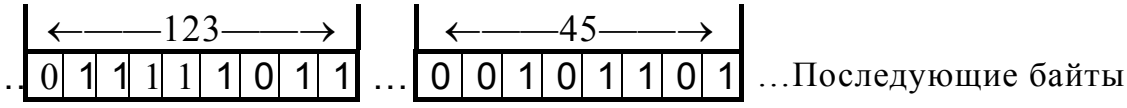
Для размещения вещественных чисел в памяти компьютера используют фрагменты памяти длиной в 4, 6, 8, 10 байтов. Формат хранения следующий (З - знак, +-порядок - диапазон порядка):

Мантисса и порядок в памяти компьютера, естественно, двоичные.

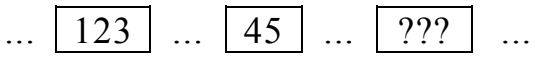
2.3.4. Поработаем с числами в памяти.

Пусть в памяти были выделены 2 однобайтовых участка и в них занесены некоторые значения, например, 123 и 45. Пусть требуется поменять местами значения в этих участках памяти - это стандартная операция при упорядочении (сортировке) числовых последовательностей (массивов) для упрощения последующего поиска нужных значений.

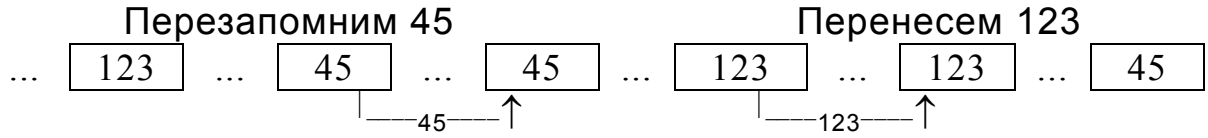
Предшествующие байты...

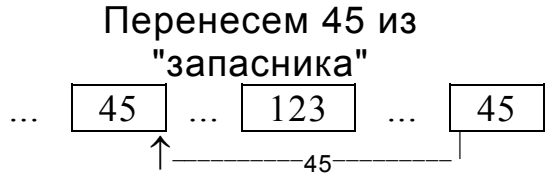


Для упрощения записи мы будем в дальнейшем весь байт изображать одним прямоугольником с записанным внутри значением в 10-тичной системе счисления, хотя помним, что фактически это 8-мибитовая группа с значениями 2-ичных цифр в каждом бите. Если мы просто перешлем значение 45 из правого байта памяти в левый, то значение 123 будет стерто и в обеих областях памяти будет записано по числу 45. Поэтому для замены значений нам понадобится выделить дополнительный участок памяти для временного сохранения одного из чисел и последовательность операций будет выглядеть так:



Выделили байт для промежуточного сохранения одного из чисел





Поставленная задача выполнена.

Обобщим метод решения, введя для дальнейшего упрощения записей имена для выделенных в памяти участков, (например участки A,B) и обозначение '=' для операции занесения в выделенный участок некоторого значения (например A=8); при этом запись вида A=B будет означать "скопировать содержимое области памяти по имени B в область памяти по имени A". Отметим попутно, что именованные участки памяти, предназначенные для хранения информации, в программировании называют "переменными".

Пусть надо поменять значения, записанные в одинаковых по размеру участках памяти (переменных) A и B.

- Выделим участок C такого же размера, как A и B (определим переменную C);
- Перепишем содержимое B в C: C=B;
- Перепишем содержимое A в B: B=A;
- Перепишем содержимое C в A: A=C;

Теперь рассмотрим реализацию решения другой задачи, известной как метод Евклида определения наибольшего общего делителя (НОД) 2-х чисел, размещенных в нашем случае в участках памяти A и B. Решим задачу:

1) Если значение в A равно B, то любое из значений есть искомый НОД и решение задачи прекращается.

2) Если A<B, то мы поменяем значения местами уже рассмотренным методом, чтобы было A>B.

... | A | ... | B | ... | R | ... (A>B)

3) В R поместим разность A-B

... | A | ... | B | ... | R=A-B | ...

4) Если R>B то перепишем его в A (A=R), иначе в A перепишем B, а в B перепишем R (A=B, B=R).

5) Перейдем к выполнению п.1).

Уже из рассмотрения этих простых задач мы ощущаем необходимость в целом ряде команд для выполнения простейших операций с числами в памяти - выделения памяти для размещения значений чисел, записи значений в выделенные участки памяти, сравнения значений в различных участках памяти, выполнения арифметических операций с числами в памяти и т. д.

2.3.7. Шестнадцатеричное представление 2-ичных чисел.

Писать (или набирать на клавиатуре компьютера) длинные цепочки единиц и нулей при задании чисел в 2-ичном формате довольно скучная работа. Так же неудобно просматривать содержимое памяти компьютера, представленное в 2-ичном формате. Поэтому разработан своего рода "стенографический" метод представления двоичных данных - каждый 8-мибитовый байт разбивается пополам и каждая 4-битовая его половина записывается в 16-ричной системе счисления. Для ее осуществления цифровой алфавит 10-тичной системы счисления дополнили шестью цифрами A, B, C, D, E, F, договорившись, что:

10 это A, 11 это B, 12 это C, 13 это D, 14 это E, 15 это F

Например, десятичное число 42936 в 2-ичном формате имеет вид

1010011110111000

После записи полубайтов 1010 0111 1011 1000 16-ричными цифрами

A 7 B 8

получаем компактную запись - A7B8.

Небольшая тренировка позволяет оценить удобство этого метода записи, повсеместно используемого в программировании.

Перевод 16-ричного формата в 10-тичный осуществляют так: все цифры, начиная с самой левой, умножаются на 16 и складываются со следующей цифрой, результат умножается на 16, складывается со следующей цифрой и т д:

A(10) * 16 = 160 + 7 = 167 * 16 = 2672 + B(11) = 2683 * 16 =42 928 +8 = 42936

Преобразование 10-тичного числа 42936 в 16-ричный формат осуществляют последовательным делением на 16 и использованием получаемых остатков в качестве цифр 16-ричного формата справа налево:

42936 / 16 = 2683 и 8 в остатке - это младшая 16-ричная цифра

2683 / 16 = 167 и 11(B) в остатке -это следующая влево цифра

167 / 16 = 10 и 7 в остатке

10 /16 =0

и 10(A) в остатке

получаем A7B8.

Следует помнить, что прибавление 1 к 16-ричному числу F даст 16-ричное 10, 16-ричное 20 эквивалентно 10-ному 32, 16-ричное 100 - 10-ному 256, 16-ричное 1000 - 10-ному 4096. Чтобы отличать 16-ричные числа, после них ставят, например, символ H:

25H это десятичное 37

или используют другие обозначения (например, в языке Паскаль 16-ричное число предваряют знаком $ - $25, а в языке Си парой символов 0x - 0x25).

2.3.8. Числовое кодирование текстовых символов.

Для кодирования текстовых символов отводится 8-битовая группа - байт, позволяющий записать 2^8=256 различных кодов. Это уже кажется избыточным, но только кажется. Обычно в

текстах могут встречаться символы различных алфавитов, например кириллицы и латинского; добавьте сюда 10 цифровых символов, все знаки препинания, различные скобки, знаки арифметических операций, необходимость строчных и заглавных букв, знаки кавычек и процентов, специальные символы типа ~ или $,#,@,& , символы для рисования таблиц - оказывается, наша потребность в различных текстовых символах достаточно велика. Кроме всего перечисленного при выводе текстов на устройства отображения (печатающие устройства, экран дисплея) текстов нам нужен целый набор управляющих кодов, например, перевода строки, возврата каретки, горизонтальной и вертикальной табуляции, подачи звукового сигнала, прогона страницы и пр. Эти коды назвали кодами "неотображаемых символов" и в американском национальном стандарте символьной информации ASCII (*American Standard Code Interchange Information*), используемом во всех странах, отвели для них первые 32 номера от 0 до 31. Этот стандарт охватывает в целом только 128 символов - половину из 256 возможных в пределах байта кодов, вторая половина не стандартизована и именно в ней размещаются символы национальных алфавитов, в том числе наша родная кириллица. В приведенной ниже таблице ASCII - кодов код символа в 16-ричном представлении - это сумма номера строки и номера столбца. Например, код буквы Ф равен 94h или 148 в десятичном выражении.

В приведенной таблице управляющие коды обозначены просто буквой 'У', хотя для удобства и различения они как правило изображаются различными рисунками, например, для кода 10- рисунок §. Участок таблицы с номерами B0-DF, выделенный цветом, содержит символы псевдографики, предназначенные для рисования таблиц. Сами эти символы не показаны, но вы легко можете увидеть их на экране, набрав десятичный код любого из низ на дополнительной цифровой клавиатуре при прижатой клавише Alt.

Собственно "печатные", отображаемые символы начинаются с кода 32 (десятичного) или 20(16-ричного) - кода пробела, вместо изображения которого мы проставили 'ПБ', так как у пробела нет рисунка.

Обратите внимание на то, что цифры от '0' до '9' закодированы числами от 30 до 39 (от 48 до 57 в 10-тичной записи) и усвойте, что цифра - это символ (буква) для обозначения числа, а код цифры не равен обозначаемому цифрой числу! Обозначаемое цифрой число может быть получено вычитанием из кода цифры кода символа '0', так как в

таблице коды всех цифр следуют последовательно друг за другом с "шагом" 1.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	У	У	У	У	У	У	У	У	У	У	У	У	У	У	У	У
10	У	У	У	У	У	У	У	У	У	У	У	У	У	У	У	У
20	П Б	!	«	#	$	%	&	‘	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
90	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
A0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
B0																
C0																
D0																
E0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F0	Ё	ё	Є	є	Ї	ї	Ў	ў	°	•	·	v	№	¤	¦	

2.3.9. Кодирование изображений текстовых символов.

Чтобы рисунок буквы был виден на экране, его цвет должен отличаться от цвета фона, на котором он изображается. Поэтому рассмотренные нами коды символов (порядковые номера в таблице кодирования) необходимо дополнить кодами цвета фона и цвета рисунков. Для этих кодов цветов добавили еще 1 байт памяти и разделили его пополам - младшую (левую) половину из 4-х битов отвели для кодирования цвета рисунка, а старшую для кодирования цвета фона. Этот байт назвали **БАЙТОМ АТРИБУТОВ СИМВОЛА** и он всегда присутствует вместе с кодом самого символа в 2-х байтовых кодах символов, передаваемых в видеопамять для отображения на экране.

В 4-х битах можно закодировать 16 цветов, а при необходимости кодирования большего количества цветов применяют многоступенчатую систему кодирования. Содержимое байта атрибутов удобно записывать в 16-ричном формате, у которого первая 16-ричная цифра в этом случае обозначает цвет фона, а вторая - цвет рисунка символа. Например, 16-ричное число 4E кодирует желтые (код желтого

цвета E или 14 в 10-й системе) буквы на красном (код красного цвета равен 4) фоне.

2-хбайтовые кодовые группы каждой буквы текста, содержащие код символа и код атрибутов его изображения для вывода на экран, записываются в память устройства управления, которое называют дисплейным адаптером, а саму память - **ВИДЕОПАМЯТЬЮ**, **ВИДЕОБУФЕРОМ** или **БУФЕРОМ РЕГЕНЕРАЦИИ**.

Последнее название подсказывает нам, что для постоянного обновления изображения на экране из этого буфера с частотой примерно 25 (или более) раз в секунду считываются коды символов и преобразуются в рисунки букв на экране. Чтобы такое преобразование стало возможным, приходится закодировать и разместить в памяти компьютера и сами рисунки букв. Вспомним, что изображение чего бы-то ни было на экране состоит из элементарных изображений - пикселов. Для изображения символов обычно отводится в зависимости от типа видеосистемы от 8 до 16 строк по 8 пикселов в строке. О каждом пикселе в изображении символа дисплейный адаптер должен знать - относится он к фону или рисунку - то есть достаточно одного бита с 2-мя состояниями. Договорились, что если бит содержит 0, то это пиксел фона, а если 1 - то это рисунковый пиксел. В этом случае рисунок буквы "H" из 8-ми строк по 8 пикселов в строке будет закодирован так:

В 2-ичных кодах	В 16-ричной записи	В 10-тичной записи
01000010	42	66
01000010	42	66
01000010	42	66
01111110	7E	126
01111110	7E	126
01000010	42	66
01000010	42	66
01000010	42	66

256 таких 8-мибайтовых (или 12-тибайтовых, или 14-байтовых или 16-байтовых) кодовых групп хранятся в памяти для рисунков всех изображаемых символов и вся эта область памяти называется **БУФЕРОМ ЗНАКОГЕНЕРАТОРА**. Адаптер дисплея "узнает" начальный адрес этого буфера (порядковый номер его начального байта, отсчитанный от начала памяти), берет из видеопамяти код символа, означающий порядковый номер его кодовой группы в буфере знакогенератора, умножает на число пиксельных строк в изображении символа и прибавляет полученное число к начальному адресу буфера

знакогенератора. Полученное число есть начальный адрес кодовой группы изображения символа. Далее видеоадаптер берет каждый байт кодовой группы изображения и работает уже с отдельными битами байта - для нулевых битов выводит пиксел цветом фона, а для единичных - цветом рисунка (коды цвета фона и рисунка он тоже берет из видеопамяти - из байта атрибутов). Вот так появляются на экране дисплея рисунки букв, тоже, как и все в компьютере, закодированные 2-ичными числами. Очень похожа картина при выводе изображений символов на печать, только коды изображений символов и их порядковые номера хранятся в памяти печатающего устройства либо постоянно, либо заносятся туда из памяти компьютера перед началом печати. Единички в кодах рисунков символов расшифровываются при этом как необходимость, например, удара соответствующей иголки в игольчатых устройствах печати.

2.3.10. Адрес числа в памяти - это тоже число.

Для доступа к размещенным в оперативной памяти данным или командам хранящейся там программы мы должны знать адреса необходимых нам объектов хранения. Память компьютера, как мы уже выяснили, - это просто линейная цепочка байтов и нетрудно определить адрес объекта (любого размера) в памяти как порядковый номер его первого байта в этой цепочке.

Данные всех типов обрабатываются в компьютере специальным устройством - центральным процессором, который имеет для размещения адресов свои внутренние участки памяти -

АДРЕСНЫЕ РЕГИСТРЫ

От размера этих регистров зависит размер памяти, к которой процессор может непосредственно адресоваться.

В процессорах первых поколений Intel 8088/8086 размер адресных регистров был 16 битов или 2 байта - максимальное число (номер байта в памяти), которое может быть записано в такой регистр, 2^16-1=65535 или, как его округленно называют - 64 килобайта

(1 килобайт = 1024 байта).

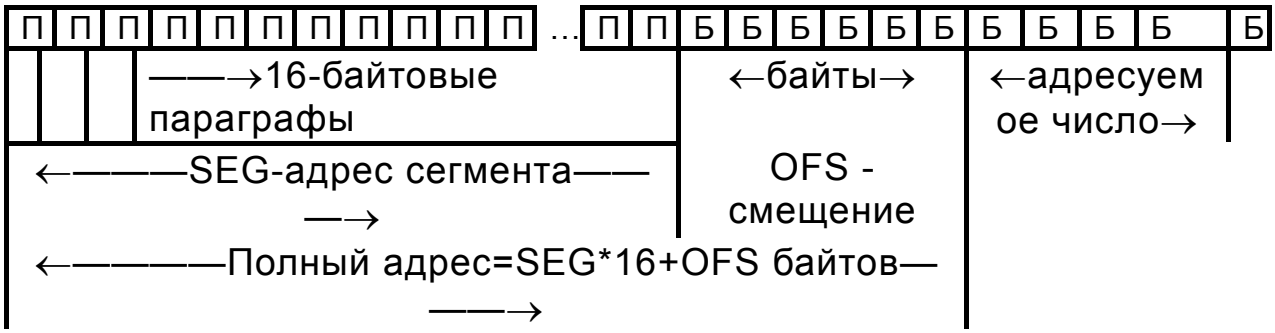
Но уже в компьютерах на базе этих процессоров возникла необходимость адресоваться к памяти размером 1024 килобайта (1 мегабайт). Чтобы выйти из затруднения с регистрами недостаточного размера, для адресации отвели 2 регистра и решили составлять адрес из 2-х частей : первая часть адреса содержит номер блока памяти как количество 16

байтовых **ПАРАГРАФОВ**, а вторая часть номер байта, отсчитанный от границы (начала) параграфа. Первую часть адреса назвали адресом **СЕГМЕНТА**, а вторую - **СМЕЩЕНИЕМ** внутри сегмента. Очевидно, что обе составляющие при 16-битовом регистре адреса не могут быть больше 65535, а для вычисления полного адреса любого объекта в памяти необходимо его сегментную часть умножить на 16 и прибавить смещение. Умножение двоичного числа на 16 -четвёртую степень основания системы счисления, двойки - равносильно приписке справа 4-х нулей или сдвигу влево на 4 двоичных разряда с "втягиванием" нулей на место освобождающихся битов. Ради удобства адреса обычно записывают в 16-ричной системе счисления. При этом умножение на 16 16-ричного числа равносильно приписыванию справа одного нуля.

Пусть сегментный адрес некоторого объекта в памяти 045F, а смещение равно 0032 ; тогда полный адрес равен

```
  045F0
      +
   0032
  ————
  04622
```

Приведенная ниже схема иллюстрирует формирование адреса числового объекта в электронной памяти.



Максимальный адрес, по которому можно обратиться к памяти при такой 2-ступенчатой адресации FFFF0 или 65535 * 16 = 1048560.

Если все смещения будут максимальными - по 64 Кбайта -, то всего можно иметь до 16 сегментов. Процессоры следующих поколений имели уже более длинные регистры для хранения адресов - 80286 использует 24 битовые и позволяет адресовать напрямую до 16 мегабайт, а 80386 имеет 32-битовые регистры и может адресовать до 4 Гигабайт. Таким образом, в компьютерах с 32-битовыми регистрами практически отпала необходимость в сегментации памяти для обеспечения "дальности" адресации.

Но сегментация сохранилась по другой причине - оказывается, удобно хранить, например, коды программных команд и обрабатываемые данные в различных сегментах - так меньше вероятность запортить программный код при изменении данных в памяти и легче отлаживать программы. Так и живут программы и данные как в многоквартирном доме - в одной и той же памяти, но в различных сегментах - кроме отдельных малогабаритных программ с размещением данных и команд в одном единственном сегменте.

Таким образом, сегментом можно называть произвольную область памяти, которая начинается на границе параграфа, то есть ее расстояние от начала памяти в байтах кратно 16.

2.3.11. Адрес и указатель.

Для удобства в дальнейшей работе мы введем новое понятие - "указатель на элемент заданного размера". Это будет тоже адрес, но с обязательной оговоркой о том, что этот адрес адресует, точнее, на элемент какой длины в памяти он ссылается. Нам это нужно для осуществления адресной арифметики - было бы удобно для получения адреса следующего элемента в памяти при "плотной укладке" однотипных элементов друг за другом просто нарастить адрес текущего на 1 - указатель должен сам знать, на сколько байтов переместиться для этого в памяти.

Таким образом, указатель при указанном типе адресуемого элемента «байт» будет называться: "указатель на однобайтовый элемент" и при наращивании его на 1 будет наращивать свое значение на 1 байт. Если же мы определим его как "указатель на 2-хбайтовый элемент", то при наращивании его на 1 он будет увеличивать свое значение на 2, а при определении указателя на 4-хбайтовый элемент увеличение указателя на 1 должно привести к увеличению содержащегося в нем адреса на 4 и так далее. Договоримся также, что "указатель вообще" или "указатель на элемент неопределенного типа (неизвестной длины)" полностью эквивалентен понятию полного адреса, но не может наращиваться, так как "не знает", на сколько байтов сместиться в памяти, чтобы ссылаться на следующий элемент.

2.3.12. Текстовые строки как последовательность двоичных чисел и некоторые операции с ними.

Что собой представляет текстовая строка в оперативной памяти? Целесообразно договориться, что строка текста - это

последовательность однобайтовых чисел - (ASCII-кодов символов), для которой можно определить начало и конец, чтобы иметь возможность обрабатывать ее отдельно от других строк текста. Вполне естественно начало строки в памяти фиксировать значением указателя на ее первый байт (символ, букву). Если символы строки нумеровать с 0, то любой i-й доступен по увеличенному на i начальному значению указателя. Для определения конца строки (ее последнего символа) используют 2 основных метода.

По первому дополнительный 0-й байт строки используют для размещения в нем числа, которое является длиной строки (количеством принадлежащих ей символов), а собственно строка начинается со следующего, первого символа. При всех операциях со строкой, приводящих к изменению ее длины (вставка символов, удаление, присоединение других строк - так называемая конкатенация), необходимо при этом заботиться о корректировке ее нулевого байта. Максимальная длина строки при этом ограничивается 255 символами - больше в 1-байтовый элемент длины записать просто нельзя.

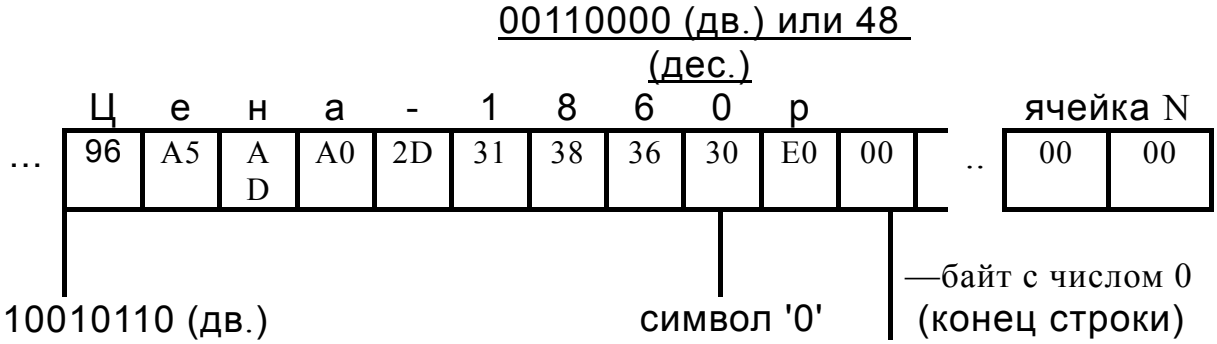
Второй метод тоже связан с увеличением реальной длины строки на 1 байт, но этот байт помещается в конец строки, за ее последним значащим символом и просто является индикатором конца. В качестве такого индикатора выбрали число 0 и строки этого типа называют **ASCIIZ**-строками (Z - от ZERO - нуль), а нуль-байт в конце строки называют еще нуль-терминатором (от to terminate - разделять, ограничивать, прерывать). Преимуществом такой организации строк является отсутствие необходимости вести постоянный счет символов в строке и корректировать начальный байт длины (мы при вставках букв просто смещаем стоящий после места вставки "хвост " строки вправо на 1 байт, а при удалении - влево вместе с завершающим строку нуль-байтом). Кроме того, при таком подходе никаких формальных ограничений на длину строки в памяти не накладывается, строка - это все символы от первого до предшествующего "нуль-байту". При побуквенной обработке строки можно двигаться от начала до встречи с числом 0. Мы будем пользоваться ASCIIZ - строками.

Пусть в памяти надо "уложить" строку

Ц	е	н	а	-	1	8	6	0	р

Воспользовавшись приведенной выше таблицей кодировки, мы можем составить 16-ричное представление байтов строки (см. ниже) в оперативной памяти. Там же представлены двоичное и десятичное представление некоторых

символов; для остальных вам предлагается проделать это самостоятельно.



Пусть адрес строки в памяти (то есть адрес ее самого начального, нулевого по счету элемента, в приведенном примере - буквы 'Ц') задан указателем по имени StrPtr на однобайтовый элемент. Разработаем некоторые методы обработки строк.

1) Пусть необходимо цифровую подстроку "1860" преобразовать в соответствующее ей число, например для использования его в арифметических операциях в будущем. Выделим для этой цели двухбайтовую ячейку в памяти и назовем ее для удобства работы каким-либо именем, например N. Сразу после выделения памяти ее содержимое не определено, но мы запишем туда число 0:

N=0;

Теперь установим указатель на первую цифру цифрового слова, в нашем случае - на цифру '1' следующими действиями:

Пока значение по адресу StrPtr меньше 30h или больше 39h наращиваем StrPtr на 1 :

StrPtr=StrPtr+1;

и в конце концов окажемся на начальной цифре числа.

Теперь преобразуем последовательность кодов цифр числа в число следующими повторяющимися действиями:

Пока значение по адресу StrPtr больше или равно 30h и меньше или равно 39h (то есть пока это цифра), вычитаем из этого значения 30h (преобразовав тем самым в обозначенное цифрой число) и прибавляем увеличенное в 10 раз значение в ячейке N (учтя тем самым позицию цифры в числе), полученный результат на каждом шаге будем записывать в ячейку N:

N=(*StrPtr-30h)+10*N

"значение по адресу" знак умножения

Здесь и в дальнейшем для краткости записи фразы "значение по адресу" мы заменили знаком '*' перед именем указателя (не путайте его со знаком умножения). В итоге в двухбайтовой ячейке N окажется число 1860, или в двоичном представлении 0000011101000100 или в 16-ричном представлении 0744 (его символьное представление занимало 4 байта - по байту на цифру).

2) В предыдущей задаче мы предполагали, что в текстовой строке обязательно есть единственная цифровая подстрока - в реальных же задачах их либо может быть несколько (и заранее неизвестно сколько), либо не быть вообще. Поэтому теперь мы обобщим задачу так: обнаружить в заданной своим адресом StrPtr строке все имеющиеся цифровые подстроки, преобразовать их в числа и выдать пользователю, например, запоминанием на внешнем запоминающем устройстве.

Для сокращения записи введем еще некоторые обозначения - восклицательным знаком будем записывать отрицание "Не", например, выражение !=0 будет означать "Не равно нулю", а двумя знаками равенства будем обозначать "равно", например, ==0 будет означать "равно 0". Кроме того, введем более удобное обозначение для кодов символов методом заключения символа в одиночные кавычки; так, запись 'A' будет обозначать ASCII-код символа A, то есть число 65, а запись '8' будет обозначать код символа 8, то есть число 56 (дес.) или 38 (16-ричн.) или 00111000 (дв.). Таким образом, запись цифры без кавычек будет у нас означать представляемое цифрой число, а запись цифры в кавычках - код соответствующего сим вола. Записи вида >= или <= будут означать "больше или равно" и "меньше или равно" соответственно.

Общий метод решения этой задачи можно записать так:

Выделим память под числовую переменную по имени flag и запишем туда число 0 - это будет счетчик числовых подстрок в общей строке.

Повторять бесконечно заключенные в фигурные скобки действия:

{ Если *StrPtr равно 0, мы достигли конца строки и поэтому выходим за закрытую фигурную скобку, покидая повторяющуюся последовательность операций.

Пока *StrPtr не код цифры (*StrPtr<'0' || *StrPtr>'9')
и не нуль (StrPtr!=0), двигаемся вдоль строки, наращивая StrPtr: StrPtr=StrPtr+1;

Это движение закончится как только *StrPtr станет кодом цифры или встретится конец всей строки.

Пока *StrPtr есть код цифры (*StrPtr>='0' && *StrPtr<='9'),
преобразуем цифровую последовательность в число уже известным нам методом:

N=*StrPtr-30h+10*N; StrPtr=StrPtr+1;flag=flag+1;

Если flag не равен 0
передаем результат пользователю и возвращаемся к первой операции в фигурных скобках.

}

Если flag равен 0, мы не встретили в строке ни одной цифровой подстроки и поэтому передаем пользователю сообщение: "В этой строке нет цифровых подстрок."

3) Пусть теперь перед нами стоит задача определить длину строки в байтах. Для результата используем ту же самую предварительно обнуленную ячейку N, на начало строки по-прежнему указывает StrPtr. Метод решения задачи выглядит очень просто:

N=0;
Пока *StrPtr не равно 0 наращиваем на 1 N и StrPtr

N=N+1; StrPtr=StrPtr+1;

"Наткнувшись" при движении указателя вдоль строки на нулевое число по его адресу, мы прекратим наращивания - в этот момент в N будет количество букв в строке, а StrPtr будет указывать на байт в памяти с записанным в нем числом 0, то есть на код конца строки.

4) Пусть необходимо скопировать строку по адресу StrPtr1 в область памяти, адрес которой StrPtr2 (в предположении, что для копии строки, начинающейся по этому адресу, выделено достаточно памяти). Метод решения тоже прост:

Переписываем число *StrPtr1 в *StrPtr2 и, пока *StrPtr1 не 0, наращиваем адреса StrPtr1 и StrPtr2:

Повторяем действия в фигурных скобках:
{
Копируем символ *StrPtr2=*StrPtr1;
Если *StrPtr1 не 0, наращиваем указатели
StrPtr1=StrPtr1+1; StrPtr2=Str1Ptr2+1;

иначе прекращаем повторения.
}

Обратите внимание на некоторое отличие в последовательности действий по сравнению с предыдущими задачами - мы сначала переписываем значение числа из одной области памяти в другую, а затем проверяем, нужно ли наращивать адреса оригинала и копии. Это сделано для того, чтобы скопировать и нуль-байт конца строки.

5) Пусть необходимо сравнить между собой 2 строки, начинающиеся по адресам StrPtr1 и StrPtr2. Так как строки - это последовательности однобайтовых целых чисел в памяти компьютера, то 2 строки естественно считать равными, если у них одинаковая длина и все числа - коды символов одинаковы. Если длины строк разные и равны l у более короткой и L у более длинной, но первые l букв у обеих строк совпадают, то большей считается более длинная. При встрече с первой же парой несовпадающих кодов символов большей сразу признается та строка, у которой в сравниваемой паре кодовое число больше.

При таком подходе напрашивается простой метод решения:

Будем двигаться синхронно вдоль обеих строк, вычитая в сравниваемой паре кодов символов код во второй строке из кода в первой, пока получаемая разность равна 0 и сумма кодов символов не равна 0 :

```
Повторять указанные в фигурных скобках действия
{
Вычислять разность значений по заданным адресам
R=*StrPtr1-*StrPtr2;
и наращивать указатели на коды символов строк
StrPtr1=StrPtr1+1; StrPtr2=StrPtr2+1;
}
пока ((R равна 0) и (*StrPtr1 + *StrPtr2 не 0));
```

При R равном 0 после окончания циклического процесса строки равны. Если R больше 0, то первая строка больше, если R отрицательно, то первая строка меньше второй. Если строки равны и по длине и посимвольно, то прекращение работы произойдет по нарушению условия (*StrPtr1 + *StrPtr2 не 0) при "наезде" указателей на нуль-байты конца обеих строк.

2.3.13. Команды обработки чисел - это тоже числа.

В предыдущих примерах мы сами выполняли различные операции над размещаемыми в электронной памяти числовыми объектами, подменяя мысленно работу устройства, которое называется центральным процессором. Чтобы наш компьютер был универсальным вычислительным роботом, то есть мог выполнять произвольные последовательности операций над произвольными исходными данными, необходимо, чтобы как данные, так и команды для осуществления нужных операций, могли при необходимости "загружаться" в память извне, например, из внешнего запоминающего устройства типа магнитного диска, на котором они могут храниться и при выключенном питании компьютера. Тогда мы можем записывать в электронную память нужные в текущий момент последовательности команд (программы обработки) и последовательности числовых объектов, представляющих обрабатываемые данные, а центральный процессор сможет брать из памяти команды одну за другой и выполнять их. Так как память - техническая реализация последовательности двоичных цифр и ничего, кроме чисел, в нее записать нельзя, то естественно представить все команды процессора тоже в виде числовых кодов - именно так они и представлены в электронных компьютерах.

Итак, команды выполняемых программ - это числовые коды действий над закодированными в числовой форме данными любого происхождения.

Какую информацию должен нести код процессорной команды? Прежде всего - это код операции или код выполняемого действия - поместить число (например, в регистр), сложить, умножить и т д. Кроме того, в коде команды, как правило, присутствуют уточняющие код операции части, кодирующие, какие именно регистры предполагается использовать и их размеры, способ адресации (непосредственная, косвенная - через указанный в определенном регистре адрес) к обрабатываемым данным. И, наконец, ссылки на сами участвующие в операции данные (операнды). В зависимости от таких уточняющих данных процессорная команда может занимать в памяти один или несколько байтов. В самом сложном случае она содержит байт кода операции, байт способа адресации и 4 байта ссылок на операнды.

Приведем примеры числовых кодов некоторых команд в 16-ричном формате. Команды с однобайтовыми кодами:

40 - увеличить значение в регистре процессора на 1;

50 - записать значение из регистра в специальную память временного хранения данных под названием "стек".

Пример 2-хбайтовой команды:

03D8 - сложить значения в двух процессорных регистрах;

F6E3 - перемножить значения в двух процессорных регистрах;

Пример 3-хбайтовой команды:

A1XXXX -переписать значение из регистра в участок оперативной памяти по адресу, заданному байтами XXXX.

Мы не собираемся излагать здесь методы программирования непосредственно в числовых машинных кодах, хотя не так уж давно этой работой занималось немало программистов, составивших столь нелегким способом много полезных программ. Приведенные примеры служат лишь для представления о том, как выглядят и как хранятся команды в оперативной памяти. Схематично для некоторой программы это расположение можно изобразить примерно так:

Адрес 0 - начало памяти	Общесистемные данные и программы управления оборудованием компьютера
Адрес нашей программы	Коды команд обработки данных
	Обрабатываемые данные
	Остальная память

Размещаются последовательности команд в электронной памяти, как правило, отдельно от данных и сегментный адрес этой последовательности записывается процессором в отдельный участок внутренней памяти - регистр кодового сегмента, чтобы исключить случайное перепутывание программного кода с обрабатываемыми данными, которые, в свою очередь, группируются в памяти и сегментные адреса этих групп тоже записаны для текущей программы в отдельный регистр процессора - регистр сегмента данных.

2.3.14. Память с произвольным доступом и память только для чтения.

Различают 2 вида оперативной памяти - память, доступная для чтения и записи, - это основная оперативная память **RAM** (***Random Access Memory***). Все размещаемые в ней программные коды и данные должны быть "загружены" извне после включения питания и исчезнут после выключения. Любое

место в этой памяти может быть перезаписано другими значениями в любой момент времени.

Возможность размещения не только произвольных данных, но и произвольных комплексов команд для обработки данных в оперативной памяти делает компьютер не просто вычислительным роботом, а универсальным, свободно программируемым вычислительным роботом.

Некоторые изменяющиеся данные, например, о составе оборудования компьютера, паролей для доступа к основным установкам и возможности запуска компьютера, а также некоторые простые программы типа часов реального времени и календаря желательно хранить в оперативной памяти произвольного доступа и после выключения питания. Для этой цели компьютеры оснащаются небольшими блоками оперативной памяти, выполненными на элементах с малым энергопотреблением, изготовленными по специальной технологии - это так называемая **CMOS** - память или **КМОП** - память (металл-окисел-полупроводник), постоянно подпитываемая при отключении от сети установленным на основной печатной плате аккумулятором небольшой емкости.

Очевидно, что некоторые программы, которые стартуют сразу после включения питания, должны всегда находиться в памяти - для этой цели служит память, в которой коды команд и данных записаны, например, методом "прожигания перемычек" - то есть навсегда. Содержимое такой памяти **ROM** (***Read Only Memory***) нельзя перезаписать - именно в ней размещены программы **ROM BIOS** (***Basic Input Output System*** - части базовой системы ввода-вывода, размещенной в памяти только для чтения) - тестирования основного оборудования, начальный загрузчик системных программ. В такой же памяти размещается иногда программа интерпретатора языка БЕЙСИК. Эта память размещается в отдельных, заранее запрограммированных микросхемах.

2.3.15. Регистры и порты устройств, входящих в состав компьютера.

Все устройства компьютера выполняются обычно как многорежимные и многофункциональные; для изменения режима работы устройства или выполнения им определенной функции необходимо иметь возможность передать ему числовой код запроса. Кроме того, бывает необходимо определить текущий режим работы устройства или его предельные возможности (например разрешающую способность дисплейного адаптера или количество цветов в текущем

режиме). Для обеспечения возможности общения с программами все устройства имеют управляющие и информационные регистры - внутренние участки оперативной памяти длиной в один или два байта, в которые можно записывать управляющие коды или из которых можно считывать информацию о текущих параметрах устройства. В сложных устройствах таких регистров может быть достаточно много (десятки). Поэтому для доступа к регистрам каждое устройство имеет закрепленные за ним специальные регистры - порты и через эти порты осуществляется доступ к управляющим и информационным регистрам устройства. Этот доступ осуществляется обычно по унифицированной для всех устройств схеме: через один порт посылается "заявка" на установление связи с нужным регистром устройства, а затем через этот же или другой (как правило, следующий за ним по номеру) порт посылается управляющий код или считывается информационный код из нужного регистра. И управляющие, и информационные коды представляют собой одно- или двухбайтовые целые двоичные числа, значение каждого бита которых наделено определенным смыслом и вызывает соответствующую реакцию устройства (если это код управления) или характеризует его режим (если это информационный код). Поэтому в составе команд компьютера должны быть такие, которые позволяют работать не только с числами в целом, но и с их отдельными цифрами - битами.

В прикладных программах часто возникает также необходимость использования элементов данных с длиной меньше байта - в 1, 2 или 3 бита. Например, если вы составляете компьютерный журнал учета успеваемости учеников класса, группы, учебного заведения, содержащий оценки каждого опрошенного по отдельным разделам, темам, предметам, то выделение для каждой оценки минимальной адресуемой единицы памяти - байта будет слишком расточительным, ведь при реально 4-бальной системе оценивания вполне достаточно 2 бита на оценку и неплохо бы "упаковать" оценки по 4 в каждый байт. Но тогда нам тоже понадобится возможность работы с отдельными битами байтов.

2.3.16. Побитовые операции с целыми двоичными числами.

"Побитовыми" называют такие операции с 2-ичными числами, которые осуществляются с отдельными цифрами одного и того же двоичного числа (унарные операции) или парами битов двух различных чисел одинакового размера

(бинарные операции). С унарной операцией инвертирования битов числа мы уже встречались - она переводит единичные биты числа в нулевые, а нулевые в единичные. Называется она побитовым отрицанием и обозначается по разному в различных формализованных языках - например записью перед числом слова **NOT**. Мы используем обозначение этой операции символом '~' - "тильда" перед числом и запись ~d будет обозначать инвертирование всех цифр двоичного числа d. Если это было число

d = 10101010(дв) или d = 170(дес),

то ~d даст результат

01010101(дв) или 96(дес).

Отдельную группу побитовых операций составляют операции сдвига для двоичных чисел. Различают простые (не обращая внимания на знаковый бит) и арифметические (с учетом знакового бита) сдвиги. При простых сдвигах влево или вправо на место "освобождающихся" битов просто "втягиваются" нули:

00011000 (24дес) сдвинем влево на 2 бита (<<2), получим

01100000 (96дес)

Фактически мы умножили число 24 на 4, а сдвиг вправо на те же 2 бита приведет, очевидно, к делению на 4:

00011000 (24дес) сдвинем вправо на 2 бита (>>2)

получим 00000110 (6дес)

"Выдвигаемые" за пределы разрядной сетки значащие биты теряются:

00110011 (51дес) сдвинем вправо на 2 бита (>>2)

00001100 (12дес) остаток от деления на 4 "пропал"

11001100 (204дес) сдвинем влево на 2 бита (<<2)

00110000 (48дес) результат умножения на 4 не разместился в одном байте и мы получили 816-3*256=48 после троекратного переполнения разрядной сетки однобайтового числа. Точно такие же эффекты вы будете получать в ваших программах, если не будете обращать внимания на то, разместится ли результат умножения в отведенном для него участке памяти.

При арифметических сдвигах вправо левые биты заполняются значениями знака исходного числа - для положительных чисел нулями, а для отрицательных - единицами, так что величины сохраняют исходный знак:

10110111 сдвинем вправо на 1 бит получим 11011011

10110111 сдвинем вправо на 3 бита

получим 11111011

Арифметические сдвиги влево полностью идентичны простым - левые биты заполняются нулями, а правые теряются.

Отдельную группу сдвигов составляют так называемые циклические сдвиги, при которых выдвигающиеся биты занимают освобождающиеся разряды:

10110111 сдвинем циклически вправо на 1 бит
получим 11011011
10110111 сдвинем циклически вправо на 3 бита
получим 01111011

Рассмотрим теперь три наиболее часто используемые в программах двухместные побитовые операции с парами двоичных чисел. Очевидно, что для получения корректного результата при попарной обработке битов с одинаковыми номерами 2-х числовых объектов, сами объекты должны быть одинаковой длины - 1-нобайтовыми, 2-хбайтовыми и т.д.

К 2-хместным побитовым операциям относятся:

⇒операции побитового "**И**" ("**AND**" - мы будем использовать для этой операции значок **&** - символ "амперсанд"),

⇒побитового "**ИЛИ**" ("**OR**" - мы будем использовать значок **|**),

⇒побитового "**ИСКЛЮЧАЮЩЕГО ИЛИ**" - ("**XOR**" - мы будем использовать значок **^**).

Операция "И" ("AND" - &) сравнивает значения битов в одних и тех же позициях двоичных чисел и результирующий бит в той же позиции делает единичным только тогда, когда оба сравниваемые биты единичны, в остальных случаях результирующий бит нулевой.

Операция "ИЛИ" ("OR" - |) сравнивает значения битов в одних и тех же позициях двоичных чисел и результирующий бит в той же позиции делает единичным, если хотя бы один из сравниваемых битов единичен, если же оба нули, то и результирующий бит нулевой.

Операция "ИСКЛЮЧАЮЩЕГО ИЛИ" ("XOR" - ^) сравнивает значения битов в одних и тех же позициях двоичных чисел и результирующий бит в той же позиции делает единичным, если ТОЛЬКО один из сравниваемых битов единичен, если же оба нули или единицы, то результирующий бит нулевой.

Рассмотрим примеры. Пусть a=01010101(85дес); b=00110011(61дес);

c=a & b;	d=a | b;	e=a ^ b;
01010101	01010101	01010101
00110011	00110011	00110011
———————	———————	———————

00010001(17дес) 01110111(119дес) 01100110(102дес).

Побитовые операции часто используются для определения значения отдельных битов чисел, для установки отдельных битов в 1 или 0.

Так, если нужно проверить значение бита с номером n (счет ведется справа налево и начинается с 0) в числе a, то используют операцию "И" этого числа с числом, у которого все биты нулевые, а единичен только бит в проверяемой позиции, то есть с числом-тестом, равным 2 в степени n. Очевидно, что результат будет нулевым, если проверяемый бит нулевой и равен числу - тесту, если проверяемый бит единичен. Пусть надо проверить значение 6-го бита в числе a:

Если (a & 64) равно 0 - 6-й бит нулевой, иначе единичен.

Проверка бита 6 числа a:

```
a=  01110101
 &
    01000000 (число 64дес)
    ————————
    01000000 - бит 6 равен 1
```

Простейший вариант использования - выяснение четности или нечетности числа: для этого не надо выяснять, нулевой ли остаток от деления этого числа пополам - достаточно выяснить, нулевое ли значение операции "И" этого числа с числом 1, так как все четные числа имеют четную последнюю цифру, а в двоичных числах только одна четная цифра - 0, и только одна нечетная - 1:

Если (a & 1) равно 0 то a четно, иначе - нечетно.

Если (a & 1) равно 1 то a нечетно, иначе - четно.

Проверка четности числа a:

```
a=  01110101
 &
    00000001 (число 1дес)
    ————————
    00000001 - число нечетно
```

Предпочтение этой операции перед делением следует отдавать всегда, так как побитовые операции выполняются процессором непосредственно и очень быстро в отличие от операций деления и умножения.

Если нам необходимо установить n-й бит числа a в 1, не изменяя неизвестных нам значений остальных цифр, то удобно использовать операцию "ИЛИ" с числом, у которого "взведен в 1" только n-й бит, то есть с числом 2 в степени n. Если при этом

устанавливаемый бит был 0, то он станет 1, если был 1, то таковым и останется, а остальные не изменят своего значения. Итак, "взвести" бит 7 в числе a: a = (a | 128).

Установка бита 7 в числе a: a = (a | 128). Например:

```
  01110101
|
  10000000 (число 128дес)
  ————————
  11110101 - изменился в 1 только 7-й слева бит.
```

Для обнуления n - го бита в числе a без изменения неизвестных значений остальных цифр используйте операцию "И" с инвертированным значением числа с единичным битом в позиции n:

Обнуление бита 5 в числе a: a = (a & ~32). Например:

```
   01110101
&
   11011111 (инверсия числа 00100000(32дес)
   ————————
   01010101 - изменился в 0 только 5-й слева бит.
```

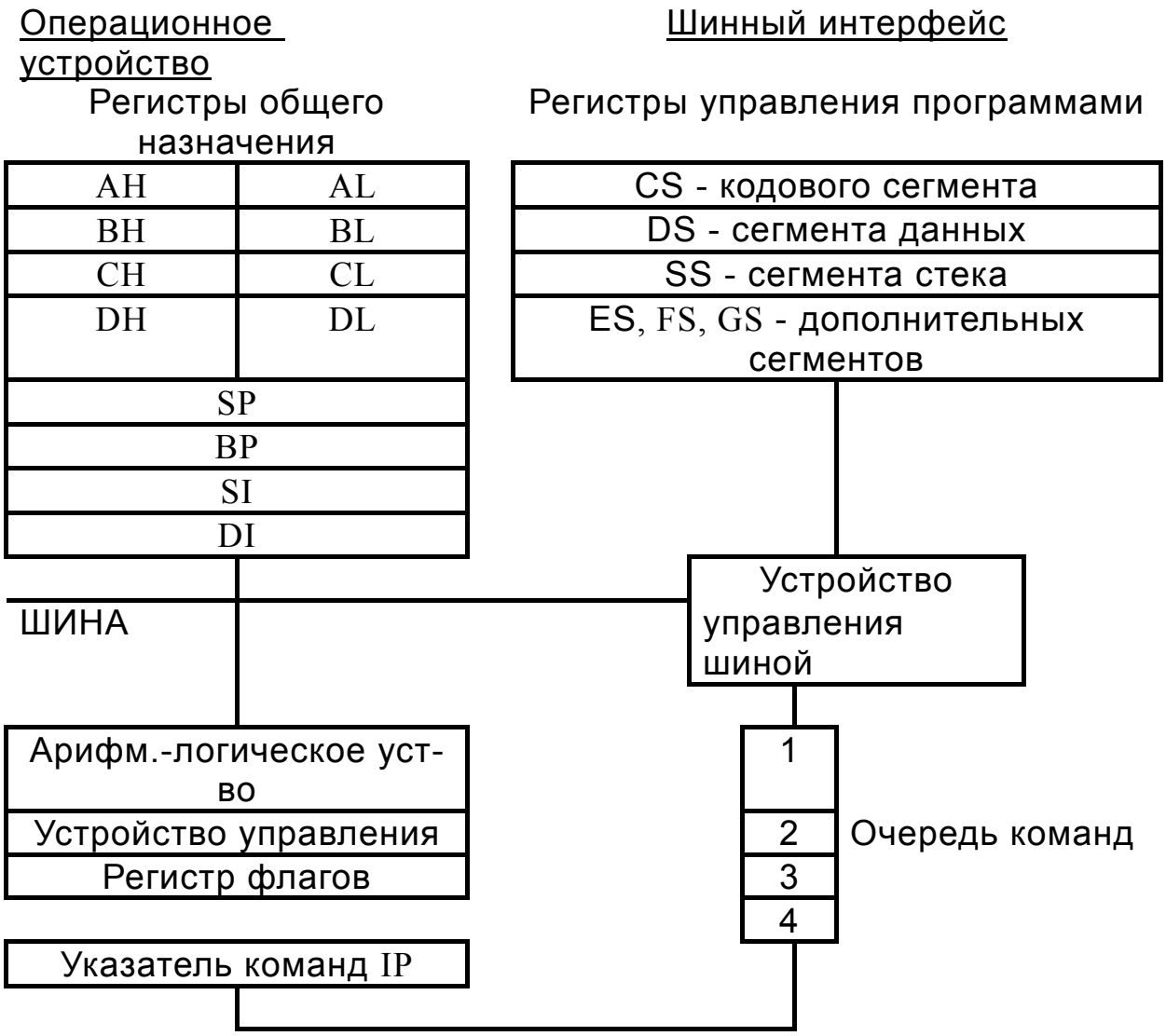
В заключение этого раздела отметим, что все участки памяти, содержимое которых может изменяться с помощью команд программы, принято называть переменными. Переменные, которые могут принимать только 2 фиксированных значения, например 0 или 1, относятся к типу так называемых "булевских" переменных (по имени математика Буля, разработавшего раздел математической логики - алгебру для таких переменных). Бит двоичного числа является идеальной моделью булевских переменных и рассмотренные побитовые операции относятся поэтому к логическим операциям над булевскими переменными.

2.3.17. Архитектура центрального процессора.

Если в памяти компьютера уложены числа-команды и числа - обрабатываемые данные, то необходимо устройство, которое будет брать из памяти команды и выполнять предписанные ими действия над лежащими в памяти объектами в числовом представлении.

Таким устройством в компьютере является центральный процессор (**ЦП**) - он и выполняет основную обработку информации, заданной в виде двоичных числовых кодов и состоит из 2-х функциональных частей - операционного устройства и шинного интерфейса.

Упрощенная блок-схема ЦП :



ШИННЫЙ ИНТЕРФЕЙС предназначен для подготовки команд к выполнению операционным устройством и содержит устройство управления шиной, очередь команд и регистры для сегментных адресов.

Устройство управления шиной управляет передачей данных на операционное устройство своего процессора, в оперативную память и на внешние устройства ввода-вывода.

Очередь команд - это 4 или более регистров памяти. Нужны они вот для чего. Все команды находятся в оперативной памяти и шинный интерфейс должен заглядывать вперед и выбирать команды так, чтобы всегда существовала непустая очередь подготовленных к выполнению команд.

Операционное устройство берет команды из очереди команд и пока оно занято выполнением первой в очереди

команды, шинный интерфейс выбирает из памяти следующую - такая параллельная работа повышает производительность.

Шесть сегментных регистров служат для хранения сегментных адресов начала программного кода (регистр CS), глобальных данных (регистр DS), локальных данных (регистр сегмента стека SS); дополнительные сегментные регистры не имеют специального назначения. Таким образом, регистры шинного интерфейса содержат адреса физических сегментов памяти, отводимой программам для кода (CS), данных (DS) и стека (SS). Каждый из этих 16-тибитовых регистров может адресовать не более 64 Кбайт памяти. Для адресации 1-Мбайтного пространства к 16-битовому регистру воображаемо присоединяют 4 нулевых бита (физически отсутствующих), что равносильно умножению содержимого на 16 и дает 64*16=1024 Кбайт, но с округлением адреса сегмента до 16-байтового.

Здесь мы встретились с незнакомым термином "стек" -это участок памяти, организованный с помощью программы таким образом, что последние помещаемые в него данные извлекаются затем первыми. Этот порядок носит название "дисциплины **LIFO** (***Last In - First Out***, то есть последним вошел - первым вышел)".Такая организация оказывается полезной прежде всего в тех случаях, когда вашей программе необходимо "на время отвлечься" для выполнения специальной работы с возвратом к команде, следующей за командой "ухода". Фрагмент программного кода, выполняющий это спецзадание, называется подпрограммой и адрес возврата из подпрограммы запоминается в стеке. Подпрограмма может иметь свои локальные данные - они тоже размещаются в стеке. Подпрограмма может в свою очередь вызвать другую подпрограмму и из нее необходимо вернуться в вызвавшую подпрограмму - вот почему извлечение более поздних записей необходимо бывает раньше предшествующих - для этого и организуется стековая память.

ОПЕРАЦИОННОЕ УСТРОЙСТВО предназначено для выполнения команд и включает указатель команд IP (это по существу адрес - смещение команды относительно сегментного адреса, хранящегося в регистре CS шинного интерфейса), регистры общего назначения и собственно процессор. Назначение регистров операционного устройства:

SP и BP- регистровые указатели для доступа к данным сегмента стека (SP-для смещения относительно сегментного адреса стека, BP - дополнительный регистр базы, облегчающий доступ к адресам и данным, размещенным в стеке).

SI и DI - индексные регистры (SI-смещение относительно адреса в DS, DI- то же относительно ES).

IP - указатель команд.

AX - основной сумматор, используется для всех операций ввода-вывода, некоторых строковых и арифметических.

BX - базовый регистр общего назначения для расширенной адресации и вычислений.

CX - счетчик циклов и операций сдвига.

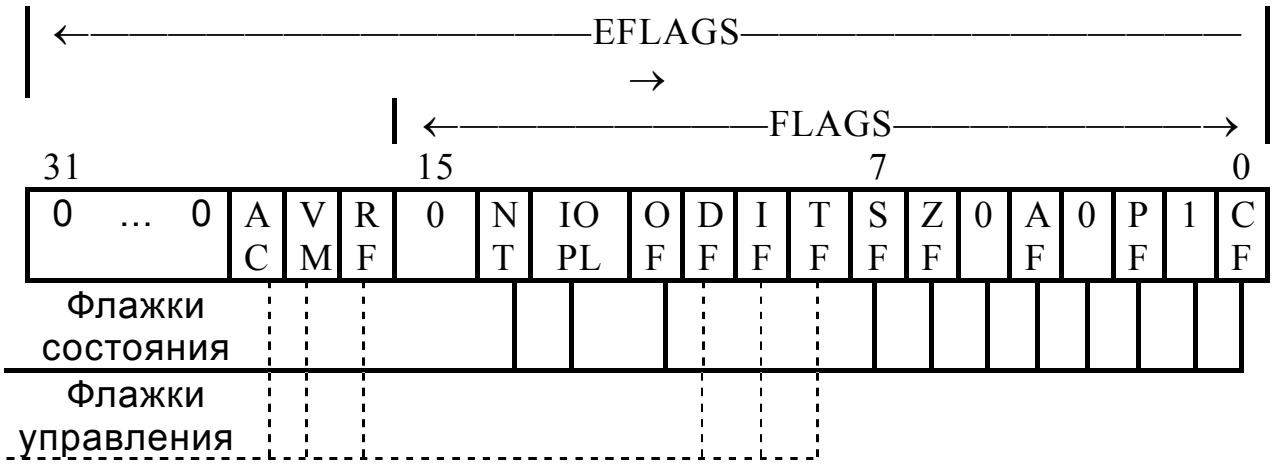
DX - регистр данных для операций умножения и деления чисел двойной длины (в паре с AX-DX:AX) и некоторых операций ввода-вывода.

Все 4 вышеперечисленных регистра AX,BX,CX,DX позволяют выполнять операции и над их старшими (с буквой H), и над младшими (с буквой L) однобайтовыми половинками. Начиная с 386 процессора все регистры, за исключением сегментных, стали 32-разрядными, то есть 4-хбайтовыми. При этом можно обратится к регистру целиком, к его половинкам и для 4-х регистров - к двум нижним четвертушкам (например, вы можете получить доступ к 32-хразрядному регистру EAX, к младшему его слову - 16-тиразрядному регистру AX и к старшему и младшему байтам последнего - AH и AL).

СОБСТВЕННО ПРОЦЕССОР состоит из арифметико-логического устройства (**АЛУ**), устройства управления и регистра флагов. АЛУ выполняет команды двоичной целочисленной арифметики , побитовые логические операции и сдвиги над любыми помещенными в регистры операционного устройства числами.

Биты **регистра флагов** определяют состояние машины и результаты выполнения команд.

Регистр флагов условий отображает информацию о состоянии центрального процессора или задает режим его работы. Он имеет вид:



Назначение флагов условий следующее:

CF (*Carry Flag*) устанавливается (делается равным 1), если при выполнении сложения возникает перенос, а при выполнении вычитания - заем из старшего бита результата;

PF (*Parity Flag*) устанавливается, если младший байт результата содержит чётное число единичных битов;

AF (*Auxiliary carry Flag*) устанавливается, если при выполнении сложения возникает перенос, а при выполнении вычитания - заем из старшего бита младшей тетрады, т.е. бита 3, результата;

ZF (*Zero Flag*) устанавливается, если результат операции равен нулю;

SF (*Sign Flag*) устанавливается в то же состояние, в котором находится старший бит результата операции (в зависимости от длины операнда это 7, 15 или 31);

Установка бита TF (*Trap Flag or Trace Flag*) переводит МП в пошаговый режим, т.е. в режим, вызывающий прерывание после выполнения каждой команды процессора;

Установка бита IF *(Interrupt Flag)* разрешает маскируемые прерывания работы МП, т.е. прерывания от внешних устройств;

Если бит DF *(Direction Flag)* установлен, то строковые команды уменьшают адрес операндов; если он сброшен, то увеличивают;

OF *(Overflow Flag)* устанавливается, если возникает переполнение результата арифметической операции; иными словами, при сложении возникает перенос в старший бит и нет переноса из старшего бита, или наоборот; при вычитании - возникает заем из старшего бита и нет заема в старший бит, или наоборот.

IOPL *(Input / Output Privilege Level)* - уровень привилегий ввода-вывода. Это двухбитовое поле является составной

частью механизма защиты 486-го процессора и действует в защищённом режиме.

NT *(Nested Task)* -флажок вложенной задачи устанавливается, когда текущая задача производит переключение на другую.

RF *(Resume Flag)* - флажок возобновления, являющийся составной частью средств отладки. Устанавливая его в 1, можно выборочно маскировать некоторые особые случаи в процессе отладки программы.

При установленном флаге VM *(Virtual Mode)* ваш новый 64-хразрядный Pentium превращается в высокопроизводительный процессор 8086.

Флажок контроля выравнивания AC *(Alignment Check)* удобен при обмене данными с другими процессорами, например с процессором i860, который требует выравнивания всех данных.

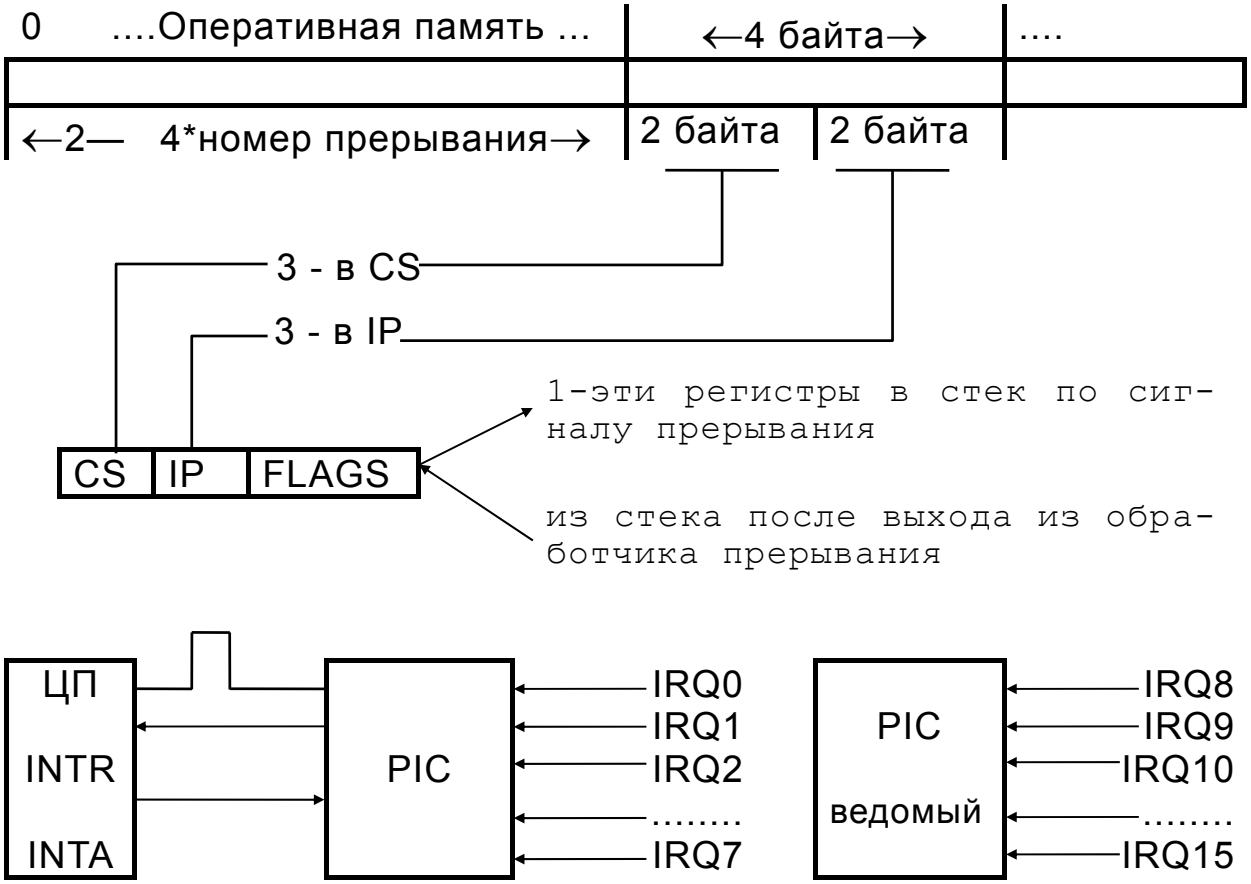
Внимание ! Старшие и младшие байты 2-хбайтовых чисел хранятся в памяти в "обратной" последовательности - младшая часть - по младшему адресу, старшая - по старшему. Об этом следует помнить при ручном заполнении байтов и непосредственной работе по абсолютным адресам.

2.3.18. Реакция центрального процессора на сигналы и команды прерывания.

Центральный процессор оснащен специальным маскируемым входом (**INTR** на рисунке), связанным с выходом микросхемы программируемого контроллера (устройства управления) прерываний (**PIC** - ***Programming Interrupt Controller***) и чувствительным к уровню напряжения. Если флаг IF равен 1 (разрешены прерывания), то на сигнал высокого уровня на линии INTR процессор после выполнения текущей команды отвечает по линии **INTA** сигналом готовности к приему по шине данных однобайтового номера аппаратного прерывания. Полученный номер он умножает на 4 и результирующее число трактует как адрес 4-хбайтового адреса входной точки подпрограммы обслуживания прерывания (**ISR** - ***Interrupt Service Routine***). Старшие 2 байта становятся сегментным адресом для регистра CS, а младшие - смещением для IP после сохранения в стеке их предыдущих значений и значения регистра флагов (это обеспечивает возможность возврата к прерванной работе). Так начинается выполнение

ISR, если она находится по этому адресу. Точно так же реагирует процессор на считанную из памяти специальную команду прерывания, только номер прерывания берет непосредственно из операнда команды.

После завершения ISR восстанавливаются сохраненные в стеке регистры и процессор продолжает выполнение прерванной цепочки команд.



Последовательность из 256 4-хбайтовых чисел в самом начале оперативной памяти называют **таблицей векторов прерываний** - она используется для хранения адресов подпрограмм обслуживания аппаратных и программных (через команды) прерываний и адресов важной системной информации.

Контроллер прерываний имеет 8 входных линий запроса прерываний (**IRQ** - ***Interrupt Request***) для подключения внешних устройств, каждой линии соответствует закрепленный за ней номер прерывания. Ведущий контроллер прерываний может иметь на втором входе подключенный каскадно второй, ведомый контроллер прерываний. Стандартное закрепление номеров линий и аппаратных прерываний следующее (неиспользуемые линии опущены):

Линия	Прерывание	Назначение
IRQ0	8h	Таймер
IRQ1	9h	Клавиатура
IRQ2	0Ah	Ведомый PIC
IRQ3	0Bh	Последовательный порт 2 или 4
IRQ4	0Ch	Последовательный порт 1 или 3
IRQ5	0Dh	Параллельный порт 2
IRQ6	0Eh	Накопитель для гибких дисков
IRQ7	0Fh	Параллельный порт 1
IRQ8	70h	Часы реального времени
IRQ9	71h	Видеоадаптер
IRQ13	75h	Математический сопроцессор
IRQ14	76h	Накопитель для жесткого диска

3. Операционная система.

3.1. Механизм прерываний и операционная система MS-DOS (*Microsoft Disk Operating System*).

Для обслуживания пользователя в части управления ресурсами компьютера (оперативной и долговременной памятью, клавиатурой и видеосистемой, печатающим устройством, портами ввода-вывода, вызовом на выполнение необходимых программ и т.д.) в дополнение к аппаратной части (*hardware*) поставляются комплексы программ (*software*) под названием ***"операционные системы"*** **(ОС).** Для персональных компьютеров, используемых в "конторской" деятельности, наибольшее применение нашла ОС фирмы Microsoft. Часть программ этой системы, обслуживающая начальный запуск ПК при включении питания, размещена в постоянном запоминающем устройстве (**ROM-BIOS -** ***Basic Input Output System*** или базовая система ввода-вывода в ПЗУ)

Центральный процессор устроен так, что при включении питания начинает работу с команды по адресу FFFF0h - входной точки программы тестирования оборудования в ROM BIOS. Затем, если тестирование прошло успешно, управление передается программе ROM-BIOS-загрузчика, которая определяет, на каком из внешних устройств долговременной памяти (магнитном диске) находится основная часть операционной системы, проверяет ее наличие, находит программу дискового загрузчика, загружает ее в оперативную память и передает ей управление. Дисковый загрузчик размещает в оперативной памяти подпрограммы дисковой части BIOS (это аппаратно-зависимые подпрограммы управления оборудованием) и аппаратно-независимые подпрограммы верхнего уровня управления компьютерными ресурсами. Адреса входных точек всех подпрограмм заносятся в таблицу векторов прерываний, чтобы их мог вызвать ЦП при возникновении аппаратных сигналов-прерываний или при выполнении команды прерывания (которая, конечно, ничего не прерывает, а только использует механизм процессорных прерываний для вызова необходимых услуг операционной системы). Все эти подпрограммы (их принято называть функциями) размещаются в оперативной памяти на все время работы ПК - такие программы называют резидентными.

После размещения в памяти всех функций ОС загружается так называемая "оболочка" или **интерпретатор команд** операционной системы - их существует большое количество, а наиболее распространенный носит название COMMAND.COM. Эта программа после загрузки в память ОС выводит приглашение к вводу команд и непрерывно проверяет, не нажал ли пользователь клавишу и при нажатиях формирует из соответствующих кодов символов командную строку. Это формирование заканчивается после нажатия клавиши Enter (return). Первое слово этой строки (отделенное от последующих пробелом) COMMAND.COM считает именем команды, последующие - параметрами команды (так называемый "хвост командной строки"). Он проверяет наличие имени команды в своих таблицах и, если находит, то после проверки допустимости параметров выполняет команду. В противном случае команда считается "внешней", размещенной на внешнем носителе; дальнейшая ее обработка осуществляется только в том случае, если после основного имени стоит отделенный точкой суффикс (расширение имени) .bat, .exe или .com.

Если имя команды имеет суффикс ".bat", то это воспринимается как размещенная на внешнем носителе макрокоманда (пакетный файл в текстовом формате), состоящая из нескольких команд - она разыскивается на внешнем носителе и, если найдена, то со всеми имеющимися в ней командами осуществляются уже описанные процедуры.

Наличие суффиксов ".exe" или ".com" свидетельствует о желании пользователя вызвать в оперативную память для выполнения находящуюся на диске программу (последовательность команд в двоичном формате) с соответствующим именем. Чтобы удовлетворить это желание, COMMAND.COM вынужден обратиться к функции DOS с номером 4Bh (по имени EXEC) с помощью команды прерывания с номером 21h.

Заметим попутно, что программы обслуживания большинства программных прерываний - многофункциональные, то есть по одному и тому же адресу из вектора прерываний находится несколько подпрограмм - функций, а иногда функции разделяются на подфункции. Все такие функции и подфункции нумерованы и описаны в справочниках по ОС. Любое обращение из программы к услугам ОС выполняется обычно так:

- в полурегистр AH регистра AX заносится номер функции, в полурегистр AL - номер подфункции (при необходимости), через другие регистры передаются необходимые для выполнения вызываемой услуги

параметры (например, для запуска на выполнение какой - либо программы необходимо передать адрес строки с именем программы и адрес хвоста командной строки с параметрами запускаемой программы);

- ♦ генерируется команда прерывания с заданным номером;
- ♦ если подпрограмма ОС должна вернуть какие-либо данные, то она помещает их в регистры процессора, откуда их забирает вызвавшая прерывание программа.

Функция 4Bh (EXEC) прерывания 21h выполняет следующие действия:

* в начале свободного блока памяти заполняет 256-байтовый справочник запускаемой программы с необходимыми для выполнения и завершения данными (в том числе хвостом командной строки запускаемой программы) - этот блок называется префиксом программного сегмента, сокращенно - PSP;
* вплотную к PSP загружается и вызывается на выполнение требуемая программа;
* после завершения вызванной программы управление возвращается в EXEC, а оттуда - в вызвавшую программу - предок, в рассматриваемом случае - в COMMAND.COM.

3.2. Внешняя дисковая память для долговременного хранения программ и данных.

Без внешней перезаписываемой памяти компьютер представляет собой или обычный калькулятор или машину для выполнения фиксированного набора программ, записанных в ПЗУ (как в специализированных игровых компьютерах, для которых каждая игра записана в микросхеме на отдельной вставной плате - картридже) - т.е. теряет универсальность. Внешняя стираемая и перезаписываемая память ПК в настоящее время выполняется в виде устройств со стационарными и съемными магнитными носителями в форме круглых пластин - дисков, покрытых с 2-х сторон слоем магнитного материала. При установке съемных (флоппи) дисков в карман дисковода к ним с обеих сторон прижимаются головки считывания - записи, которые могут с помощью специального двигателя перемещаться вдоль радиуса диска с определенным шагом, устанавливаясь на концентрические окружности - дорожки или треки. Из-за прямого контакта головок с поверхностью вращающегося носителя скорость вращения и скорость доступа к информации невелики. Емкость

современных съемных дисков от 2 до 120 мегабайт, но проблема скорости обмена остается и для съемных дисков большой емкости.

Стационарные жесткие дисководы выполняются в герметично закрытом корпусе с несколькими металлическими дисками на одном валу, головки чтения - записи не соприкасаются с поверхностью носителей, плавая на воздушной подушке над ней. Поэтому скорость вращения дисков - несколько тысяч оборотов в минуту и соответственно сокращается время записи-чтения. Емкость жестких дисковых устройств измеряется сейчас в гигабайтах - на них хранятся все необходимые для работы компьютера программы и данные.

Магнитные ленточные запоминающие устройства для персональных компьютеров (так называемые стриммеры) имеют съемные носители большой емкости, внешне напоминающие аудиокассеты. Скорость чтения-записи на ленточных устройствах невелика, кроме того, для поиска на них нужной информации необходима перемотка в начало ленты, а запись возможна только в конец ленты (это объясняется невозможностью устранить вытяжку носителя и плавающей из за этого позицией нужного адреса).

Запись на магнитные носители осуществляется, естественно, тоже в двоичном числовом виде - намагниченный микроучасток соответствует значению 1, ненамагниченный - значению 0.

ДОРОЖКИ И СЕКТОРЫ. Для того, чтобы данные могли быть записаны на диск, его поверхность необходимо структурировать - т.е. разделить на секторы и дорожки. ***ДОРОЖКИ*** - это концентрические окружности на поверхности диска, на которые осуществляется запись. Ближайшей к внешнему краю диска дорожке присвоен номер 0, следующей за ней - 1 и т.д. Каждая дорожка разбивается на участки, называемые ***секторами***. Секторам также присваиваются номера. Первому сектору на дорожке присваивается номер 1, второму - 2 и т.д. Обычно сектор занимает 512 байт.

ФОРМАТИРОВАНИЕ ДИСКА. *Служебная программа операционной системы (утилита)* FORMAT.COM служит для разбиения диска на дорожки и секторы. При этом на диск записываются данные, необходимые при его дальнейшем использовании.

ЗАГРУЗОЧНАЯ ЗАПИСЬ СИСТЕМЫ. На нулевой стороне каждого диска с первого сектора нулевой дорожки FORMAT формирует особую запись - запись старта MS-DOS. Она состоит из:

1) таблицы, содержащей информацию о диске (о его разбиении на логические тома);
2) программы - загрузчика, загружающего саму операционную систему в оперативную память.

ФАЙЛЫ. Человеку удобно и привычно давать имена всем объектам, с которыми связывается его деятельность. Поэтому операционная система идет ему в этом навстречу и предусматривает присвоение имен блокам хранимой информации. Файлом называют именованную область информации в памяти компьютера, как правило, во внешней памяти и чаще всего на дисках.

КЛАСТЕР - это определенное количество секторов, выделяемых файлу при его создании и последующем расширении. Таким образом, файл - это цепочка кластеров, совсем не обязательно расположенных подряд друг за другом.

Одна из основных обязанностей ОС - обслуживание (хранение, создание, уничтожение и т.п.) файлов. Файл в MS-DOS - это набор взаимосвязанных данных, находящихся в известных системе местах на диске. При обработке файла он загружается в оперативную память машины. И загрузка в память, и хранение файлов входят в функции операционной системы.

РАСПОЗНАВАНИЕ (ИДЕНТИФИКАЦИЯ) ФАЙЛОВ. Каждый файл в MS-DOS должен иметь имя. Сложное имя состоит из основного имени длиной до 8 символов и необязательного расширения (суффикса) длиной до 3-х символов. По имени файл распознается операционной системой. Имена некоторых файлов (например, системных) заранее определены; они резервируются операционной системой. К системным файлам относятся файл IO.SYS (содержит вызываемые через механизм прерываний аппаратно-зависимые подпрограммы управления устройствами компьютера - так называемые драйверы BIOS), файл DOS.SYS (содержит вызываемые через механизм прерываний аппаратно-независимые подпрограммы верхнего уровня для управления вводом - выводом, файловой системой на дисках и пр.). Зарезервированными операционной системой являются также имена текстовых файлов дополнительного конфигурирования системы CONFIG.SYS и файла автозапуска системы AUTOEXEC.BAT. Оба этих файла обрабатываются при их наличии в процессе запуска системы.

СИСТЕМНЫЕ ФАЙЛЫ. Если в командной строке FORMAT указан параметр /s, то на форматируемый диск записываются копии системных файлов. В MS-DOS имеется три системных файла - IO.SYS, DOS.SYS и COMMAND.COM. Системные файлы хранятся на диске, с которого загружается операционная

система. Обычно (но не обязательно) IO.SYS размещается непосредственно после директория диска, а MSDOS.SYS записывается непосредственно после файла IO.SYS. MSDOS.SYS - это ядро операционной системы, содержащиеся в нем подпрограммы отбирают все запросы на сервисное обслуживание (например, открытие или чтение файла) и передают их в файл IO.SYS. Протокол взаимодействия MSDOS.SYS и IO.SYS идентичен протоколу взаимодействия двух операционных систем. Поэтому считается, что файл MSDOS.SYS независим от электронного оборудования (внешних устройств и самого компьютера).

Файл COMMAND.COM. Он служит связующим элементом (интерфейсом) между операционной системой и пользователем. Команды файла выводят на экран стандартный запрос системы, обрабатывают посланные с клавиатуры команды.

Имена остальных файлов назначаются пользователем. Обычно стараются придумать имя, отражающее назначение находящейся внутри файла информации. расширение используется для обозначения типа файла, например, текстовый txt или файл данных dat, или файл с текстом паскалевской программы .pas.

СПЕЦИАЛЬНЫЕ ФАЙЛЫ И ДРАЙВЕРЫ ВНЕШНИХ УСТРОЙСТВ. MS-DOS поддерживает 3 вида файлов: регулярные, специальные символьные файлы - устройства и директории. Регулярный дисковый файл состоит из блоков фиксированной длины - кластеров, каждый из которых в свою очередь состоит из фиксированного количества секторов стандартной длины 512 байт. При работе с регулярными файлами ОС использует специальную программу - драйвер блокового устройства, осуществляющему за одно обращение к нему перенос одного или более секторов (блоков).

Специальные символьные файлы - это по существу не файлы, а устройства посимвольного ввода-вывода, управляемые специальной программой - драйвером символьных устройств, но трактуемые ОС как файлы с уникальными зарезервированными именами:

CON	Клавиатура при вводе, экран при выводе
AUX (COM1)	1й порт адаптера последовательной связи
COM2,COM3,COM4	порты адаптеров последовательной связи
LPT1 (PRN)	Первый параллельный порт (принтер)
LPT2,LPT3	Параллельные порты
NUL	Фиктивное устройство при выводе "в никуда", при вводе сразу получаем "конец файла".

Доступ к драйверам блоковых устройств осуществляется по буквам шифра дисковода.

С точки зрения программиста и специальный символьный и регулярный файлы представляют собой цепочку байтов с определенным началом и концом. В файле ОС определяет "указатель чтения-записи" - это просто число, представляющее собой расстояние в байтах от начала файла до текущей позиции, изменяющееся при всех операциях переноса, имитируя автоматическое "перемещение указателя" вперед на перенесенное число байтов. Достижение конца файла определяется ОС, передающей в программу условие **EOF - *End Of File*** - конец файла. Переход за конец файла возможен только при записи или специальными подпрограммами ОС. Возможность установки указателя в произвольную позицию обеспечивает реализацию не только последовательного (байт за байтом), но и прямого доступа к данным. До любых переносов данных файл в ОС должен быть открыт - под этим подразумевается создание в памяти информационного блока о файле. Использованный файл может быть закрыт - т.е. разрушена необходимая для работы с ним информация. Открытие файлов-устройств выполняется ОС автоматически и никаких действий со стороны программиста не требуется.

ДИРЕКТОРИЙ И ЕГО СТРУКТУРА. Файловая система MS DOS имеет иерархическую древовидную структуру; в узлах дерева размещаются специальные файлы - директории. Самый первый в иерархии - корневой (root directory) размещается в фиксированном месте на диске. Состоит директорий из 32-байтовых элементов (для каждого содержащегося в нем файла или поддиректория) следующей структуры:

Имя файла	Расши-рение	Атрибуты файла	Резерв	Дата	Время	Номер 1-го кластера	Размер в байтах

Атрибут представляет собой побитно закодированную информацию о типе файла, при этом биты при установке в 1 означают:

0 файл только для чтения (read only);
1 скрытый файл (hidden);
2 системный файл (System);
3 элемент каталога есть метка тома;
4 это подкаталог и номер кластера - для кластера подкаталога;
5 архивный бит, устанавливается в 1 при создании файла и выполнении операций записи, что позволяет обнаружить изменявшиеся файлы;

При записи файла на диск его имя автоматически помещается в область памяти диска, называемую каталогом (или директорием). Имя файла должно быть уникальным в пределах директория. Другими словами, директорий диска не может включать два одинаковых имени (хотя, если имена сложные, то могут совпадать либо их простые имена, либо расширения). например, нельзя назвать два файла именем letters.txt - операционная система не будет знать, в каком случае вы обращаетесь к первому из них, а в каком - ко второму. Однако, можно пользоваться одним и тем же простым именем файла, указывая разные расширения.

Для образования имени файла нельзя использовать символы:

* + = [ ] : ; " ,. / ? пробел символ табуляции и управляющие символы

Строчные буквы в MS-DOS интерпретируются как заглавные, поэтому имя команды и параметры командной строки (в частности, имя файла) могут набираться как маленькими, так и большими буквами (можно вводить комбинацию из строчных и заглавных букв).

СПЕЦИФИКАЦИЯ ФАЙЛА. Чтобы операционная система могла обнаружить файл, ей нужно кроме имени знать полный путь доступа к файлу - диск, директорий и все вложенные в него поддиректории до того включительно, в котором находится файл. Для обозначения диска используется шифр устройства (дисковода), на котором он установлен. Шифр устройства представляет собой букву алфавита, за которой следует двоеточие. Первому дисководу в системе присваивается шифр "A:", второму - "B:". Первый дисковод для жесткого диска обычно помечается шифром "C:". Дисководы различаются по шифру устройства и называются: дисковод A, дисковод B и т.д. Директории именуются по тем же правилам, что и файлы, но их имена не могут иметь расширений. Шифр устройства, записанный в комбинации с именами директориев по маршруту доступа и имени файла, представляет собой спецификацию файла. Спецификация вводится в формате:

("шифр_устройства:\имя_директория\имя_поддиректория\простое_имя.расширение").

Например, спецификация файла с простым именем "instruct" и расширением "txt", находящегося на диске A в каталоге DOCUMENT, подкаталоге INSTRUCT, выглядит следующим образом:

"A:\DOCUMENT\INSTRUCT\instruct.txt".

ВНУТРЕННЯЯ СТРУКТУРА ДИСКА. MS DOS представляет дисковые устройства как логические тома, обозначенные кодом дисковода A, B, C и т. д. и содержащие необязательную метку тома, корневой каталог и любое число дополнительных каталогов и файлов. Каждый логический том ОС рассматривает как непрерывную последовательность логических секторов, начиная с сектора 0. Типичная карта логического тома выглядит следующим образом:

СЕКТОР ЗАГРУЗЧИКА
ЗАРЕЗЕРВИРОВАННАЯ ОБЛАСТЬ
ТАБЛИЦА РАЗМЕЩЕНИЯ ФАЙЛОВ FAT
ДОПОЛНИТЕЛЬНЫЕ КОПИИ FAT
КОРНЕВОЙ КАТАЛОГ
ОБЛАСТЬ ФАЙЛОВ

ЗАГРУЗОЧНЫЙ СЕКТОР. Логический сектор 0 называют загрузочным - он содержит всю важную информацию о диске. Первый байт всегда содержит команду безусловного перехода на программу начальной загрузки операционной системы. Далее идет 8-мибайтовое поле для названия фирмы - изготовителя оборудования. В байтах 0BH - 17H размещен блок параметров BIOS (BPB). Последний элемент загрузочного сектора - программа начальной загрузки ,которая считывается в оперативную память начальным загрузчиком ПЗУ, который переводит головки дисковода на 0-ю дорожку, читает 1-й физический сектор диска и передает управление считанной с диска программе загрузки.

ТАБЛИЦА РАЗМЕЩЕНИЯ ФАЙЛОВ (FAT - File Allocation Table). При создании или расширении файла ОС назначает ему группы секторов (кластеры) из файловой области, причем число секторов в кластере есть степень двойки. Число секторов на кластер записано в BPB (блоке параметров BIOS) по адресу 0DH в секторе загрузчика. Таблица размещения файлов разделена на поля размером 12 бит (если на диске меньше 4087 кластеров), 16 или 32 бита. Каждому кластеру на диске соответствует одно поле FAT. Первые 2 поля в FAT всегда зарезервированы. Первые 8 бит первой зарезервированной области FAT содержат копию байта описания носителя из сектора загрузки, 2-, 3-й и 4-й байты всегда содержат 0FFH. Остальные записи FAT описывают использование соответствующих кластеров диска, интерпретируя их следующим образом:

0 - кластер свободен
FF6H - зарезервированный кластер

FF7H - дефектный кластер
(FFFFF)FFFH - последний кластер файла
(XXXXX)XXX - следующий кластер файла

Каждая запись в каталоге, соответствующем файлу, включает номер первого назначенного ему кластера - точка входа в FAT. Начиная с этой точки каждая запись FAT содержит номер следующего кластера вплоть до метки последнего кластера. На каждом томе может поддерживаться 1 или 2 копии FAT. При расширении файла корректируются обе копии. При ошибках обращения к сектору ОС пытается читать вторую копию FAT. К примеру, программа CHKDSK при проверке целостности диска сравнивает копии FAT.

КОРНЕВОЙ КАТАЛОГ. Вслед за таблицей размещения файлов располагается корневой каталог диска. Как уже отмечалось, он содержит 32-байтовые записи, описывающие файлы, другие каталоги и необязательную метку тома. Запись, начинающаяся с E5H, свободна для повторного использования - она представляет удаленный файл или каталог. Запись, начинающаяся с 0-го байта , характеризует логический конец каталога - она сама и все последующие никогда не использовались.

Корневой каталог имеет ряд отличий от остальных. Его размер и положение фиксированы - их определяет программа форматирования. Если диск системный, то 2 первые записи корневого каталога описывают файлы расширения BIOS и ядра DOS.

ОБЛАСТЬ ФАЙЛОВ. Все остальное пространство диска после корневого каталога - совокупность кластеров, каждый из которых в зависимости от формата диска содержит 1 или более секторов. Каждому кластеру соответствует запись в FAT, описывающая его текущее состояние - свободен, зарезервирован, назначен файлу или поврежден. Первому кластеру назначен номер 2 и при расширении файла ОС просматривает FAT от последнего назначенного кластера. Все каталоги, кроме корневого, - это просто файлы специального типа. Пространство для них выделяется из файловой области, а их содержимое - это 32-байтовые записи, совпадающие по формату с записями корневого каталога и описывающие файлы и другие каталоги.

Записи, описывающие другие каталоги, имеют байт атрибутов с установленным в 1 4-м битом, нуль в поле длины файла. Поле 1-го кластера указывает на 1-й принадлежащий каталогу кластер. Остальные кластеры можно найти только прослеживанием их цепочки в FAT.

Все каталоги, кроме корневого, имеют 2 специальные записи каталога с именами "." и ".." (одна и две точки). ОС размещает эти записи при создании каталога и их нельзя удалить. Одна точка - это синоним текущего каталога, поле кластера этой записи указывает на кластер, в котором находится каталог. 2 точки - это синоним родительского каталога, ее поле кластера указывает на первый кластер родительского каталога. Если родитель - корневой каталог, то поле кластера записи с 2-мя точками содержит 0.

3.3. Средства MS DOS для доступа к файлам.

Открытие файлов выполняет функция FH=3dh прерывания 21h. Пара регистров DS:DX должна получить адрес ASCII-строки со спецификацией открываемого файла, полурегистр AL - режим открытия файла (для чтения -0, для записи 1). После выполнения обработчика прерывания в регистре AX будет возвращено целое число, которое называют префиксом (handle) или дескриптором, на который в дальнейшем ссылаются при всех операциях с файлом. Для закрытия файла надо выполнить функцию AH=3ch с полученным при открытии префиксом в регистре AX - он будет освобожден.

Префикс - это порядковый номер элемента в таблице открытых файлов, а значением элемента этой таблицы является порядковый номер записи в массиве описаний открытых файлов, каждая запись которого содержит флаги режима доступа к файлу, текущее значение указателя чтения - записи и другая необходимая для работы с файлом информация.

Таблица открытых файлов первоначально размещается в PSP по смещению 18h, где ей отведено 20 байт. При необходимости открыть более 20 файлов вместе с 5-ю автоматически открываемыми файлами - устройствами она переписывается в свободную оперативную память, а ее адрес хранится в PSP по смещению 32h.

Функция AH=3Fh прерывания 21h используется для чтения из файла или устройства, функция 40h для записи. Физически перенос информации между оперативной и дисковой памятью может осуществляться только блоками, кратными 512 байт за одно обращение, поэтому MS DOS осуществляет обмен через внутренний буфер, из которого программа получает запрошенное число байтов. Если оно меньше считанного, то следующая порция выдается из буфера без обращения к диску. Аналогичная картина при записи - данные попадают сначала в буфер, а перенос на диск осуществляется либо при заполнении

буфера, либо при закрытии файла, либо по запросу на очистку (флэширование) буфера из прикладной программы.

3.4. Внутренние команды DOS.

Приведем справочник по наиболее употребительным внутренним командам DOS. Вам рекомендуется поупражняться в использовании этих команд на компьютере.

КОМАНДА	НАЗНАЧЕНИЕ, ПАРАМЕТРЫ
BREAK	ФОРМАТ: BREAK=ON или BREAK=OFF¦ При ON проверка необходимости прерывания текущей программы производится при каждом обращении к подпрограммам ОС,а при OFF - только при вводе-выводе. Можно включить ее в файл CONFIG.SYS
BUFFERS	ФОРМАТ: BUFFERS=xx.(xx-количество файловых буферов от 1 до 99). Используется только в CONFIG.SYS
CD (CHDIR)	ФОРМАТ: CD [диск\путь\] Изменить рабочий директорий или вывести на экран рабочий путь по дереву директориев (без параметра)
CLS	ФОРМАТ: CLS Очищает экран
COPY	ФОРМАТ: COPY что куда /v/a или /b Копирование файлов. Объединение нескольких файлов в один. Обмен данными между периферийными устройствами и файлами ПРИМЕРЫ: copy File1 copy File1+File2 b:File3 copy con File4.txt (копировать текст с экрана в файл) /a-текстовые файлы, /b-двоичные /v-с проверкой правильности
DATE	ФОРМАТ: DATE [mm-dd-yy] Выводит и ввод даты.
DEL (ERASE)	ФОРМАТ: DEL[d:][path][Filename[.ext] Уничтожение дисковых файлов
DEVICE	ФОРМАТ: DEVICE=[d:][path]Filename[.ext] Включает в операционную систему драйвер периферийного устройства. Используется в CONFIG.SYS ПРИМЕР: device=ansi.sys

DIR	ФОРМАТ: DIR [d:][Filename[.ext]][/P][/W] /p-для постраничного вывода /w-во весь экран Вывод содержимого директория ПРИМЕРЫ: dir dir b: dir b:\subdir1*.doc/w
ECHO	ФОРМАТ: ECHO [ON|OFF|сообщение] Управляет выдачей на экран команд командного файла в процессе его выполнения. ПРИМЕРЫ: echo on echo Off echo Передай привет другу Васе
FILES	ФОРМАТ: FILES=xx Задает объем памяти, выделяемый для ссылок на управляющую запись файла в файле конфигурации ПРИМЕР: Files=40
FOR	ФОРМАТ:FOR %% variable IN (параметры) DO command Обеспечивает повторное выполнение команды для каждого из группы заданных параметров ПРИМЕР: for %%a IN (File1 File2 File3) DO del %%a
GOTO	ФОРМАТ: GOTO label Переход на метку, обозначенную именем с двоеточием после него ПРИМЕР: goto four
IF	ФОРМАТ: IF [NOT] condition command Обеспечивает выполнение команды при выполнении заданного условия ПРИМЕРЫ: if exist someFile.dat type someFile.dat if %1==roses goto roses if not exist File.bak copy File.txt File.bak
MD (MKDIR)	ФОРМАТ: MKDIR [d:]path Организация поддиректория ¦ ПРИМЕРЫ: mkdir\write md b:\programs\business
PATH	ФОРМАТ: PATH [[d:]path[[;[d:]path]...]] Определяет спецификатор пути поддиректория или файла ПРИМЕРЫ: path \program1\business path b:\program2\write1;b:\program2\write2
PAUSE	ФОРМАТ: PAUSE [comment] Прерывание выполнения командного файла ПРИМЕР: pause
PROMPT	ФОРМАТ: PROMPT [text или метапеременные] Определяет изображение системного запроса Метапеременные:

	$t Время по таймеру $d Дата, загруженная в память $p Рабочий директорий рабочего диска. $v Работающая версия MS-DOS $n Шифр рабочего диска $g Символ ">" $l Символ "<" $b Символ "|" $q Символ "=" $$ Символ $
RENAME	ФОРМАТ:RENAME [d:][path]Filename[.ext]Filename[.ext] REN [d:][path]Filename[.ext]Filename[.ext] Переименование файла ПРИМЕРЫ: rename File1 File2 ren newFile.txt oldFile.txt
RD (RMDIR)	ФОРМАТ: RMDIR [d:]path Удаление поддиректория ПРИМЕРЫ: rmdir \write rd b:\programs\business
SET	ФОРМАТ: SET [name=[parameter]] Вводит указанную символьную переменную в операционную среду MS-DOS ПРИМЕР: set xyz=abc
SHELL	ФОРМАТ: SHELL=[d:][path]Filename[.ext] Загружает командный процессор SHELL может использоваться только в качестве оператора файла CONFIG.SYS ПРИМЕР: shell=custom.com
STACKS	ФОРМАТ: STACKS=n,s (количество и размер кадров стека для обработки прерываний от оборудования ПРИМЕР: stacks=12,256
TIME	ФОРМАТ: TIME [bb:mm:ss:xx] Выводит значение текущего времени (по таймеру). Корректирует это значение ПРИМЕРЫ: time time 12:20
TYPE	ФОРМАТ: TYPE [d:][path]Filename[.ext] Выдача на экран содержимого файла ПРИМЕР: type b:letter.txt
VER	Печатает имя и версию операционной системы

3.5. Командные файлы.

Командный файл - это текстовый файл (в коде ASCII), состоящий из группы команд MS-DOS. Правила идентификации

командных файлов совпадают с общими правилами идентификации файлов. Единственное исключение - командный файл всегда записывается на диск с расширением ".BAT" (BATch). Обратиться к командному файлу крайне просто. Набирается команда старта - имя файла, и нажимается клавиша Enter. После введения команды файл выбирается из рабочего директория указанного или рабочего диска. Если в рабочем директории его нет, то поиск файла будет производиться в директориях, описанных командой PATH. При нахождении файла первая из его команд загружается в память, отображается на экране и выполняется. Этот процесс повторяется последовательно для всех команд файла (от первой до последней команды). Выполнение командного файла можно прервать в любой момент, нажав на клавиши Ctrl-Break.

Для вызова командного файла из выполняющегося командного файла с возвратом в последний включают команду

CALL имя .bat-файла

3.6. Замещаемые параметры или шаблоны имен файлов.

Большинство команд MS-DOS допускают использование замещаемых символов в именах файлов и расширениях - в тех случаях, когда команда относится к группе файлов, как например часто бывает при копировании. Эти параметры могут включать один или несколько замещаемых символов ? - для замещения одного символа в имени или * для замещения группы символов.

3.7. Выполняемые файлы.

Все прикладные программы MS-DOS обрабатываются при их создании программой редактирования связей LINK.EXE, которая оформляет программу в виде выполняемого файла, определяет месторасположение отдельных его частей в памяти машины и устанавливает связи между этими частями. Затем адресная информация записывается в заголовок в начале выполняемого файла. Всем обработанным редактором файлам присваивается расширение ".EXE". Если файл типа EXE удовлетворяет трем следующим требованиям, то его можно преобразовать в файл типа COM:

- файл (программа и данные) занимает менее 64 К памяти;
- машинный код, данные и стек программы помещаются в одном и том же сегменте;

– короткий адрес первой команды программы равен 100H (адрес от начала файла).

Для преобразования файла типа EXE в файл типа COM служит утилита EXE2BIN. Файлы типа COM не имеют заголовка.

При загрузке файлы типа COM всегда располагаются, начиная с адреса 100H от начала сегмента программы. Первые 100H байтов сегмента отводятся для psp. Адрес сегмента программы записывается во все четыре сегментных регистра. Значение 100H - в регистр IP. Регистр SP содержит адрес верхней границы сегмента программы. Затем в последние 2 байта стека помещается значение 00H и управление передается команде, находящейся по адресу CS:100. Программа начинает выполняться.

При загрузке файла типа EXE, заголовок файла помещается в сегмент программы, начиная с адреса 100H. Оставшаяся часть файла располагается в соответствии с данными заголовка. Регистры CS, IP, SS и SP заполняются также соответственно данным заголовка. Регистры DS и ES содержат длинный адрес psp. Затем управление передается команде, находящейся по адресу CS:100. Программа начинает выполняться.

4. Алгоритмизация и программирование.

4.1. Понятие алгоритма и автомата (исполнителя алгоритма).

Термин "алгоритм" произошел от имени средневекового узбекского математика Аль-Хорезми, который еще в 9-м веке предложил правила выполнения 4-х арифметических действий в десятичной системе счисления.

Понятие алгоритма - одно из основных в математике и информатике. Под алгоритмом понимают точное предписание (инструкцию) для выполнения в определенной последовательности некоторой системы операций для решения всех задач данного типа. Это определение не является строгим математическим, это скорее одно из толкований термина "алгоритм", поясняющее его смысл.

В математике тот или иной класс задач считается решенным, если для его решения установлен алгоритм. В алгебре, например, установлены алгоритмы для определения количества корней алгебраического уравнения и вычисления их значений с заданной точностью по известным коэффициентам уравнения.

Алгоритмизация не является только прерогативой математики. в живой природе всякой массовой целенаправленной деятельности сопутствует заранее созданный алгоритм - по определенному алгоритму осуществляется массовое изготовление обуви и пошив одежды, сборка автомобилей на конвейере, выпечка хлеба и плавка металла; таким образом, алгоритм можно трактовать как технологическую инструкцию из отдельных предписаний, выполнение которых в заданной последовательности приводит к заранее предвидимому результату. Сама жизнь на земле зародилась тогда, когда на молекулярном уровне был записан алгоритм жизненного цикла белковой клетки от рождения до смерти. Все живые организмы осуществляют свой жизненный цикл по такому созданному природой алгоритму с некоторыми отклонениями, вызванными влиянием окружающей среды. По существу алгоритм - это один из видов информации, являющейся внутренней для действующего объекта и используемой для обработки внешней, принимаемой по каналам связи, информации. Мы будем заниматься алгоритмами искусственного типа, создаваемыми с участием

человека, преимущественно в области математики, не забывая при этом, что это - капля в безбрежном море алгоритмов.

Основные свойства, которыми должен обладать любой алгоритм:

1.Массовость - алгоритм должен функционировать с различными исходными данными, то есть решать не индивидуальную задачу, а серию однотипных задач. Другими словами алгоритм должен рассматриваться как метод решения определенного класса задач, пригодный для использования в некотором диапазоне изменения исходных данных.

2.Детерминированность - алгоритм должен содержать конечное число предписаний, не допускающих произвола исполнителя, не оставляющих исполнителю свободы выбора. Многократное повторение алгоритма с одинаковыми исходными данными должно приводить к одному и тому же результату.

Создание алгоритма для решения задач определенного типа связано обычно с тонкими и сложными рассуждениями, требующими высокой квалификации. Но с того момента, когда алгоритм уже создан, процесс решения соответствующих задач должен быть таковым, чтобы его мог выполнить исполнитель, не имеющий ни малейшего понятия о сущности решаемой задачи - требуется только, чтобы он умел выполнять те элементарные операции, на которые разложен процесс решения задачи. С другой стороны, разработчик алгоритма должен учитывать тот арсенал операций, которым владеет предполагаемый исполнитель - иначе некому будет осуществить исполнение. Исполнитель алгоритма - его неотъемлемая часть; "квалификация" исполнителя должна учитываться составителем при выборе инструкций - все они должны выбираться из списка возможностей исполнителя, иначе на стадии выполнения произойдет отказ, алгоритм окажется невыполнимым, перестанет быть алгоритмом. Алгоритмы раскроя металла или тканей, сборки или сварки, вычисления корней уравнений или поиска экстремума функции, начисления зарплаты или расчета затрат на строительство всегда ориентированы на определенную квалификацию исполнителей. Неразрывное единство алгоритма и его исполнителя выпукло отражено в названии одной из книг Н. Винера "Акционерное общество Творец и Робот" - оба компонента нерасторжимы, не в состоянии дать продукт друг без друга.

Одно из возможных определений исполнителя:

Исполнитель - это человек или коллектив людей, вооруженных набором инструментов и обученный выполнению некоторой совокупности операций в заданной последовательности, или автоматическое устройство

(электронное, электромеханическое и т.п.), изготовленное таким образом, что, будучи включенным в работу, выполняет заданную последовательность операций над некоторым исходным продуктом, преобразуя его в заданный конечный продукт. Другими словами, исполнитель - это автомат, запрограммированный на выполнение заданной последовательности операций, действий. Запуск исполнителя на выполнение алгоритма называют вызовом алгоритма. Еще одно важное свойство алгоритмов и исполнителей - это их иерархическая структура. Алгоритм решения сложной задачи разбивается на ряд более простых - исполнитель для решения систем уравнений может вызывать исполнителей нижнего уровня для обращения матриц или вычисления определенных интегралов, эти исполнители в свою очередь могут вызывать исполнителей для сортировки данных и где - то на нижнем уровне располагаются алгоритмы и исполнители для арифметических действий и логических операций, составляющих основу для всех процедур высокого уровня.

4.2. Методы описания алгоритмов.

Разработанный алгоритм (метод решения определенного класса задач) должен быть как-то сохранен для использования другими алгоритмами, для последующего перевода в другие формы записи - другими словами его существование начнется с того момента как он будет записан.

Наиболее простой формой записи алгоритмов является естественный язык. Практически все алгоритмы проходят стадию формулировки на естественном языке - даже в тех случаях, когда физически алгоритм сразу записывается глубоко формализованными способами, его словесная формулировка присутствует в сознании разработчика пусть даже не нанесенная на бумагу или другой носитель информации.

В качестве дополнительного примера рассмотрим запись алгоритма вычисления числа Фибоначчи с заданным номером; напомним, что члены ряда чисел Фибоначчи, начиная с третьего, равны сумме 2-х предыдущих в последовательности.

1.Получить у пользователя (заказчика) номер искомого числа Фибоначчи и запомнить его в переменной по имени N.

2.Присвоить 3-м переменным, например с именами F1, F2 и F3, значения соответственно 1, 1, и 2. Переменной n присвоим значение текущего номера, вначале - 3. Перейти к операции 3.

3.Сравнить n и N. Перейти к операции 4.

4.Если n меньше N, F1 присвоить значение F2, F2 - значение F3, а F3 - сумму значений F1 и F2. n увеличить на 1.

Перейти к операции 3. Иначе искомое число Фибоначчи равно F3. Вычисления прекратить.

В этом примере алгоритм получает извне на стадии выполнения одно число - номер искомого числа Фибоначчи. Алгоритмы, в которых основную роль играют арифметические действия, называют численными и задаются на первой стадии разработки в виде словесных предписаний или разного рода формул и схем. Эти алгоритмы получили широкое распространение потому, что к 4-м арифметическим действиям можно свести другие более сложные операции интегрирования, дифференцирования функций, знакомый вам алгоритм вычисления корня квадратного и т д.

Алгоритм всегда должен быть составлен в достаточно общем виде - конкретные данные он получает только на стадии выполнения - именно так обеспечивается его важнейшее свойство - массовость. Поэтому все компьютерные алгоритмы должны содержать процедуры получения данных для своей работы.

4.3. Управление операциями в алгоритмах.

Уже в приведенных простеньких примерах видно, что для алгоритма характерно наличие выбора, какую операцию выполнять на текущей стадии вычислений. Предписания, связанные с выбором очередной операции в зависимости от тех или иных условий, называют

УПРАВЛЯЮЩИМИ СТРУКТУРАМИ АЛГОРИТМОВ

В 1977 году математики Бем и Якопини доказали, что алгоритмы сколь угодно сложной структуры могут быть реализованы с использованием всего 3-х управляющих структур :

- Последовательное выполнение операций;
- Ветвление алгоритма на группы операций в зависимости от выполнения некоторых условий;
- Циклическое многократное выполнение группы операций до выполнения некоторого условия, формируемого в процессе вычислений.

Все 3 указанных фактора присущи приведенным примерам. Заранее неизвестно, сколько раз придется выполнить переприсвоение значений и сложение в алгоритме вычисления числа Фибоначчи - это определяется его номером. Точно так же неизвестно, сколько вычитаний и переприсвоений придется осуществить для определения НОД двух чисел - это

зависит от значений самих чисел, которые при создании алгоритма не определены и задаются на стадии выполнения.

Краткое название управляющих структур - **СЛЕДОВАНИЕ, ВЕТВЛЕНИЕ, ЦИКЛЫ,** а соответствующие им операторы в записи алгоритмов называются **УСЛОВНЫМИ ОПЕРАТОРАМИ И ОПЕРАТОРАМИ ЦИКЛОВ** (следование не имеет специального оператора и выражается в записи просто последовательной записью инструкций).

Условные операторы в наших примерах звучат как **ЕСЛИ <УСЛОВИЕ ВЫПОЛНЕНО>** последовательность операций **ИНАЧЕ** другая последовательность операций.

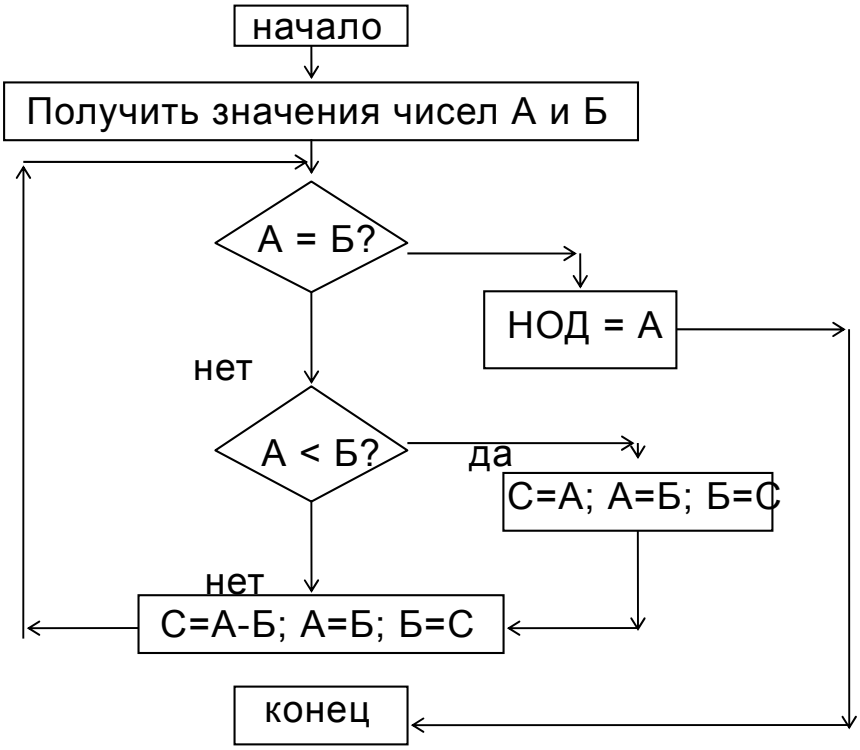
Операторы циклов в описаниях на естественном языке мы формулировали словами "Пока истинно некоторое условие - повторять Заданные действия"," Повторять бесконечно" и пр.

4.4. Запись алгоритмов в виде блок-схем.

Иногда используют графически - текстовую форму записи алгоритмов. От чисто текстовой на естественном языке она отличается следующим:

- последовательно выполняемые операции записываются текстом в обрамленных рамками прямоугольниках;
- условия ветвления алгоритмов записываются внутри ромбовидных фигур;
- прямоугольники и ромбы соединяются линиями со стрелками, указывающими направления перехода от одной операции к другой.

Алгоритм вычисления НОД в виде блок-схемы выглядит примерно так:



Блок-схемы используют для наглядности обычно начинающие алгоритмисты, при создании алгоритмов значительных объемов их рассмотрение становится неудобным, хотя иногда может быть полезным при выявлении "потерянных" ветвлений.

Существует разумеется много других способов графического начертания алгоритмов с другими обозначениями, но мы не будем их здесь рассматривать.

4.5. Языки программирования и трансляторы.

Программирование прямым числовым кодированием процессорных команд очень трудоемкое занятие, поэтому попытки приблизить текст программы к естественному языку с автоматическим переводом этого текста в числовые коды команд с помощью специальных программ-трансляторов вполне естественны.

Первым шагом к облегчению создания программ было присвоение каждой процессорной команде имени, доступного для понимания ее действия, как правило в виде аббревиатуры естественного языка. Перевод в машинные коды в этом случае осуществлялся по таблицам и язык назывался языком мнемонического кодирования команд или мнемокодом. В дальнейшем стали именовать уже не только отдельные команды, но их совокупности для выполнения специализированных составных операций. Машинно-ориентированные языки Ассемблера поставляются и сейчас со

всеми инструментальными системами разработки программ, но даже в системном программировании при разработке операционных систем они используются в небольшом объеме из-за низкой производительности труда.

В прикладном программировании используют сегодня высокоавтоматизированные инструментальные системы разработки программ, основной частью которых является программа - транслятор, способная осуществлять перевод в машинные коды текстов алгоритмов, записанных на алгоритмических языках высокого уровня, для которых характерно максимально возможное приближение к естественному языку - в силу значительного опережения США в этой области, таким языком оказался английский. В настоящее время существует большое количество формализованных алгоритмических языков как общего назначения, так и проблемно - ориентированных, предназначенных для отдельных областей применения. Языки общего назначения наиболее универсальны, но могут реализовывать различные стили (парадигмы) программирования. Так, языки SmallTalk и Simula реализуют так называемый объектно-ориентированный стиль, C++ и Паскаль - являются гибридными и поддерживают как процедурный, так и объектно-ориентированный стили, язык Лисп - стиль логического программирования и т. д. На протяжении ряда лет осуществлялось сближение трансляторов с различных языков по ассортименту предоставляемых программистам услуг, но специфические для каждого языка синтаксические и структурные различия сохраняются.

Современному программирующему пользователю ЭВМ приходится быть "полиглотом", то есть знать несколько языков, чтобы иметь возможность выбирать подходящий для конкретной задачи инструмент разработки. Это не так уж сложно в рамках одного и того же стиля программирования; переходы между различными стилями могут быть более затрудненными, но тоже вполне осуществимы.

4.6. Типы трансляторов.

Основная задача транслятора с любого формализованного алгоритмического языка - перевести текст алгоритма в последовательность 2-ичных числовых кодов команд процессора. Существует 2 основных типа транслирующих программ - интерпретаторы и компиляторы.

Интерпретаторы анализируют каждую отдельную строку текста алгоритма и при отсутствии в ней синтаксических ошибок (несоответствий принятым в языке правилам составления

текста алгоритмов, наличии в тексте "незнакомых" транслятору слов и пр.) переводят записанные в строке операторы языка в машинные команды и тут же их выполняют. Алгоритмы, записанные для интерпретатора, не могут выполняться в отрыве от этого интерпретатора под управлением только операционной системы, так как результат трансляции в машинные коды не сохраняется, он теряется сразу после выполнения очередного оператора.

Компиляторы обрабатывают и переводят в машинные числовые коды весь текст алгоритма, сохраняя (при отсутствии синтаксических и орфографических ошибок) результат компиляции в виде так называемого объектного файла. Этот файл не может еще быть вызван для выполнения, так как в нем адреса всех данных и процедур носят условный характер, они не согласованы между собой. Для превращения объектного модуля в исполнимый необходимо осуществить редактирование внутрипрограммных адресных ссылок в соответствии с требованиями операционной системы и для этой работы в составе ОС есть программа - редактор адресных ссылок LINK.EXE (она поставляется также дополнительно к любому компилятору). Эта программа обрабатывает объектный файл и другие, например библиотечные объектные файлы и компонует все это в исполняемый файл с расширением. EXE. Этот файл уже может быть оторван от среды разработки и выполняться под управлением операционной системы - компилятор ему уже не нужен.

Следует понимать, что отсутствие синтаксических ошибок не означает отсутствие алгоритмических - транслятор не может знать, что мы с вами хотели запрограммировать, он не может анализировать смысл алгоритма, а занимается только чисто формальной разборкой текста и проверкой его соответствия принятым в языке правилам правописания. Требования к синтаксису и орфографии при Записи алгоритмов на транслируемых алгоритмических языках должны выполняться неукоснительно - в отличие от человека, способного догадаться, что слово "пиисят" означает "пятьдесят", современные трансляторы "догадываться" не умеют и ошибка даже в одном символе вызовет недоумение транслятора, сопровождаемое диагностическим сообщением и отказом от окончательного завершения его работы.

4.7. Требования трансляторов к текстам программ.

Каждая транслирующая программа, чтобы не стать "необъятной", не может допустить свободного изложения алгоритма и вынуждена предъявить определенные требования к структуре его текста. Из каких основных частей состоит текст алгоритма?

Прежде всего транслятор должен пропустить "беллетристику", которая не должна переводиться в машинные коды и не относится к самому транслятору, - это так называемые комментарии, без которых нельзя обойтись ни в одной программе. В них содержатся сведения о классе задач, обслуживаемых алгоритмом, словесное описание метода решения, различные ограничения, способ использования, сведения о разработчике и многое другое. Говорят, что программы составляются для людей, которым предстоит их корректировать и совершенствовать, исправлять в них алгоритмические ошибки и устранять недоработки, поэтому роль комментариев трудно переоценить. Однако транслятору они не нужны, поэтому они как правило выделяются обусловленными для данного языка символами в начале и в конце и транслятор, встретив символ начала комментария, пропускает все символы, пока не встретит символ окончания комментария. К сожалению, во всех языках символы-ограничители комментариев разные, эта часть не унифицирована, как и многие другие элементы синтаксиса алгоритмических языков - это просто свидетельство незрелости самой отрасли программирования.

Далее транслятор должен выделить директивы управления процессом трансляции - это фрагменты текста, не являющиеся составными частями алгоритма решения задачи, а только настраивающие сам транслятор на определенный режим работы или выполнение определенных действий, например: печать заголовка при выводе листинга программы, установка размеров страниц при печати программного текста, включение перед трансляцией текстов алгоритмов из других файлов и пр.

Транслятор должен иметь возможность определить область текста алгоритма, в которой записаны операции обработки данных; кроме того, при вызове программы на выполнение в регистры процессора необходимо поместить адрес первой команды, с которой начнется выполнение алгоритма; поэтому все компиляторы требуют, чтобы программа состояла из одной или нескольких подпрограмм, одна из

которых должна быть обозначена как главная, с первого оператора которой начнется выполнение алгоритма в целом.

Все операторы обработки данных размещаются при этом внутри подпрограмм, тела которых ограничиваются специальными символами или словами. Так в ассемблере это слова ИМЯ и PROC в начале и ИМЯ и ENDP или END в конце, в Си это открытая и закрытая фигурные скобки после имени подпрограммы со списком параметров в круглых скобках, в Паскале это слова `BEGIN` и `END` после имени подпрограммы со списком параметров в круглых скобках и разделов объявлений типов, переменных и констант. Главная подпрограмма выделяется в ассемблере по закрывающей фразе ИМЯ_ПОДПРОГРАММЫ `END`, в Си она имеет фиксированное имя `main()`, в Паскале она выделяется точкой после слова `END` (все другие подпрограммы заканчиваются словом `END`; с точкой с запятой, т.е. символом "семиколон").

Следующий вопрос формализации программного текста - размещение данных в памяти. Любая программа оперирует с некоторыми данными - исходными (которые задает пользователь), результатами промежуточных вычислений, конечными результатами - все эти данные должны быть размещены в оперативной памяти компьютера. Так как компьютер - техническое устройство, размер памяти, отводимой под каждый элемент данных всегда ограничен определенным количеством байтов (минимально адресуемых единиц памяти). Оперативная память - быстро расходуемый и дефицитный ресурс компьютера, каждый элемент данных для возможности работы с ним должен иметь адрес, поэтому каждый транслятор требует, чтобы все элементы данных имели четко обозначенный тип - именно тип определяет количество байтов, занимаемых в памяти тем или иным элементом данных. Данные могут располагаться в одном из 3-х участков оперативной памяти, распределяемых программе операционной системой: в сегменте данных программы (глобальные данные), в программном стеке для временного хранения на период выполнения специализированной подпрограммы или в динамически распределяемой области ("куче", хипе - heap). Память в сегменте данных распределяется компилятором на стадии компиляции текста программы (если размер необходимой памяти известен при составлении текста), а в куче выделяется по запросу программы на этапе ее выполнения. Для данных, размер которых при различных запусках программы может изменяться пользователем и на стадии составления алгоритма неизвестен, лучше всего запрашивать данные в куче после выяснения их размера. Любой элемент данных во всех

языках программирования может иметь имя и обязательно должен иметь указанный тип, определяющий размер и допустимые с этим типом операции.

4.8. Выбор языка высокого уровня для начального обучения.

"Учебный" язык должен обладать рядом необходимых качеств, обеспечивающих легкость восприятия и приемлемую скорость освоения методов составления текстов программ:

⇒как можно более короткий словарь - чтобы основные усилия обучаемого сосредоточить на методической стороне программирования;

⇒словарь языка должен быть "англоязычным" - т.е. базироваться на языке, ставшим де-факто языком межнационального общения; обучение на национальных "суррогатах" - напрасная трата времени;

⇒учебный язык не должен содержать абстрактных типов данных, не совместимых в операциях с числовыми типами, например "символьный тип", "строковый тип", "булевский тип" и пр.; поддерживаемые в языке типы данных должны прямо отражать способность компьютера обрабатывать только числа и не вносить ненужной путаницы в сознание обучаемого между числовыми кодами и способами их трактовки и отображения;

⇒язык должен обладать функциональной полнотой профессиональных языков программирования.

Сегодня есть только один такой язык общего назначения - это язык Си (и его надмножество C++), сделанный в свое время инженером Деннисом Ритчи не для коммерческого распространения, а "для себя" и, наверное, поэтому получившийся столь удачным и для профессионального применения, и для начального обучения. Найдется много оппонентов этому утверждению среди сторонников традиционных методов обучения на школьном русскоязычном алгоритмическом языке или на GW-BASIC, или в лучшем случае на Паскале. Обычно среди этих оппонентов нет знающих Си и работающих на нем, поэтому дискуссия стаёт бессмысленной - отмечу только, что на Си работают профессионалы, а они выбирают для своей работы наиболее простые в освоении и эффективные в работе инструменты.

4.9. Описание некоторых алгоритмов для последующего программирования на языках Си и Паскаль.

4.9.1. Простые числа.

Задача: вывести на экран все простые числа, не превышающие заданного в командной строке.

Тривиальный алгоритм решения состоит в том, что мы берем поочередно все нечетные числа и, если число не делится без остатка на все предыдущие нечетные и еще не превышает заданного предела, выводим его на экран. Уменьшение количества выполняемых операций достигается тем, что в качестве тестирующих делителей можно использовать числа, не превышающие корня квадратного из делимого (доказательство этого положения проработайте с вашим преподавателем математики).

4.9.2. Сравнение 2-х строк, заданных указателями на их начала и заканчивающихся нулевыми байтами.

Осуществляется эта операция поэлементно; договоримся, что при абсолютном равенстве длин и содержимого 2-х строк результат сравнения будет число 0, если первая строка больше второй - положительное число, если первая меньше второй - отрицательное число. Тогда алгоритм может быть таким: двигаясь вдоль обеих сравниваемых строк (наращиванием указателей на 1), будем определять разность кодов символов с одинаковыми номерами и прекратим работу или по ненулевой разности или по нулевой сумме кодов символов - последний случай соответствует равной длине и содержимому строк.

4.9.3. Выделение и преобразование в целое цифровой подстроки в строке, заданной указателем на начало и завершающейся нулем.

Алгоритм таков: пока значение по текущему значению указателя не нуль (не конец строки) и не цифра - наращиваем его; оказавшись на цифре: пока значение по текущему значению указателя - цифра, определяем числовое значение цифры (вычитанием из ее кода кода цифры '0') и прибавляем его к умноженному на 10 текущему значению числа (перед началом выполнения алгоритма значение числа равно 0).

Оформите эту работу в теле функции, получающей в виде параметра указатель на строку (пока мы преобразуем в число только одну цифровую подстроку).

Программирование рекурсивных алгоритмов.

Вызов подпрограммы из другой подпрограммы - обычный прием в процедурном программировании. При составлении алгоритмов часто бывает удобно на некоторой фазе обработки данных из тела текущего алгоритма вызвать на выполнение "самого себя", уже с измененными аргументами - такие алгоритмы называются рекурсивными. Различают прямую или непосредственную рекурсию (вызов алгоритмом самого себя) и опосредованную рекурсию, при которой из алгоритма 'А' вызывается алгоритм 'Б', а он в свою очередь вызывает алгоритм 'А'.

4.9.4. Рекурсии - задача о Ханойских башнях.

Классическим примером использования рекурсии для построения алгоритма является задача о Ханойских башнях: на одном из 3-х стержней надето N дисков с убывающими к вершине радиусами; необходимо переложить диски на другой стержень в том же порядке с использованием 3-го как промежуточного, но нельзя класть больший диск на меньший.

Обозначив стержни через a, b, c и полагая? что задан перенос с a на b, запишем алгоритм на псевдоязыке:

если N==1 перенести верхний диск с a на b

иначе вызвать этот же алгоритм для переноса N-1 дисков с a на c,

вызвать его же алгоритм для переноса одного диска с a на b,

вызвать этот же алгоритм для переноса N-1 дисков с c на b.

Решение этой задачи без использования рекурсии может быть таким:

Стержень с наименьшим диском запомним как a;

В бесконечном цикле повторяем действия:

{

Если количество стержней на b или c равно N - задача решена и мы выходим из бесконечного цикла;

1-й минимальный переложить по часовой стрелке и перезапомнить новый стержень с этим диском;

Для двух оставшихся стержней (кроме того, на котором теперь лежит самый маленький диск) переложить меньший верхний на больший;

}

Для реализации нерекурсивного алгоритма нам понадобятся следующие переменные:

atop, btop, ctop - для номеров верхних дисков на каждом из стержней

min - для обозначения стержня с самым маленьким диском Размеры дисков придется перенумеровать от 0 до N, причем номер N - несуществующий самый большой. Для каждого стержня придется организовать в памяти массивы дисков a, b, c, в исходном состоянии для массива a atop=0, a[atop]=0;...a[n]=n; для остальных стержней btop=ctop=n; b[top]=c[top]=n;

В нашем нерекурсивном алгоритме диски будут переложены со стержня 'a' на 'b' при нечетном их числе и на 'c' при четном. Добавив в программу анализ на "чет-нечет" и изменяя направление перекладывания наименьшего диска, вы можете добиться укладки на заданный стержень.

4.9.5. Рекурсии - программный генератор перестановок N попарно различных чисел.

Если предположить, что подлежащие перестановкам числа размещены в массиве, то перестановки чисел можно заменить перестановкой индексов массива. Т. о. для решения задачи достаточно знать количество чисел, остальное - это вопросы ввода-вывода. Количество чисел мы традиционно возьмем из командной строки, а печать элементов произвольных массивов по перестановкам их индексов оставляем на самостоятельную работу.

Рекурсивный вариант генератора перестановок выглядит так:

если N>1

{вызываем алгоритм для N-1 чисел;

начиная с последней пары чисел и до первой (от i=N-1 до 1)

повторяем:

{переставляем значения в паре чисел с номерами i-1 и N-1;

вызываем алгоритм для N-1 чисел;

переставляем значения в паре чисел с номерами i-1 и N-1;

}

}

иначе печатаем последовательность чисел как перестановку.

По приведенной в разделе 6.11.3 программе прорисуйте на бумаге по шагам его алгоритм например для 3-х чисел (индексы 0,1,2) - он станет более понятен. Здесь мы имеем дело с вариантом, когда рекурсивный вариант воспринимается не без усилий.

Один из возможных нерекурсивных вариантов основан на построении упорядоченной последовательности перестановок и генерации на этой основе каждой последующей перестановки из предыдущей. Чтобы по данной перестановке (а начинаем мы с последовательности 0, 1, 2, 3, ..., m) построить следующую, мы будем просматривать числовую последовательность с конца, пока не встретим i-й меньший стоящего справа от него - первый раз это случится на втором с конца числе. Если такой пары нет, то, значит, текущая перестановка имеет вид m, m-1, m-2,...,3,2,1,0 и является последней - решение задачи завершено. Очевидно, что члены после i-го упорядочены по убыванию; найдем среди них просмотром с конца первый j-й уже больший i-го и поменяем их местами. После перестановки "хвоста" последовательности, начиная с i+1-го по возрастанию искомая перестановка получена.

4.9.6. Рекурсия и алгоритмы перебора вариантов с возвратом.

За сотни лет появилось много решений известной задачи о расстановках на шахматной доске размером NxN N не бьющих друг друга ферзей, а в теории программирования эта задача заняла особое место как иллюстратор таких разделов как системный анализ, формирование рациональных структур данных, алгоритмы перебора вариантов, рекурсия.

Услышав словесную формулировку этой задачи, алгебраист может сказать: "Требуется найти все возможные наборы из N элементов матрицы NxN, не связанные общими строками, столбцами и диагоналями".

Последовавший за этим представлением системный аналитик, готовящий задание для компьютерного решения этой задачи, прежде всего подумает о наиболее рациональном способе отображения результатов ее решения. Предположим, что он предложил представить каждый вариант расстановки в виде 1-мерного числового массива F размером N, индекс элемента которого будет представлять номер строки с "ферзем", а значением элемента будет номер столбца - это

более - менее экономично и удобно для печати на экране или бумаге, хотя пользователю - не математику придется несколько напрягаться, чтобы осмыслить эти цепочки цифр.

Но при необходимости каждый такой массив можно без особого труда трансформировать в рисунок доски с ферзями - это уже работа программиста, не требующая помощи аналитика.

Далее аналитик подумает о необходимости выделить память под массивы, в которых придется хранить сведения о занятости столбцов и диагоналей для анализа при попытках взять в качестве "несвязанного" кандидата очередной столбец в исследуемой строке. Количество столбцов равно размерности матрицы N и для отметки их занятости отведем массив COL размером N. Количество диагоналей ND= 2*N-1 для каждого из 2-х типов.

Придется различать диагонали с одинаковой суммой индексов строк и столбцов принадлежащих им элементов и диагонали с одинаковой разностью индексов и отвести под них массивы DS и DR размером 2*N-1 каждый. Если массивы индексируются с нуля, а разность индексов строки row и столбца col может быть отрицательной, не превышающей ND/2, номер разностной диагонали можно вычислять по индексам строки и столбца как (ND/2)+row-col. Номер "суммирующей" диагонали равен, очевидно, row+col.

Задача решается простым перебором вариантов:

В цикле с постусловием в row-й строке (начиная с 0-й) выбираем поочередно col-столбцы (начиная с 0-го), если соответствующие этим индексам элементы массивов COL, DS, DR свободны (равны 1), отмечаем их занятость обнулением соответствующих элементов массивов COL[col], DS[row+col], DR[DN/2+row-col]. Номер столбца row заносим при этом в row-й элемент массива F и наращиваем row. Если удалось выбрать по элементу в каждой из N строк (row==N), печатаем заполненный массив F, иначе рекурсивно повторяем описанную процедуру. Если в некоторой строке не нашлось свободного элемента, возвращаемся в предыдущую, освобождая помеченные элементы во всех массивах и наращивая в ней столбец.

4.9.7. Определение простых чисел "просеиванием" нечетных на "решете Эратосфена" с битовой упаковкой при хранении.

Древнегреческий математик Эратосфен предложил следующий алгоритм определения списка простых чисел. Простые - это нечетные (кроме 2-йки), т.е. представляют собой

арифметическую прогрессию с шагом 2 и по номеру (индексу) нечетного его значение вычисляется просто как сумма первого члена ряда с удвоенным номером. Кроме того, следующее в последовательности число, кратное текущему, имеет номер, равный сумме значения текущего и его номера. Это дает возможность "вычеркнуть" из последовательности нечетных все кратные текущему простому, как бы "просеивая" остаток ряда после текущего простого.

Вычисляемость значений нечетных по заданному номеру в ряду позволяет не хранить сами значения и каждому нечетному отвести только 1 бит в памяти, моделирующий, как мы знаем, логическую булевскую переменную, принимающую только 2 значения - 0 или 1. Присвоив вначале всем битам значения 1 как кандидатам в простые, взяв в качестве первого члена ряда заведомо простое число 3, мы можем обнулить все кратные ему в ряду, вычисляя номера кратных как сумму значения и индекса текущего простого. Следующее в ряду число с признаком 1 уже будет наверняка простым и мы можем повторить обнуление битов, соответствующих кратным новому простому и т.д.

Отдельный бит не адресуется в памяти и наш массив будет иметь больший размер элементов - например, 1-байтовый, с количеством элементов в 8 раз меньшим, чем количество исследуемых "на простоту" нечетных чисел. И в этом массиве нам предстоит находить и обнулять нужные биты. Составим вначале для лучшего понимания алгоритм, принимающий в качестве аргументов указатель на байтовый массив и номер бита в этом массиве, который ей надо обнулить:

sbros_bit(номер бита i, адрес массива байтов b)

Первым нашим действием будет очевидно определение адреса байта в массиве, содержащего искомый бит - этот адрес равен адресу b 0-го байта, заданного в аргументах, плюс номер байта, вычисляемый как частное от деления номера бита на 8 (размер байта в битах). Само деление на степень 2-ки обычно осуществляют для скорости выполнения сдвигом вправо на равное степени 2-ки число битов, в нашем случае на 3. Очевидно, что номер бита в байте при счете слева равен остатку от деления номера бита на 8 и этот остаток, тоже для скорости, можно определить побитовой операцией "И" с наибольшим возможным остатком, т.е. с числом 7 (только при делении на степень двойки!). При привычном счете справа налево этот результат надо вычесть из номера самого старшего бита, т.е. из 7.

Обнуление бита с номером nbit в 1-байтовом числе делается операцией "И" с числом, получаемым операцией "НЕ" (инверсия) с числом, равным сдвинутой на nbit влево единицей.

4.9.8. Запись в файл.

Задача: Вывести в файл с заданным в командной строке именем текущую таблицу кодировки символов.

План решения задачи таков:

- вначале проверим, задал ли пользователь имя файла для размещения таблицы кодировки; если забыл - выйдем с сообщением на экран причины невыполнения задания;
- откроем файл с именем, размещенным в 1-м элементе массива слов командной строки и проверим успешность открытия;
- таблицу спланируем из 16 строк и 16 столбцов (всего 256 символов) и нумеровать их будем в 16-ричной системе счисления - так получим наиболее компактный и удобный для использования формат таблицы.
- Выведем верхнюю строку с номерами столбцов
- Выведем строки символов, предваряя каждую номером строки
- Закроем файл

4.9.9. Чтение из файла.

Составить программу, подсчитывающую относительные частоты всех символов (частное от деления общего количества появлений каждого символа на суммарное число символов в анализируемом тексте) латинского алфавита в тексте из произвольного, предположительно англоязычного, файла, имя которого задано в командной строке ОС.

Мы знаем, что в английском алфавите 26 букв, поэтому нам понадобится целочисленный массив на 26 элементов для счетчиков каждой буквы - и первым нашим действием будет размещение такого массива в памяти, например в глобальной области. Чтобы хватило размера счетчика даже для приличного по размеру литературного произведения, сделаем элементы массива счетчиков длинного целочисленного типа.

Первые 2 действия в теле программы - проверка командной строки и открытие файла - абсолютно идентичны предыдущей задаче, за исключением того, что файл будет открываться не для записи, а для чтения.

Далее в цикле от начала и до конца файла будем читать по одному символу, проверять его на принадлежность к буквам латинского алфавита, приводить к верхнему (или нижнему -

безразлично) регистру, так как большие и маленькие буквы для этой работы придется считать одинаковыми, и наращивать соответствующий этому символу элемент массива счетчиков. Индекс (номер) элемента массива будем вычислять как разность между кодом символа и кодом буквы 'A' - это возможно потому, что символы латиницы лежат в таблице кодировки подряд, без просветов, и код каждого последующего на 1 больше предыдущего. Сами коды нам знать не надо - мы можем использовать символьный способ задания целочисленных однобайтовых констант.

Попутно будем наращивать и предварительно обнуленный счетчик общего количества букв.

После завершения просмотра файла и заполнения массива счетчиков останется вывести на экран частные от деления каждого элемента массива счетчиков на значение общего счетчика букв.

4.9.10. Подмена таблицы знакогенератора.

Задача: для 3-го текстового режима адаптера VGA подменить таблицу знакогенератора своей, которая будет отличаться только изображением одного из символов псевдографики и вывести этот символ на экран для демонстрации, а после нажатия произвольной клавиши восстановить стандартную таблицу и вывести символ с тем же кодом.

Необходимая справочная информация:

30h-я подфункция 11h-й функции 10h-го прерывания дает информацию для работы с таблицей знакогенератора. При вызове:

AH=11h, AL=30h, BH=6(интересуемся набором 8*16)

На выходе: CL - высота символа, DL - число строк, ES:BP - указатель на активную в текущий момент таблицу рисунков.

Для установки своей таблицы есть подфункция 0 функции 11h 10h-го прерывания:

надо в AH=11h, AL=0, ES:BP - адрес своей таблицы CX-число символов, DX - смещение в таблице, BL-номер таблицы, BH-байтов на символ.

Для восстановления стандартной таблицы знакогенератора используется 4h-я подфункция 11h-й функции 10h-го прерывания с 0 в BX при вызове.

4.9.11. Обработка одномерных числовых массивов (векторов).

Общие сведения.

Программная обработка числовых последовательностей (числовых массивов) - одна из наиболее часто встречающихся задач во всех отраслях обработки информации и является стандартной темой при изучении методов программирования.

Обычно непрограммирующий пользователь готовит предназначенные для того или иного вида обработки числовые массивы в среде обычного текстового редактора, без которого немыслима сегодня работа за компьютером. Числовой массив в файле может быть также порождением другой программы, например, программы учета расходов различных видов энергоносителей, опрашивающей с заданным интервалом времени автоматические датчики - расходомеры или счетчики на предприятии - в этом случае файл будет скорее всего не текстового, а двоичного формата, непригодный для прямого просмотра на экране в среде редактора.

Во всех вариантах программы обработки должны предоставить пользователю по возможности полный сервис, например:

◊ уметь извлечь массивы или их элементы из файла с произвольным именем в оперативную память для обработки ;

◊ предоставить пользователю меню возможных услуг по обработке для выбора в режиме диалога - по возможности дружественного и удобного;

◊ выполнить заданную обработку и отобразить результаты в удобной для пользователя форме;

Надо учесть при этом, что размеры поставляемых на обработку массивов естественно меняются в очень широких пределах от одного вызова нашей программы до другого и становятся известными только во время выполнения - поэтому нет альтернативы динамическому выделению памяти после определения ее необходимого размера.

Объясним, почему числовые массивы часто называют векторами. Дело в том, что любую последовательность чисел можно при желании трактовать как набор значений длин проекций некоторого вектора в многомерном пространстве на координатные оси - геометрическая интерпретация бывает удобной при решении целого ряда задач, в том числе алгебраических, например, решения системы линейных уравнений. Для такого, например 3456-мерного, вектора можно определить некоторые чисто векторные характеристики, например модуль вектора как корень квадратный из суммы

квадратов всех 3456 чисел (многомерный аналог теоремы Пифагора), направляющие косинусы углов с координатными осями - как отношение каждого из чисел к модулю, скалярное произведение с другим вектором той же размерности - как сумму произведений соответствующих координат и может быть что-нибудь еще. Попутно заметим, что квадрат модуля вектора - это скалярное произведение вектора с самим собой.

Файл с числовым массивом для определенности будем считать текстовым, содержащим только цифровые слова, разделенные пробелами или неотображаемыми управляющими символами перевода строки - возврата каретки.

Имя файла будет задаваться в командной строке, количество элементов массива - вводом с клавиатуры (программа и сама может определить общее количество чисел в файле, но часто бывает необходимо обработать часть массива - подмассив и в этом случае задание его пользователем необходимо, в общем случае как номер начального элемента и количество элементов).

Получение имени файла из командной строки, диалог с пользователем, вызов подпрограммы обработки возложим на главную подпрограмму, так как эти действия не зависят от характера обработки.

Открытие и закрытие файла, выделение памяти, ее заполнение из файла, обработку, отображение результатов и освобождение памяти возложим на специализированные подпрограммы.

Отметим попутно, что выделение памяти под размещение всего массива необходимо далеко не во всех задачах - например, при определении среднего арифметического значения составляющих при известном количестве элементов можно извлекать элементы по одному, выполнять деление на размерность массива и тут же суммировать - при этом достаточно памяти только для одного элемента массива.

Поскольку роль пользователя мы тоже вынуждены выполнять сами, то целесообразно создать вначале обрабатываемый файл, заполнив его, например, случайным набором вещественных чисел.

4.9.12.Обработка двумерных числовых массивов (таблиц или матриц или массивов векторов).

Общие сведения.

Многофункциональная программа обработки 2-мерных числовых массивов может включать в себя, например, следующий набор предоставляемых вычислительных услуг:

- почленное сложение и вычитание матриц одинаковых размеров (с одинаковыми количествами строк и столбцов);
- умножение матриц на скаляр;
- транспонирование матриц (взаимная замена строк и столбцов);
- сортировки строк и столбцов матриц по заданному критерию;
- преобразование матриц к треугольной и диагональной форме;
- преобразование матрицы в набор взаимно ортогональных векторов;
- вычисление матрицы - произведения двух матриц;
- обращение матриц (вычисление матрицы, произведение которой на исходную дает единичную матрицу);
- решение систем линейных уравнений, заданных матрицами коэффициентов;

Как и для векторов, будем предполагать для обрабатываемых матриц размещение в текстовом пользовательском файле с произвольным именем в виде цифровых слов, разделенных пробелами и/или символами возврата каретки и перевода строки.

Перенос матриц в оперативную память не сложнее переноса векторов, если не задана избирательность (например перенос только заданных столбцов и пр. Но для обработки матриц уже недостаточно одного размера, необходимо получать у пользователя тем или иным способом две характеристики - количество строк и их длину.)

Для создания диалога с пользователем по выбору задач, файлов с числовыми данными и определению размеров матриц вы можете воспользоваться одним из методов, рассмотренных для одномерных массивов. Мы же рассмотрим только отдельные функции обработки матриц. В качестве аргументов наши функции будут получать:

- ♦ указатель на массив матричных строк из элементов например типа `double`, количество строк и длину строк. Перенос матрицы в память и определение ее адреса при этом возлагается на вызывающую подпрограмму, например, на главную функцию.
- ♦ имя текстового файла с матрицей и ее размеры; в этом случае сама подпрограмма конструирования матрицы в памяти будет Заниматься выделением памяти, а ее освобождение может быть выполнено вызывающей программой;

4.9.12.1.Сложение двух матриц.

Эта операция определена только в том случае, если обе матрицы - слагаемые имеют одинаковое количество строк и столбцов - в этом случае сложение осуществляется почленно:

$$\begin{pmatrix} 1 & 2 & 3 \\ & & \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 1 & -2 & 0 \\ & & \\ 3 & -6 & -10 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 3 \\ & & \\ 7 & -1 & -4 \end{pmatrix}$$

4.9.12.2. Умножение матриц.

Понятие произведения приложимо к прямоугольным матрицам, но имеет смысл только в том случае, если число столбцов первой матрицы равно числу строк второй.

Умножение производится по следующему правилу: элемент на пересечении строки j и столбца k матрицы - произведения мы получим как сумму почленных произведений элементов строки j первого множителя на элементы столбца k второго множителя.

Пример:

$$\begin{pmatrix} 1 & 1 & 0 \\ -1 & -2 & 1 \\ 2 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 4 \\ 1 & 3 & 2 \\ 5 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 6 \\ 3 & -9 & -15 \\ 5 & 2 & 9 \end{pmatrix}$$

4.9.12.3. Решение системы линейных алгебраических уравнений (СЛАУ) методом Гаусса.

Общая формулировка задачи:

Необходимо решить СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = a_{1(n+1)} \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = a_{2(n+1)} \\ . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \quad . \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = a_{n(n+1)} \end{cases}$$

где x_k -неизвестные, a_{ij}-заданные коэффициенты.

В матричной записи:

$$[A]x=y \qquad (2)$$

Общее решение находится просто: домножим обе части матричного уравнения (2) слева на обратную матрицу $[A_{об}]$:

$$[A_{об}][A]x=[A_{об}]y \quad (3)$$

$$x = [A_{об}]y \quad (4)$$

Т.о., задача сводится к обращению матрицы [A].

Метод Гаусса с выбором главного элемента.

Метод Гаусса относится к прямым методам и состоит в последовательном исключении переменных из уравнений таким образом, чтобы в каждом из них осталась только одна - то есть чтобы система уравнений (1) распалась на независимые уравнения с одной неизвестной.

Алгоритм состоит из двух этапов.

На первом выполняются следующие процедуры:

- последовательно для каждой i-й строки из всех последующих строк выбираются уравнения с наибольшим по модулю коэффициентом в i-м столбце: эта процедура называется выбором главного элемента и предназначена для исключения деления на 0 при последующем нормировании строк по диагональным элементам;
- нормирование i-х уравнений по коэффициентам при x_{ii} путем деления всех коэффициентов уравнения на a_{ii};
- исключение i-й переменной из всех последующих уравнений, начиная с j=i+1-го путем почленного вычитания из них домноженного на коэффициент с индексом [j,i] i-го уравнения, т.е. приведение матрицы коэффициентов к верхней треугольной форме;

На втором этапе выполняется подстановка x_n в (n-1)-е уравнение, x_{n-1} - в (n-2)-е и т.д., выполняя т.н. обратный ход алгоритма Гаусса.

4.9.12.4. Вычисление определителя матрицы методом Гаусса.

Вычисление определителей для квадратных матриц может быть выполнено с использованием алгоритма прямого хода Гаусса, приводящего матрицу к треугольной форме, так как определитель треугольной матрицы равен произведению ее диагональных элементов.

ВОЗНИКАЮЩИЕ ПРОБЛЕМЫ: При малых диагональных элементах произведение их может привести к исчезновению порядка, при больших - к переполнению.

4.9.12.5. Метод ортогонализации исходного базиса и его использование для решения СЛАУ.

Метод позволяет осуществить его реализацию при помощи чрезвычайно компактного алгоритма и компьютерной программы, не требует никаких проверок сходимости и сколько-нибудь существенных преобразований исходной системы.

Рассмотренные ранее методы всеми этими свойствами не обладают, что обусловило широкое использование данного метода в прикладных задачах. В качестве вспомогательного средства метод используется в других вычислительных алгоритмах, например, в задачах приближения (аппроксимации) заданных таблично функций.

Сущность метода при решении СЛАУ.

Перенесем свободные члены в исходной системе в левую часть уравнений и положим

$$x_{n+1}=1$$

Получим систему в виде

$$\sum_{j=1}^{n=1} a_{ij} x_j = 0, i \in (1,2,...,n)$$

Суммы в левых частях уравнений можно интерпретировать как скалярные произведения векторов $\vec{a_i}$ и $\vec{x}$; в этом случае искомым решением системы будет некоторый вектор $\vec{X}$ в (n+1)-мерном пространстве, ортогональный базису, образованному системными векторами $\vec{a_i}$. Так как сам базис в общем случае не ортонормирован, то необходима дополнительная процедура его ортогонализации.

Для поиска решения системы добавим к векторам $\vec{a_i}$ линейно независимый от них вектор $\vec{a_{n+1}}$=(0,0,….,0,1). В векторном пространстве размерности n+1 будем строить его ортонормированный базис.

Для поворота какого либо вектора в пространстве на углы, обеспечивающие его ортогональность некоторому ортонормированному базису векторов, достаточно вычесть из него векторы, направленные по ортам базиса и равные по модулю проекциям ортогонализируемого вектора на эти орты (то есть скалярным произведениям вектора и соответствующих орт).

Таким образом, алгоритм вычислений решения СЛАУ методом ортогонализации состоит из следующих процедур:

- нормируем вектор $\vec{a}_1$ по модулю делением его компонент на корень квадратный из суммы квадратов этих компонент;
- начиная со второго вектора и до (n+1)-го последовательно осуществим ортогонализацию с предыдущими и затем нормирование по модулю, повторив эти процедуру несколько раз (3-5) для предотвращения возможной неустойчивости процесса ортогонализации и повышения точности вычислений в случаях близких к нулю значений определителя исходной системы.
- последнюю строку расширенной матрицы разделим на элемент $a_{(n+1)(n+1)}$, который должен по определению быть равен 1, получив таким образом искомое решение в n первых элементах.

4.9.13. Двоичный поиск в упорядоченных массивах.

Очевидно, что основным назначением любых сортировок является ускорение последующих процессов поиска необходимой информации в массивах данных с элементами любого типа. При поиске элементов в неупорядоченных массивах с заданным или ближайшим к заданному значением приходится осуществлять последовательный просмотр массива от начала до встречи с искомым элементом - в больших массивах даже на быстродействующих компьютерах это может занимать неоправданно много времени.

В упорядоченных массивах есть возможность существенно улучшить сам алгоритм просмотра массива - одним из таких методов является широко используемый двоичный поиск. Сущность его проста мы исследуем "срединный" элемент массива (с индексом, равным половине длины массива) - если он больше искомого (в массиве, упорядоченном по неубыванию), то исследуем срединный элемент верхней половины массива, если меньше - срединный элемент нижней половины и т.д., пока либо не достигнем полного равенства исследуемого элемента с заданным, либо размер исследуемого подмассива не достигнет 1-го элемента.

В последнем случае мы находим ближайшее к заданному значение. По результатам сравнения элементов массива с заданным мы изменяем индекс последнего элемента при переходе в верхний подмассив или индекс первого элемента при переходе в нижний подмассив. Индекс рабочего элемента в обеих случаях равен среднему между верхним и нижним. Нам понадобится так называемый брейк-флаг, чтобы знать, было ли

в процессе поиска найдено точное совпадение и решить, выводить ли ближайшее к искомому.

Алгоритм поиска не зависит от типа элементов массива, меняется только метод их сравнения.

Для демонстрации метода мы дополним программу сортировки строк следующим: пусть в командной строке может задаваться не только имя файла с несортированным массивом, а и слово-строка для поиска. Если такое слово задано, то после сортировки будет выполняться двоичный поиск ближайшего по значению слова.

При работе со строками при отсутствии точного совпадения искомой строки с имеющимися в массиве определение ближайшей по значению несколько сложнее, чем для числовых данных. Для предоставления этой услуги мы составили простую подпрограмму сравнения строк cmpstr(), которая при отсутствии точного равенства сравниваемых строк возвращает число, значение которого тем меньше, чем больше количество первых совпавших символов. Эту функцию мы и используем для исследования элементов массива в окрестностях индекса, при котором поиск завершился - ближайшее по значению слово предположительно либо соответствует индексу завершения, либо находится на один номер выше или ниже, это зависит от четности (нечетности) количества элементов в массиве и индекса при завершении поиска.

5. Краткое описание языка Си.

5.1. Базовые понятия.

Си - универсальный язык для лаконичной и в то же время понятной записи алгоритмов, оснащенный удобными управляющими структурами и достаточным для профессионального программирования набором операций. В нем практически отсутствуют абстрактные, искусственно создаваемые типы данных - они все числовые.

КОММЕНТАРИИ В ТЕКСТЕ Си-ПРОГРАММЫ.

Комментарии - это те фрагменты программного текста, которые транслятор "пропускает", не обрабатывает, но которые позволяют описать, документировать программу. При отладке отдельные фрагменты текста программы, подозреваемые в наличии ошибок, могут быть заключены в "комментаторные скобки" для "локализации" ошибок - если после исключения фрагмента из процесса трансляции программа заработала, то ошибка - в закомментированном участке, иначе в комментарий "загоняется" следующий фрагмент.

В качестве комментаторные скобок в Си используются пары символов

```
/* Это многострочный
   комментарий */
```

В C++ добавлена возможность организации однострочных комментариев `//`:

```
//Весь текст от двойного слэша до конца строки-комментарий
```

Многострочные скобки могут охватывать вложенные в них одно- и многострочные комментарии.

ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ТЕКСТА Си-ПРОГРАММЫ

Компилятор текста Си-программы автоматически вызывает другую специальную программу cpp.exe под названием "препроцессор", осуществляющую предкомпиляционную обработку текста и подготовку его к компиляции. Помимо многих сервисных услуг компилятору типа предварительной нумерации строк, препроцессор распознает предназначенные для него директивы по символу '#' с последующим словом-директивой.

ДИРЕКТИВА ПОДКЛЮЧЕНИЯ ТЕКСТА ИЗ ВНЕШНЕГО ФАЙЛА.

Для вставки в программный текст уже готовых файлов, чаще всего заголовочных файлов связи с библиотеками, используется директива препроцессора

```
#include <имя_включаемого_файла>
#include "имя_включаемого_файла"
...
```

На места этих директив будут подставлены тексты указанных файлов; при угловых скобках они отыскиваются в каталоге, указанном при настройке компилятора (раздел DIRECTORIES в меню OPTIONS интегрированной среды), а при двойных кавычках - сначала в текущем каталоге, а затем в том же, что и для угловых скобок.

ДИРЕКТИВА МАКРООПРЕДЕЛЕНИЙ.

Служит для замены препроцессором всех вхождений некоторого имени, стоящего за `#define`, заданным текстом; в простейшем случае может использоваться для задания значений именованным константам (в C++ для этого лучше использовать модификатор `const`):

```
#define MAX_POINT 50
```

В более сложных вариантах в директиве `#define` могут быть указаны параметры:

```
#define sum(x,y) (x+y)
#define sq(x) ((x)*(x))
```

ИМЕНА (ИДЕНТИФИКАТОРЫ) В Си.

Типы, переменные, константы, функции - все должны иметь имена, по которым их распознают в операциях и операторах, состоящие не более чем из 32 символов - латинских букв, цифр, символа подчеркивания. Имя не должно начинаться с цифры и совпадать с зарезервированными самим языком словами. Большие и малые буквы в Си различаются; неписаный стандарт программистов на Си состоит в использовании маленьких букв во всех именах, кроме имен констант. Для улучшения "разборчивости" составных имен используют символ подчеркивания как "соединительную планку" между различными по смыслу частями имен.

Зарезервированные слова:

```
asm auto break case cdecl char const continue default
do double else enum extern far float for goto huge if
int interrupt long near pascal register return short
signed sizeof static struct switch typedef union
unsigned void volatile while _cs _ds _es _ss _AH _AL
_AX _BH _BL _BX _CH _CL _CX _DH _DL _DX _BP _DI _SI _SP
```

5.2. Типы данных и их внутреннее представление.

В программах обрабатываются константы и переменные.

Под переменными в С подразумеваются именованные участки оперативной памяти с указанным размером и делятся на простые (одноэлементные, скалярные) и многоэлементные (агрегированные, состоящие из простых и многоэлементных). В эти участки памяти в процессе выполнения программы могут заноситься произвольные числовые значения. Размер размещаемой в памяти переменной определяется ее типом, который указывается слева от имени переменной. Имена переменных придумывает программист - приведенные ниже совершенно произвольны и вы можете заменить их любыми другими.

В языке определены 6 основных числовых типов для одноэлементных переменных - 3 для целых чисел и 3 для вещественных. Каждый из этих типов предназначен для определения соответствующей ему длины выделяемой памяти. Самым "коротким" является однобайтовый целочисленный тип `char`:

`char` ch1,ch2; - будет выделено по одному байту для целочисленных переменных с именами ch1 и ch2.

Более "длинные" целочисленные типы имеют размер кратный байту и именуются `int` (от слова `integer` - целый) и `long` - длинный, в 2 раза длиннее `int`:

`int` i1,i2,i3; - будет выделена память для 3 целочисленных переменных с длиной, определенной для типа `int`;

`long` l1,l2; - будет выделена память для размещения 3-х целочисленных переменных длиной в 2 раза больше, чем `int`.

Особую структуру имеет производный тип - указатель на переменные того или иного типа, предназначенный для хранения адресов других переменных. Размер отводимой под хранение указателя памяти не зависит от размера адресуемого его значением объекта, а определяется только способом адресации или началом отсчета адреса - если адрес отсчитывается от границы сегмента (ближний или `near` указатель), то под него отводится вдвое меньше байтов, чем для так называемого дальнего или `far` - указателя, содержащего и адрес сегмента в старших байтах и смещение от границы сегмента в младших байтах. По существу ближний указатель есть беззнаковый целочисленный тип, а дальний указатель состоит из 2-х беззнаковых целых. Указательный тип по синтаксису С записывается с помощью звездочки после названия адресуемого типа:

`char` * ch_ptr1; -указатель на тип `char`;

`long` * `long`_ptr; -указатель на тип `long`;

Вещественные числовые типы носят названия `float`, `double`, `long double`. Мы рассмотрим их подробнее чуть позже, а настоящее краткое описание типов необходимо для понимания рассказа о константах.

КОНСТАНТЫ. Простые константы - это некоторые вполне определенные значения, они предназначены для размещения в отведенных для переменных участках памяти, поэтому они тоже имеют тип, который распознается транслятором по способу написания значений или указывается явно при описании константы.

Константы в С могут задаваться при присваивании значений переменным:

- непосредственными значениями в 10-тичной системе счисления:

  cd=12;

- непосредственными значениями в 16-ричной системе счисления (только целые - значению константы должны предшествовать пара символов 0x):

  ih=0xa34e;

- непосредственными значениями в 8-ричной системе счисления (только целые - значению константы должен предшествовать 0:

  co=0754;

- символьным значением константы, равной по значению ASCII-коду соответствующего символа, заданного в одиночных кавычках:

  uc='S';

- заданием числового кода символа в 8-ричной или 16-ричной системах счисления с предшествующей ему обратной косой чертой в одиночных кавычках:

  uch='\xa2'; (16-ричн)

  uco='\0100';(8-ричн)

Кроме того, в языке предусмотрен ряд символьных констант для обозначения управляющих ASCII-кодов:

\a- при выводе на экран вызывает звуковой сигнал;

\b- на позицию влево;

\n- перевод строки;

\r- возврат каретки (переход в начало строки);

\t,\v- горизонтальная и вертикальная табуляция;

Обозначению ряда символов должна предшествовать обратная косая: \\, \',\",\?.

После значения константы могут стоять уточняющие ее тип символы: l для типа `long`, u для `unsigned`. Например 567ul - беззнаковая длинная.

Еще раз подчеркнем, что в Си символьный литерал - это не более чем способ задания числовой константы и операции вида

```
int c3= 'B' + 12;
```

вполне допустимы.

Вещественные константы-значения состоят из цифр, десятичной точки и знаков десятичного порядка e или E и по умолчанию имеют тип `double`:

```
float f1= 2e1;
double d1 = 3.14159265359;
double d2 =.1234;
float f2 = 2.1e-12;
```

Строковые константы - значения (строковые литералы) оформляются в виде текста в двойных кавычках и обычно используются для определения значений указателей на тип `char`:

```
char *str_ptr1 = "Это строковый литерал";
```

По такому описанию компилятор подсчитывает количество байтов, необходимых для размещения в памяти этой строки, добавляет в конец 1 байт с завершающим строку символом с кодом 0, выделяет необходимую память и адрес ее первого байта помещает в переменную указательного типа str_ptr1, предварительно выделив память и для нее.

Адресные константы-значения задаются полным адресом - для ближних адресов 2-байтовым целым, а для дальних - 4 - байтовым. Например, адрес видеобуфера в 3-м текстовом режиме видеосистемы может записываться так:

```
char far * vbuf = (char far*)0xB8000000;
```

а смещение указателя головы буфера клавиатуры относительно начала области BIOS (адрес 0x40):

```
_es=0x40;
char register _es *head = 0x1a;
```

"Адресный нуль" обозначен в Си именем NULL и обозначает отсутствующий адрес.

Именованные константы в Си задаются:

- директивой `#define` препроцессора для макроподстановок:

  ```
  #define const1 234
  #define const2 98.456
  #define const3 «Навуходоносор»
  ```

Препроцессор, встречая в тексте программы имя константы, подставляет определенное вторым словом ее значение.

В C++ это можно сделать использованием модификатора `const` в объявлении типа :

```
const int n1=245,n3='A';
const long double pi = 3.14159265358979323846;
```

Такая псевдопеременная не может использоваться в левой части оператора присваивания, т.е. ее значение после начальной инициализации не может быть изменено. Использование именованных констант улучшает читабельность программ и позволяет избежать трудноуловимых при отладке ошибок набора значений при многократном использовании одних и тех же констант.

ПЕРЕМЕННЫЕ. Каждая используемая в программе переменная должна иметь имя и определенный тип; имя обеспечивает возможность работы с отведенным для переменной участком памяти и представляет собой последовательность до 32 букв латинского алфавита, цифр, знака подчеркивания '_'.

Указанием типа используемой переменной мы сообщаем программе компиляции размер необходимой для ее размещения памяти и фиксируем допустимый с этим типом набор операций.

Синтаксис объявления переменных в Си чрезвычайно прост и удобен:

слева направо записывается тип и список разделяемых запятыми имен размещаемых в памяти переменных этого типа, возможно, с "попутным" присвоением начальных значений:

ТИП ИМЯ1, ИМЯ2=ЗНАЧЕНИЕ, ИМЯ3;

В конце списка ставится точка с запятой.

К **СКАЛЯРНЫМ** (одноэлементным) типам в Си относятся:

- тип `void` - пустой, неопределенный тип, не имеющий размера;
- числовые (арифметические) типы:

ЦЕЛОЧИСЛЕННЫЕ

`char` - однобайтовые целые;

`int` - целые с размером кратным байту (2 или 4);

`short` - короткое (половинное) целое;

`long` - длинное (удвоенно по размеру) целое;

Все целые типы могут иметь перед названием типа модификаторы

знаковый - signed; например, signed long sl;

беззнаковый - unsigned; например, unsigned char ch;

У всех знаковых старший бит отводится под код знака.

ВЕЩЕСТВЕННЫЕ ЧИСЛОВЫЕ

float (4-байтовое),

double (8-мибайтовое),

long double (10-батовое).

УКАЗАТЕЛИ НА ТИП - используются для хранения адресов других размещенных в памяти объектов определенного типа и объявляются со звездочкой между названием типа и именем указателя, например:

void * vp; -vp есть указатель на пустой тип, для него недопустимы операции адресной арифметики. Указатель типа void может быть преобразован в указатель на любой определенный тип.

char * ch_ptr; - ch_ptr есть указатель на тип char, ему может быть присвоено значение адреса переменной типа char.

long * long_ptr; - long_ptr есть указатель на тип long, может хранить адрес переменной типа long.

Различают ближние (near) и дальние (far) указатели. Ближние указатели занимают 2 байта памяти и содержат смещение в байтах относительно адреса, заданного в регистре DS.

Дальний 4-хбайтовый указатель содержит полный адрес объекта в памяти - сегментный адрес в 16-байтовых параграфах в 2-х старших байтах и смещение в 2-х младших.

Частным случаем far-указателя является нормализованный huge - указатель с максимально возможным значением сегментного адреса и следовательно с минимальным (менее 16 байтов) смещением.

Форма указателя может быть задана явно:

```
char far * strings;
huge norm_ptr;
```

Если явное задание формы указателя отсутствует, используется значение по умолчанию в зависимости от заданной при компиляции "модели памяти" - в C их поддерживается шесть:

tiny - "крохотная", small - малая, medium - средняя, compact -компактная, large - большая, huge - "огромная".

Опишем их чуть подробнее.

tiny (крошечная) - во все сегментные регистры CS, DS, SS, ES засылается один и тот же адрес, под программный код, статические данные, динамически размещаемые данные и стек отводится 64 Кбайт, все указатели только ближние 2-байтовые.

small (маленькая) - под код программы отводится 64-Кбайтовый сегмент, а стек, куча и статические данные "прописываются" в одном "коммунальном" 64-Кбайтовом сегменте. Все указатели в такой программе будут 2-байтовыми ближними.

medium (средняя) - код программы может быть до 1 Мбайта, т.е. в программном коде указатели на подпрограммы будут дальними 4-байтовыми. Стек, куча и статические данные по-прежнему размещаются в одном 64-Кбайтовом сегменте и все данные адресуются ближними указателями.

compact (компактная) - 64 Кбайт для кода и до 1 Мбайт под данные, в том числе по до 64 Кбайт под статические данные и стек. Вызовы подпрограмм с помощью ближних указателей, а данные адресуются дальними.

large (большая) - 1 - Мбайтный код, 64 Кбайт под статические данные, до 1 Мбайт куча. И программа, и данные адресуются дальними указателями.

`huge` (огромная) - такая же как большая, но объем статически данных может быть больше 64 Кбайт.

Всем указателям в Си можно присвоить безопасное значение адреса - нуль целого типа - гарантируется. что этот адрес не совпадет ни с одним использованным адресом. Для этого можно либо использовать именованную константу NULL, либо непосредственно 0.

```
char * ch_ptr = 0;
int * int_ptr = NULL;
```

Если указатель описан в глобальной области (вне тела какой-либо подпрограммы) или в теле функции с предписанием `static`, то компилятор сам заботится об его обнулении, как и любых других данных, размещаемых в сегменте данных. Объявление же указателя в локальной области (в теле любой подпрограммы, в том числе главной `main`) не приводит к инициализации указателя каким-либо осмысленным значением и может служить источником ошибок при попытках записи чего-нибудь по содержащемуся в указателе бессмысленному адресу.

Осмысленные значения указатели получают, когда им присваиваются значения адресов переменных, значения других уже инициализированных указателей или значения адресов, возвращаемых функциями динамического распределения памяти.

В трех первых (младших) моделях все указатели на данные - ближние по умолчанию. Вторая тройка (старших)

моделей обеспечивает по умолчанию дальние указатели на данные.

Отметим, что при использовании дальних `far` -указателей всегда существует неоднозначность - один и тот же адрес может быть задан множеством пар значений сегмента и смещения, поэтому сравнение `far`-указателей не может считаться корректной процедурой.

Вторая опасность в работе с `far`-указателями состоит в возможности переполнения смещения в операциях адресной арифметики - при наращивании указателя растет только значение в 2-х байтовом смещении и при переполнении перенос в старшие байты сегмента не происходит. Этих недостатков лишен `huge`-указатель, который автоматически нормализуется к минимальному смещению после любой операции, что позволяет адресовать из С - программ блоки более 64 Кбайт.

Особым типом ближнего `near`-указателя являются указатели

тип `_ds`* имя, тип `_es` * имя, тип `_ss`* имя, тип `_cs`* имя

Смысл приведенных записей следующий: указатель с приведенным именем является ближним и его значением есть смещение от сегмента, заданного в соответствующем регистре. Эти указатели могут естественно использоваться при правильной установке соответствующих регистров. Для доступа к сегментным и другим регистрам процессора в языке С можно использовать псевдопеременные `_cs`, `_ds`, `_es`, `_ss` и т.п.:

```
unsigned _es * ptr= (unsigned _es*)0x0000;
_es=0xb800;
...
*ptr='A'; ptr=ptr+2;...
```

Надо отметить, что работа с этой формой указателей и указанный способ доступа к регистрам через псевдопеременные требует осторожности, т.к. регистры могут неявно переопределяться операторами С - программ.

О МЕСТЕ ОБЪЯВЛЕНИЯ ПЕРЕМЕННЫХ. В С++ допускается объявление переменных в любом месте программы, там, где они вам понадобились, даже внутри операторов. В "классическом" Си - в любом месте глобальной области (вне тел подпрограмм, не ограниченной, не охваченной фигурными скобками границ блоков).

ОБ ИНИЦИАЛИЗАЦИИ ПЕРЕМЕННЫХ. Переменные могут инициализироваться (получать начальное значение) как в месте объявления, так и отдельной операцией в теле подпрограммы, в том числе в "последовательной" цепочке присвоений:

```
int x=y=z=v=1;
float f;
....
f=3.789;
```

К **АГРЕГИРОВАННЫМ (МНОГОЭЛЕМЕНТНЫМ)** типам в Си относятся массивы, структуры и объединения.

МАССИВЫ - последовательность расположенных в памяти вплотную друг к другу элементов одного и того же типа, нумеруемых от нуля; массив имеет имя, представляющее собой константный указатель на нулевой элемент последовательности - значение этого указателя определяется компилятором при обработке объявления и в дальнейшем не может быть изменено.

Объявление массива с известным размером осуществляется указанием типа составляющих его элементов, имени массива и его размера (количества элементов в массиве) в квадратных скобках после имени:

`int` `int`_array[123]; - `int`_array есть массив из 123 целых.

`float` `float`_array[456]; `float`_array-массив из 456 вещественных.

`char`* ch_ptr[32];ch_ptr - массив из 32 указателей на тип `char`.

`float` (* arr_ptr)[45];arr_ptr- указатель на массив 45 элементов типа `float`;

Обращение к элементам массивов осуществляется по имени массива и номеру (индексу) элемента в массиве:

`int`_array[6]=8; 6-му элементу присвоено значение 8;

`float`_array[24]=789.234; 24-му элементу присвоено 789.234ж

`float` f=`float`_array[4]; в переменную f типа `float` скопировано значение из 4-го элемента массива `float`_array.

Компилятор отводит под массив `sizeof`(тип)*(число элементов) байтов памяти (`sizeof`(имя_типа или имя объекта) - операция определения размера типа или конкретного объекта в памяти) .

Имя массива в Си есть указатель на его 0-й элемент, поэтому к элементам массива возможен и другой способ доступа - через операцию '*' - "разадресации" указателя на нужный элемент. Указатель на элемент с номером index равен увеличенному на index имени массива:

`int`_array +6 - это значение указателя на 6-й элемент массива `int`_array,

*(`int`_array +6)= 8; равносильно `int`_array[6]=8;

`float`_array+24 - это значение указателя на 24-й элемент массива `float`_array.

*(float_array+ 24)=789.234;
равносильно float_array[24]=789.234;

Наращивание указателя на 1 изменяет его значение на размер адресуемого типа, наращивание на N приводит к наращиванию на N размеров типа.

В связи с тем, что имя массива есть указатель на его начало, значения массивов нельзя присваивать друг другу одной операцией. Запись типа

```
char a[40],b[40];
...
a=b;
```

приведет к тому, что мы получим два указателя на начало массива b, при этом массив a будет "утерян", так как в место хранения его адреса запишется адрес массива b. Копирование значений одного массива в другой в C можно осуществлять только поэлементно!

Массивы можно инициализировать значениями в месте объявления, как и простые переменные - начиная с начала и до любого элемента:

```
char ca[40]={'A','B','C','U'};
```

Компилятор запишет в первые 4 элемента массива целочисленные коды заданных символов, а остальные элементы останутся неинициализированными и будут иметь либо нулевые значения, либо случайные, в зависимости от места объявления массива (в глобальной области, т.е. вне тела любых подпрограмм компилятор обнуляет все переменные сам, а значения локальных переменных подпрограмм, размещаемые в стеке, остаются при объявлении на усмотрение программиста).

При наличии процедуры инициализации элементов массива в месте объявления размер в квадратных скобках может быть опущен:

```
float fa[]={1.234, 43.28, 0.12345};
```

Компилятор сам подсчитает количество значений, выделит под них память и адрес ее начала запишет в переменную по имени fa.

Если имя массива есть указатель, то и имя любого (кроме void*) указателя можно трактовать как имя массива:

float* f_ptr; - f_ptr - можно трактовать как массив неизвестного размера и невыделенной под массив памятью, поэтому обращение к любому элементу массива для записи может привести к непрогнозируемым последствиям для выполняемой программы. Такого рода ошибки не "вылавливаются" компилятором Си и за это он обычно

подвергается критике. Но это оказывается удобным при необходимости работы по абсолютным адресам, например, прямой работы с видеопамятью, адрес которой мы обычно знаем:

```
char far* vbuf=(char far *)0xb8000000;
```

Мы присвоили указателю vbuf значение адреса видеобуфера в текстовом режиме и теперь можем работать с видеобуфером как с обычным массивом, результаты при этом будут немедленно отображаться на экране:

vbuf[0]='A'; это приведет к выводу буквы 'A' в левом верхнем углу экрана;

Трактовка указателя как массива используется часто для формирования литеральных константных массивов символов - ASCIIZ -
строк с текстом в двойных кавычках:

```
const char* str="Строка сообщения об ошибке";
```

Компилятор сам подсчитывает количество символов в строке, выделяет память на 1 символ больше и в последний записывает 0, как признак окончания строки, а адрес начала выделенной памяти заносит в переменную str.

МНОГОМЕРНЫЕ МАССИВЫ. Описываются с указанием всех размеров в отдельной паре квадратных скобок и могут инициализироваться в месте объявления:

```
double matrix[40][400];
int m[3][3]={00,01,02,
             10,20,30,
             20,30,40};

char message[3][80]={
"Первое сообщение",
"Второе сообщение",
"Третье сообщение"};
```

Элементы многомерных массивов хранятся в памяти построчно, т.е. в порядке возрастания самого правого индекса. Список начальных значений соответствует их размещению в памяти.

Имя 2-мерного массива есть указатель на массив указателей. Элементы массива указателей суть указатели на начальные элементы строк массива.

При инициализации многомерных массивов можно не указывать одну (самую левую) размерность массива - компилятор сам определит размерность по списку инициализации :

```
int m[][3]={00,01,02,
            10,20,30,
            20,30,40};
```

Если необходимо присвоить значения не всем, а только нескольким первым элементам строк, строки следует выделять фигурными скобками:

```
int m[][3]={00},
           {10,20},
           {20,30,40}};
```

Все сказанное чаще всего используется для константных массивов, а для обрабатываемых программами массивов - переменных чаще всего размерность неизвестна при составлении программы, вернее, программа составляется обычно так, чтобы она обрабатывала массивы любых заданных пользователем на стадии ее выполнения размеров. В этом случае в распоряжении программиста есть только указатель на начало массива A, возвращаемый обычно подпрограммой DOS выделения памяти - и нам приходится работать, например, с одномерным массивом как с 2-мерным, для которого мы узнали на стадии выполнения программы количество строк N и количество элементов в каждой строке M. Если элемент 2-мерного массива характеризуется номером строки row и номером столбца col, то доступ к нему по адресу 0-го элемента можно осуществить так:

```
*(A+row*M + col)=2.35;
```

(указатель на 0-й элемент + номер строки, умноженный на ее длину + номер столбца). Другая и обычно лучше воспринимаемая форма записи:

```
A[row*M + col]=2.35;
```

СТРУКТУРЫ И ОБЪЕДИНЕНИЯ.

Это многоэлементные типы данных, объединяющие под одним именем в одну область памяти любое количество переменных возможно различных типов, объединенных для пользователя общим смыслом. Например, в одну структурную переменную могут быть объединены все переменные, содержащие различные сведения о стоящих на учете в Госавтоинспекции района или города автомобилях - марка, цвет кузова, номер госрегистрации, фамилия владельца, адрес владельца, адрес стоянки автомобиля, дата последнего техосмотра и т.п. Входящие в структуру переменные называют

полями. Формирование структурных переменных может осуществляться в 2 этапа:

-на первом этапе объявляется шаблон структуры по следующему синтаксису:

ключевое слово `struct` имя_шаблона {

тип_поля1 имя_поля1;
тип_поля2 имя_поля2;
...
тип_поляN имя_поляN;
};

Естественно, что при этом формируется только созданный программистом новый тип данных и никакое место в памяти для размещения самих данных не отводится.

-на втором этапе объявляется переменная созданного структурного типа по следующим синтаксическим правилам:

`struct` имя_шаблона имя_переменной.

Например:

```
struct BOOK
{
    char name[20]; - массив для фамилии
    char title[50]; - для названия книги
    int year; -  для года издания
    float price; - для цены
};
```

Затем может быть объявлен одна или массив переменных этого типа:

```
struct BOOK a,b[1500];
```

Синтаксис С позволяет совмещать описание шаблона и создание в памяти переменной по этому шаблону:

```
struct BOOK{ char name[20];
char title[50];
int year;
float price; }b[1500];
```

Доступ к отдельным полям структурных переменных осуществляется по имени переменной и имени поля, разделенных точкой:

```
b[81].name="Джон Голсуорси";
b[81].year=1990;
```

Две структурные переменные одного и того же шаблона могут участвовать в операции присвоения с любой стороны -

при этом значения всех полей правой переменной копируются в поля левой:

```
b[200]=b[300];
```

При описании структурных переменных допускается их попутная инициализация:

```
struct BOOK b[]={
    {"Ильф и Петров", "Двенадцать стульев", 1991, 6.34},
    {"Чехов А.П.", "Анна на шее", 1981, 1.32},
    {"Сталин И.В.", "Марксизм и вопросы языкознания",1991, 6.34}
};
```

Поля структурных переменных могут иметь любой известный компилятору тип, в том числе тип структурный, если шаблон этой структуры описан раньше, чем используется. Полями структурных переменных могут быть и указатели на собственный структурный тип.

```
struct BOOK{ struct BOOK *b_ptr;
        char name[20];
        char title[50];
        int year;
        float price;
        };
```

Естественно, есть возможность работы и с указателями на структурный тип, как и на любой другой тип; доступ к элементам структурных переменных через указатель осуществляется с помощью операции -> :

```
struct BOOK * book_ptr,b;
...
book_ptr=&b;-определяем адрес структурной переменной
book_ptr-> name = "Митчел Уилсон";
```

БИТОВЫЕ ПОЛЯ В СТРУКТУРАХ.

В языке С структуры могут использоваться для доступа к отдельным бита байтов или слов; для этого битовые поля структурного шаблона должны объявляться по синтаксису:

тип [имя]: ширина_в_битах;

Тип битового поля может быть знаковым и беззнаковым; для поля будет выделено столько битов, сколько заказано в поле ширины, но общая длина всех битовых полей всегда будет кратной длине типа `unsigned`.

Пример:

```
stuct BIT_FIELDS{ int i:2;
                  unsigned j:2;
                  int:2;
                  int k:2;
                  int l:8;}my_struct;
```

Для полей с опущенными именами память все равно выделится, но доступ к ним невозможен, а к именованным полям доступ полностью аналогичен "полномерным" полям.

ОБЪЕДИНЕНИЯ.

Описание объединений формально отличается только ключевым словом `union` вместо `struct`, но по содержанию отличие очень существенно - в объединениях (иногда переводят словом "союз") все поля независимо от типов хранят свои значения в одной и той же области памяти:

```
union ALERNATETIVE_DATA
{
char s[4];
unsigned j[2];
}data;
```

Объединения могут использоваться для преобразования одних типов данных в другие - в этом случае запись осуществляется как бы в поле одного типа, а считывание - из поля другого типа, но так как они лежат в одном и том же месте, "перекрываются", то происходит преобразование одного типа в другой. В отдельных случаях, когда нам необходимо использовать или цельное отображение участка памяти, или его часть, использование объединений оказывается очень удобным - например, объединение полей для хранения значений полных процессорных регистров и полурегистров, 2-байтовых и 1-байтовых кодов клавиш, формирование дальнего указателя из сегмента смещения или его разборка на указанные элементы и пр.

ТИП ПЕРЕЧИСЛЕНИЯ служит для улучшения читабельности программ, приближения их текста в некоторых случаях к естественному языку. При использовании вначале описывается само перечисление, задаваемое ключевым словом `enum` с именем шаблона и перечисляемыми элементами в фигурных скобках:

```
enum color_type {red, amber, green };
```

Затем может быть описана переменная перечислимого типа:

`enum` color_type c1;

и она может принимать любое значение из перечисленных в скобках:

c1=green;

С помощью директивы `typedef` можно описать новые переменные перечисления более экономно:

`typedef enum` {понедельник=1, вторник, среда, четверг, пятница, суббота, воскресенье}DAYS;

DAYS anyday;

Собственно в фигурных скобках перечислимого типа находятся именованные целочисленные константы, которым по умолчанию компилятор присвоит значения от 0 и далее с шагом 1. Но мы можем и сами приписать желаемые значения каждому перечисляемому элементу:

`enum` color_type {red=4,amber=14,green=3 };

5.3. Операции над данными в Си.

Операции над целыми типами		
Сложение	+	
Вычитание	-	
Умножение	*	
Деление (получение частного)	/	
Деление (получение остатка)	%	
Побитовое И	&	
Побитовое ИЛИ	|	
Побитовое исключающее ИЛИ	^	
Инверсия битов	~	
Сдвиг влево	x << n	x-целое число n-количество разрядов
Сдвиг вправо	x >> n	
Операции с вещественными числовыми типами		
Сложение	+	
Вычитание	-	
Умножение	*	
Деление (получение частного)	/	

Операции с любыми числовыми типами		
Инкремент	++	увеличение на 1
Декремент	--	уменьшение на 1
Получение размера типа или объекта	`sizeof` (тип или объект)	
Получение адреса объекта в памяти	&	результат надо поместить в указатель. Пример : `char` *cptr = &ch;
Присвоение значения	=	
Операция с последующим присвоением значения	операция=	Например: x+=2; x увеличен на 2 y/=13; y уменьшен в 13 раз
Операции сравнения		их результат - 1 или 0
больше	>	x>y
меньше	<	x<y
равно	==	x==y
не равно	!=	x!=y
больше или равно	>=	x>=y
меньше или равно	<=	x<=y
Оператор возврата из подпрограммы в программу-предок	`return` (возвращаемое значение)	
Оператор переименования типов	`typedef старое_имя новое_имя`	Пример: `typedef unsigned` WORD

5.4. Общая структура Си-программы.

Ознакомившись с типами данных и операциями над ними, естественно попытаться составить для начала какую-нибудь простую программу, чтобы понять «как это делается». Для этого необходимо знать, какие общие требования предъявляет компилятор к тексту программы. Си - программа должна состоять из одной или нескольких подпрограмм, но одна из них должна иметь опознавательный знак, определяющий ее как главную, с которой начнется выполнение всей программы в целом и на которой это выполнение завершится в случае благополучного исхода. Таким опознавательным знаком в Си является фиксированное имя главной подпрограммы (функции) - «`main`» (в Си все подпрограммы называются функциями и

других типов подпрограмм язык не поддерживает). Обычно с нее и начинают составление программы.

Синтаксис определения любой функции в Си требует записывать слева направо:

тип_возвращаемого_функцией_значения
имя_функции(Список типов и имен аргументов)

скобки после имени обязательны - именно по ним компилятор распознает функцию.

Аргументы разделяются запятыми, если они не нужны, то в скобках пишется `void`

{.....} - пара обязательных фигурных скобок обозначает границы тела функции, а самим телом являются операторы определения локальных данных, операции над данными, вызовы вспомогательных функций и пр.

Таким образом, ничего не делающая и ничего не возвращающая главная функция в Си выглядит так:

```
void main(void) {}
```

Возвращаемое главной функцией значение - это значение, возвращаемое всей программой в вызвавшую программу - предок и оно может быть однобайтовым целым числом в диапазоне 0-255 по абсолютному значению. Это ограничение касается только `main` и связано с тем, что чаще всего программа вызывается на выполнение из операционной системы (COMMAND.COM), которая считает этого достаточным и сама вообще говоря не собирается анализировать это значение - разве что вы составите соответствующий командный файл для этой цели. Для возврата значения используется оператор `return` с возвращаемым значением после него в круглых скобках или без них. Круглые скобки обычно используют, если возвращаемое значение есть вычисляемое выражение:

```
return a; или return(a*b - c);
```

Главная функция обычно используется как «генеральный план» программы в целом и помимо некоторых подготовительных операций содержит вызовы других функций - составленных вами или библиотечных.

Любая другая вспомогательная подпрограмма имеет такую же структуру при определении:

```
double func1(int i, double d, char* ch_otr)
    {double dd;...dd=5.25;...return dd;}
```

Приведенное определение представляет функцию с именем func1, требующую при вызове на выполнение передать ей значения 3-х параметров в заданной последовательности по типам `int`, `double`, `char*` и возвращающую вещественное

значение типа `double`. Возвращаемое значение может быть «принято» в вызвавшей функции в переменную соответствующего типа, стоящую в левой части оператора присваивания:

```
void main(void) {...double d1=func1(5,3.256,&ch);...}
```

Если возвращаемое значение вызвавшей функции не нужно, она может его игнорировать :

```
void main(void) {...func1(5,3.256,&ch);...}
```

5.5. Более подробно о подпрограммах вообще и функциях Си.

Итак, подпрограммы - это языковая реализация основного и вспомогательных алгоритмов, на которые как правило распадается решение общей задачи в процедурном программировании. В самом деле, даже решение достаточно узкого класса задач на компьютере, например, вычисление числа Фибоначчи с произвольным номером, потребует от нас выполнения ряда вспомогательных алгоритмов.

В частности, основная программа должна узнать, какой номер числа Фибоначчи интересует пользователя нашей программы. Можно ориентировать нашу программу на диалог с пользователем по этому поводу, то есть вывести на экран запрос типа : «Введите номер интересующего вас числа Фибоначчи» и принимать ввод пользователя с запоминанием введенного номера. Можно ориентировать программу на ввод пользователем номера числа сразу после имени программы в командной строке. В первом случае нам понадобится подпрограмма (вспомогательный алгоритм) диалога, во втором - подпрограмма извлечения заданного номера из префикса программного сегмента. В свою очередь, подпрограмме диалога могут понадобиться вспомогательные алгоритмы сохранения, очистки, восстановления содержимого экрана, чтобы не вести экранный ввод - вывод на фоне предыдущих записей, алгоритмы установки курсора для эстетического оформления вопросов и ответов в диалоге и т д.

Таким образом, решение общей задачи даже в простейших случаях представляет собой многоступенчатый процесс расчленения на все более простые действия. Конечно, можно составить весь алгоритм без явного выделения вспомогательных действий, но такую программу будет не только неудобно составлять и отлаживать, ее будет чрезвычайно сложно читать и практически невозможно совершенствовать.

Другой причиной необходимости подпрограмм в алгоритмических языках является наличие в разных программах

большого количества однотипных, шаблонных действий - прием ввода пользователя с клавиатуры, вывод строк на экран или печатающее устройство, сортировки чисел по возрастанию в числовых массивах или сортировка списков фамилий по алфавиту, различные алгоритмы поиска необходимых данных в массивах - этот перечень стереотипных работ можно продолжать долго. Совершенно очевидна целесообразность однократного составления таких шаблонных, часто употребляемых вспомогательных алгоритмов, хранение их на диске и последующее использование в самых разнообразных задачах.

Но тогда программа должна будет состоять из главной подпрограммы (она нужна для определения начала и конца общего алгоритма), которая что-то делает сама, а что-то поручает вызываемым на выполнение вспомогательным подпрограммам, которые в свою очередь делают что-то сами и возможно вызывают на выполнение другие вспомогательные подпрограммы и т д. Каждая вспомогательная подпрограмма должна быть достаточно универсальной, т.е. выполнять свою работу для различных вариантов данных, а, следовательно, при ее вызове на выполнение ей необходимо будет передать данные или их адреса для обработки. После завершения подпрограмма должна возвратить управление в точку вызова - точнее, на оператор, следующий за вызовом подпрограммы. Поэтому механизм работы с подпрограммами требует запоминания точки ухода в подпрограмму и возврата из нее.

В языке Си функция - это логически самостоятельная именованная часть программы, которой могут передаваться параметры и которая может возвращать значение. Стандарт ANSI языка Си предполагает следующий формат определения функции:

```
[возвр тип данных] имя_функции (список аргументов | void)
{
описание локальных данных
операторы, в т. ч. Вызова других функций
[return] (выражение)
}
```

Совокупность предложений в фигурных скобках - тело функции. По синтаксису языка Си нельзя в теле одной функции определять другую - т.е. не допускается вложенное определение функций. Если поле «возвращаемый тип» опущено, считается, что возвращаемый тип - целый, а при слове `void` - функция не возвращает значения. Поле «имя

функции» - это особый тип указателя на функцию; его значение - адрес входной точки функции. В поле «список параметров» - любая комбинация типов и имен, разделенных запятой и составляющих список формальных параметров. При каждом вызове функции происходит замена формальных параметров значениями фактических с размещением их в стеке. Список формальных параметров может быть и пустым или содержать слово `void`.

Стандарт ANSI требует для функций, возвращающих не целое значение предварительного объявления, называемого прототипом функции:

```
float func1(int,char *,unsigned);
char * func2(int var1,int var2);
```

В прототипах функций можно указывать только типы формальных параметров без имен переменных, так как прототип содержит только заголовок функции без тела - это только объявление, которое служит компилятору для проверки правильности составления определения функции и вызовов ее на выполнение.

Функция в процессе выполнения извлекает копии аргументов из стека. При передаче параметров в функцию (в стек) выполняется дополнительно преобразование их к типу, который указан в прототипе.

Существует 2 способа передачи параметров:

* передача значений;
* передача адресов переменных.

Вызов функции с передачей адресов позволяет строить функции с доступом к массивам и другим протяженным структурам данных.

В Си допускается наличие функций с переменным числом аргументов - признаком такой функции является многоточие в списке параметров прототипа. Такая функция должна иметь способ определения точного числа параметров при каждом вызове - или число передаваемых параметров задается первым аргументом, или последний параметр должен быть индикатором конца списка, т.е. иметь обусловленное значение, которое не может принимать ни один параметр. Функции с переменным числом аргументов должны иметь хотя бы 1 фиксированный параметр, чтобы было к чему привязаться при извлечении их из стека.

Для упрощения работы с переменным числом аргументов определен особый тип, который и используется для описания указателя на начало списка фактических аргументов:

```
typedef void* va_list;
```

Для задания начального значения указателю типа va_list используется макро va_start:

```
void examp(int arg1,...)
{
  va_list ptr;
  va_start(ptr,arg1);
}
```

Доступ к значениям в списке фактических параметров выполняется через макро va_arg():

```
void example(int,...);
const EOL=0  /*ограничитель списка параметров */
void main(void)
{int var1=5,var2=6,var3=7,var4=8,var5=9;
example(var1,EOL);
example(var1,var2,EOL);
example(var1,var2,var3,var4,var5,EOL);
}

void example(int arg1,...)
{int number=1,value;
va_list ptr;
va_start(ptr,arg1);
value=va_arg(ptr,int);
Здесь можно проверить value на 0 - не конец ли списка
...
}
```

Си поддерживает рекурсивные вызовы функций - как непосредственные (самой себя), так и опосредованные (вызов текущей функции из вызванной ею функции). Функция `main` в этом смысле не является исключением, хотя ее опосредованные вызовы - это все же экзотика.

Преимуществом рекурсии есть компактный код, а недостатки - потери времени на вызовы и возможность переполнения стека.

ПЕРЕДАЧА ПАРАМЕТРОВ В ГЛАВНУЮ ФУНКЦИЮ `main()`.

Аргументы функции `main` - это аргументы программы в целом, задаваемые пользователем программы при ее вызове из командной строки операционной системы.

Для доступа к аргументам функции `main()` ее необходимо описать в одной из следующих форм:

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

[тип] `main`(`int` argc,`char` *argv[],`char` * env[])

{…}

В первом случае функция будет иметь доступ к словам командной строки, а во втором - еще и к строкам среды программы.

Argc - число слов в командной строке;

*argv[] - массив указателей из argc элементов; при этом argv[0]-спецификация запускаемого на выполнение файла;

*env[] - массив указателей завершающийся нулевым указателем, а каждый его элемент указывает на строку среды программы)

В заголовочном файле dos.h определены 2 глобальных переменных

– _argc (целочисленная) - в нее помещается количество разделенных пробелами слов в командной строке DOS;
– _argv (массив указателей на char) -в нее помещены все слова командной строки и их можно получить простым индексированием этого массива, например _argv[0] - это имя исполняемого файла самой программы, _argv[1] - указатель на строку с первым словом в командной строке после имени исполняемого файла.

При использовании этих переменных функция main() может быть и без параметров.

УКАЗАТЕЛИ НА ФУНКЦИИ. МОДИФИКАТОРЫ near,far,huge.

Синтаксис Си позволяет применять указатели для вызова функций. Имя функции в Си - это указатель-константа на функцию, равный адресу точки входа функции. Помимо константных возможно также описание указателей - переменных на функции:

Возвр_тип (* имя_ук) (список_арг);

Указатели на функции используются в следующих случаях:

1- Многие библиотечные функции в качестве аргумента получают указатель на функцию. Например, функция сортировки qsort() получает 4-м аргументом указатель на составляемую пользователем функцию сравнения сортируемых элементов
2- Использование указателей на функции в качестве аргументов позволяет разрабатывать универсальные функции - например, численного решения уравнений, интегрирования и дифференцирования

3- Указатели на функции могут использоваться для косвенного вызова резидентных программ, точка входа в которые записана в известное место памяти, например, по одному из неиспользуемых векторов прерываний

Как и обычные переменные, указатели на функцию могут объединяться в массивы. Описать и инициализировать массив указателей на функции можно так:

Есть набор функций:

```
int cmp_year(const void*,const void*);
int cmp_price(const void*,const void*);
int cmp_title(const void*,const void*);
int cmp_name(const void*,const void*);
int cmp_tot(const void*,const void*);
```

Организуем и инициализируем массив указателей:

```
int(*fcmp[5])()={cmp_name,cmp_title,cmp_year,cmp_price,cmp_tot};
```

Доступ к элементам массива - как обычно:

```
int index=0;
fcmp[index](ptr1,ptr2);
```

Меняя значение индекса, можно вызывать другие функции - например, в цикле.

Указатели на функции тоже могут иметь тип `near`,`far` или `huge`. Модификаторы типа функции должны совпадать у прототипа и определения:

`char far *far` func(`void`)-это `far`-функция, возвращающая `far`-указатель на тип `char`.

Тесно связано с функциями понятие видимости переменных и функций.

КЛАССЫ ПАМЯТИ И ВИДИМОСТЬ ПЕРЕМЕННЫХ.

Для установления связей между именами переменных и соответствующими им объектами в памяти компилятору необходимо сообщить как минимум 2 сведения: тип и класс хранения.

Тип определяет размер памяти под переменные, а также способ интерпретации хранимых в памяти чисел. Класс хранения определяет место, где будет расположен объект (регистры процессора, сегмент данных программы или сегмент стека) и одновременно время жизни объекта (все время выполнения программы или только на время выполнения ее определенной части).

Класс хранения задается либо явно, либо косвенно, когда компилятор определяет его сам по расположению описания в программе. С точки зрения времени жизни различают 3 типа переменных: статические, локальные, динамические. С

понятием «класс хранения» тесно связано и понятие «видимости» или области определения идентификатора (имени). Область определения - это та часть программы, в пределах которой идентификатор доступен для обработки скрытой под ним информации. Есть несколько типов областей определения (видимости)

- в пределах блока (локальная видимость);
- в пределах функции;
- в пределах файла.

При необходимости сделать идентификатор видимым явно используется объявление его с ключевым словом `extern`.

ОБЛАСТЬ ОПРЕДЕЛЕНИЯ И ВИДИМОСТЬ.

Текст программы (функция `main()` и все остальные функции) может быть размещен целиком в одном текстовом файле. Это не всегда удобно, а иногда невозможно. Тогда текст программы размещается в нескольких текстовым файлах. Для объединения они компонуются совместно. Информация о всех объединяемых файлах помещается в файл проекта. Для каждого исходного текстового файла порождается свой объектный файл, а затем они объединяются компоновщиком в загрузочный модуль.

Объектный файл - не готовый к выполнению программный код. Для обеспечения работоспособности отдельные объектные модули должны быть связаны перекрестными ссылками. Эту работу выполняет компоновщик, а необходимая для него информация готовится либо программистом при явном указании классов хранения, либо компилятором по размещению объявлений в тексте программы.

Если идентификатор описан внутри блока { }, он имеет локальную область определений, ограниченную размерами блока. Отсюда следует, что все переменные, описанные внутри функций, в том числе и формальные параметры - локальные.

Если идентификатор описан вне каких-либо блоков, он является глобальным, его область определения начинается в месте описания и продолжается до конца файла. Чтобы идентификатор был видимым и выше точки описания или из других файлов используется атрибут `extern`:

```
extern int varl;
```

Внешние (`extern`) переменные нельзя инициализировать (присваивать значения) в месте описания.

Сначала компоновщик, встретив объект `extern` разыскивает соответствующий ему глобальный идентификатор в текущем файле, если не находит - то просматривает

глобальные данные других файлов проекта, если нет и там - поиск продолжается в библиотеках.

Внешние переменные - один из способов передачи функциям аргумента и возврата из функций необходимых значений. Взаимодействующие функции используют в этом случае общие области памяти. Это экономичнее и быстрее, чем передача фактических параметров через стек, но чревато опасностью «неожиданного» изменения содержимого общих ячеек памяти при несогласованной работе использующих их подпрограмм.

Частным случаем локальных переменных являются регистровые - объявленные ключевым словом `register` (переменные некоторых целых типов и указатели). Компилятор будет хранить их в регистрах, если есть свободные, а если нет - в стеке, как обычные автоматические. Использование `register` переменных позволяет ускорить работу программы.

СТАТИЧЕСКИЕ ПЕРЕМЕННЫЕ - КЛАСС `static`.

Статические объекты существуют все время выполнения программы. Память под статические переменные выделяется в сегменте данных. По умолчанию статическими являются строковые литералы и все глобальные переменные.

Но при явном объявлении со словом `static` любой объект, в том числе локальный, может быть отнесен к статическому. Компилятор заполнит их нулевыми байтами или кодами начальной инициализации.

Локальные статические объекты все равно не видимы из других функций, хотя имеют статическое время жизни. Спецификатор `static` может использоваться не только для данных, но и для функций - статические функции невидимы для загрузчика, т.е. недоступны из других файлов.

`void` func8(`void`) {`static int` count;...}

Переменная count будет сохранять свое значение от вызова к вызову подпрограммы func8, но доступ к ней из других подпрограмм закрыт - она статическая локальная.

5.6. Обработка текстов программ.

Текст программы (даже «пустышки», как в нашем случае), чтобы стать исполняемым файлом, должен пройти обработку программой-компилятором (в результате будет создан так называемый объектный файл с расширением .obj) и программой - редактором адресных ссылок tlink.exe (будет создан исполняемый .exe - файл). В системе разработки программ поставляются как минимум 2 компилятора -

компилятор командной строки (tcc.exe, bcc.exe или bcc32.exe для продуктов фирмы Borland) и компилятор интегрированной среды IDE (tc.exe, bc.exe или bcw.exe для той же фирмы). Интегрированная среда представляет собой объединенные в одну систему многооконный редактор текстов, компилятор, редактор связей (компоновщик) и отладчик программ.

Обе программы должны быть «настроены» на определенный режим работы с помощью уточняющих аргументов - опций. Для интегрированной среды вы найдете эти опции в меню Options, а для командной строки вы их узнаете, запустив компилятор без параметров. В простейшем случае в командной строке надо указать модель памяти и имя файла с текстом программы на Си:

bcc -ml my_prog.cpp

(компиляция в модели памяти large)

При этом компилятор сам вызовет компоновщик, если не обнаружит синтаксических ошибок, иначе выведет на экран список ошибок с указанием их типов и номерами строк, в которых они совершены по его предположению. Вам удастся на начальной стадии обучения так запутать синтаксический анализатор компилятора, что он может и ошибаться в номере строки - поэтому, если вы не видите ошибки в указанной им строке, то поищите ее раньше по тексту - особенно обратите внимание на скобки и разделители между операторами ‘;’.

5.7. Управляющие структуры в Си.

Мы уже обсуждали управляющие структуры при освоении словесных описаний алгоритмов и блок - схем. К управляющим структурам, требующим специального синтаксиса записи относятся условное выполнение одного или группы операторов (ветвление) и циклическое (повторяющееся) выполнение одного или группы операторов.

Различают двух и многоальтернативные ветвления.

Различают также несколько видов циклов:

- ♦ циклы с предусловием, работающие по схеме **ПОКА УсловиеВыполняется ВыполнятьБлокОператоров**
- ♦ циклы с постусловием, реализующие схему **ВыполнятьБлокОператоров ПОКА УсловиеВыполняется**

  Отличия их очевидны: цикл с предусловием может не выполниться ни разу, если не выполняется предварительно проверяемое условие; цикл с постусловием всегда выполнится хотя бы 1 раз, т. К. Проверка условия вынесена в конец цикла.

- бесконечные циклы, в которых условие прекращения повторения блока операторов формируется непосредственно в блоке операторов, а выход из цикла осуществляется с помощью специальных операторов выхода за пределы тела цикла.

  ВЫПОЛНЯТЬ_БЕСКОНЕЧНО БлокОператоров
- циклы с фиксированным количеством итераций, представляющие собой частный случай циклов с предусловием и дополнительном формировании счетчика «оборотов»; работают они так: в специальную переменную-счетчик заносится некоторое начальное положительное или отрицательное значение, сравнивается с заданным числом и принимается решение о выполнении блока операторов; после каждого прохода блока операторов это число уменьшается (или увеличивается) на заданный шаг. Выход из цикла осуществляется, когда счетчик станет больше (или меньше) заданного числа. Циклы этого типа в различных инструментальных системах имеют различные модификации схем функционирования.

УСЛОВНЫЕ ОПЕРАТОРЫ В Си.

Операторы ветвления выбирают в программе из группы альтернатив возможное продолжение вычислительного процесса. Выбор выполняется по значению заданного выражения. В Си 2 оператора ветвления:

- 2-х альтернативный `if`(условие)...`else`
- многоальтернативный переключатель `Switch`.

Оператор `if` имеет следующую общую форму записи:

```
if(выражение)
    блок продолжения по истине
else
    блок продолжения по ложь
```

При выполнении оператора `if` сначала вычисляется логическое выражение в скобках; если результат истина (не нуль!), выполняется стоящий вслед за скобками блок операторов. Если результат ложь (равен 0), то блок истины пропускается ; если присутствует ключевое слово `else` (иначе), то выполняется блок ЛЖИ. Блоки истинности и ложности сами могут быть операторами `if`, образуя вложенные ветвления. Компилятор интерпретирует вложенные `if`, сопоставляя каждое слово `else` с последним перед ним словом `if`, не

имеющим своего else. Соответствие ищется в пределах блока, в котором заключен if. Внутренние и внешние блоки при этом не рассматриваются. Если соответствия не найдено, считается, что if не имеет ветки else.

Если блок условно выполняемых операторов не заключен в фигурные скобки, то выполняется только один оператор, следующий за if или else.

Пример:

```
int a,b;
if (b >o)
a=1;
else
if(b==o)
a=o;
else a=-1;
```

То же самое:

```
int a,b;
if(b>o)
a=1;

if(b==o)
a=o;
else a=-1;
```

Операция условия ?:является частным случаем оператора if...else. Ее вид:

Логическое выражение? Блок истины : Блок лжи

Пример:

```
int a=4,b=3,c;
c=a>b? A*a+b*b:0; /*c=a*a+b*b=25*/
```

Фрагмент if(выражение) x=y; else x=y+4; эквивалентен x=(выражение) ? y:y+4, что получается короче.

Операторы switch и break.

Выбор одного варианта из многих возможных можно осуществить с помощью многократно вложенных if( )...else, но более удобно использовать оператор переключения switch, общий вид которого следующий:

```
switch (логическое выражение)
{ case константа 1:блок 1;[break;]
...
case константа i:блок i;[break;]
...
case константа n: блок n;[break;]
[default:блок n+1;]
}
```

Выполняется он так : сначала вычисляется логическое выражение, тип его должен быть одним из целых. Это

выражение поочередно сравнивается с каждым константным выражением в блоке и при совпадении значений выполняется следующий блок, и если присутствует `break`, то осуществляется выход из `switch`. В противном случае работа продолжается до встречи с `break` или до конца блока `switch`.

ЦИКЛЫ В Си.

В языке Си есть удобные операторы циклов. Всего их три:

1- Цикл с предусловием `while`(логическое выражение) блок операторов;
2- Цикл с постусловием `do` блок операторов `while`(логическое выражение);
3- `for`(1-я секция; логическое выражение ;2-я секция);

Оператор `while` организует повторение блока операторов, пока логическое выражение не станет ЛОЖЬ (0). Это цикл с предусловием, т.к. истинность логического выражения проверяется перед входом в блок (тело цикла). Если необходимо обеспечить выполнение блока хотя бы раз, использует цикл с постусловием `do...while`(логическое выражение) - сначала выполняется блок, а затем проверяется истинность условия. Повторение продолжается, пока выражение в скобках после `while` не примет значение ЛОЖЬ (0).

Наиболее сложная и универсальная форма цикла - оператор `for`.

В его 1-й секции, заканчивающейся точкой с запятой, могут быть выполнены (через запятую) все необходимые до первого входа в тело цикла операторы присвоений - в принципе они могут размещаться и до заголовка цикла, тогда 1-я секция будет отсутствовать, вернее, будет представлена только знаком ‘;’.

2-я секция полностью аналогична условию в цикле `while`

В 3-й секции размещаются через запятую те простые операции над любыми переменными, которые должны выполняться после каждого прохода тела цикла.

Выполнение осуществляется в следующей последовательности:
если 1-я секция присутствует, то выполняются записанные в ней операции; если 2-я секция присутствует, то вычисляется содержащееся в ней логическое выражение и при его истинности начинается выполнение блока операторов в теле

цикла - если при этом встретится оператор `break`, то произойдет прекращение итераций и принудительный переход на следующий за циклом оператор, а при встрече с оператором `continue` осуществляется принудительный досрочный переход к выполнению секции 3. После завершения блока операторов в теле цикла или выполнения оператора `continue` выполняются операции в секции 3, после чего осуществляется переход к выполнению секции 2 и т. Д.

Благодаря этим свойствам цикл `for` универсален и может использоваться вместо любого типа циклов. Например, вместо цикла `while` он записывается просто:

`for`(;Условие;) {Блок операторов}

Для замены цикла `do` {Оператор }`while`(Условие); цикл `for` записывается так:

`for`(Оператор; Условие;) Оператор

Бесконечный цикл моделируется циклом `for` так:

`for`(;;) {...`if`(Условие) `break`;...}

Для замены цикла со счетчиком

`for`(Счетчик=Значение; Условие; Наращивание счетчика с произвольным шагом) { Блок операторов}

При использовании циклов необходимо следить, чтобы не происходило бесконечного зацикливания.

Оператор `continue` может использоваться только в теле циклов, он передает управление во всех видах циклов на вычисление выражения, определяющего допустимость продолжения выполнения тела цикла (для цикла `for` сначала на секцию 3).

Оператор `return` обеспечивает выход из текущей функции в точку ее вызова (возврат из подпрограммы).

5.8. Краткий обзор некоторых библиотек.

5.8.1. Общие замечания.

Вы получили уже много сведений о компьютере, операционной системе, языке программирования Си - естественно возникает вопрос о том, когда же будет написана первая программа.

Дело в том, что Си - «маленький язык высокого уровня» и в силу своей малости не имеет даже встроенных операторов ввода - вывода. Вся «мощь» языка - в объектных библиотеках, содержащих скомпилированные наборы подпрограмм управления компьютерными ресурсами и разнообразного сервиса, использующие, в свою очередь, при необходимости услуги операционной системы через механизм программных прерываний. Эти подпрограммы извлекаются из библиотеки и включаются в вашу программу программой tlink.exe компоновки загрузочного модуля.

Практически невозможно сделать сколько-нибудь полный обзор всех наработанных библиотек Си и C++, поэтому мы ограничимся лишь самыми необходимыми на начальной стадии обучения. В профессиональном программировании приходится тратить немало времени на изучение возможностей, предоставляемых библиотеками, но, как правило, программист специализируется в некоторой прикладной области и использует наиболее подходящие для этой области библиотеки.

Механизм использования библиотечных подпрограмм состоит в следующем. Помимо скомпилированных до объектных модулей комплексов подпрограмм, которые включаются в наши программы компоновщиком, в наше распоряжение предоставляются так называемые заголовочные файлы текстового формата - они содержат объявления необходимых для работы с библиотекой типов данных, именованных констант, переменных и прототипы содержащихся в объектной библиотеке подпрограмм (функций). Эти файлы имеют обычно расширение .h (от слова header - заголовок). Приведенные в заголовочных файлах прототипы библиотечных подпрограмм позволяют осуществлять их вызовы на выполнение в соответствии с их заголовками, если соответствующий заголовочный файл включен в программу директивой препроцессора `#include`. Например, для работы с библиотекой математических подпрограмм необходимо включить в программу директиву

`#include` <math.h>

для работы с подпрограммами обработки строк директиву

`#include` <string.h> и т.д.

5.8.2. Стандартные математические функции

(прототипы -в заголовочных файлах stdlib.h,math.h)

`int` abs(`int` x) - возвращает абсолютное значение целого аргумента

double acos(double x)- угол в радианах от -pi/2 до pi/2 для заданного косинуса от -1 до 1

double asin(double x) - угол для заданного синуса

double atan(double x) - угол для заданного тангенса

double atan2(double y,double x) - арктангенс отношения y/x

double cabs(struct complex znum) - модуль комплексного числа

double ceil(double x) - ближайшее целое с округлением в большую сторону

double cos(double x) - косинус для угла x радиан

double cosh(double x) - косинус гиперболический

div_t div(int number, int denom) - делит целые с возвратом типа div_t из stdlib:

```
typedef struct
{int quot;/* частное */
int rem;/* остаток */
}div_t;
```

double exp(doble x) - возвращает e в степени x

double fabs(double x) - абсолютное значение x

double floor(double x) - ближайшее целое с округлением в меньшую сторону

double fmod(double x,double y) - остаток от деления, если частное - наибольшее целое

frexp(double value,int * exponent) - возвращает нормализованную мантиссу и порядок записывает по адресу exponent числа

double hipot(double x,double y) - гипотенуза 3-угольника с катетами x и y

double ldexp(double x, int exp) - число с мантиссой x и порядком exp

double log(double x) - натуральный логарифм

double log10(double x) - десятичный логарифм

unsigned long lrotl(unsigned long val,int count) - значение беззнакового целого, сдвинутого на count битов влево

unsigned long lrotr(unsigned long val,int count) - значение беззнакового целого после сдвига вправо

max(a,b),min(a,b) - максим и минимум любых числовых типов double modf(double x,double *ipart) - возвращает дробную часть числа x, а целую - по адресу ipart

double pow(double x,double y) - x в степени y

double pow10(int p) - 10 в степени p

int rand(void) - случайное число от 0 до 32767 (RAND_VAX в stdlib.h)

int random(int num) - случайное целое от 0 до num-1

void randomize(void) - старт генератора случайных чисел

unsigned _rotl(unsigned value,int count) - циклические сдвиги целых чисел влево

unsigned _rotr(unsigned value,int count) - вправо

double sin(double x) - синус числового аргумента в радианах

double sinh(double x) - гиперболический синус

double sqrt(double x) - квадратный корень x

void srand(unsigned seed) - начальное целое seed для генерирования случайных целых

double tan(double x)

double tanh(double x)

5.8.3. Функции классификации и преобразования символов

(определения - в заголовочном файле ctype.h)

Все функции классификации возвращают ненулевое значение(истина), если анализируемый символ соответствует названию функции

isalnum(c), isalpha(c), isascii(c), iscntrl(c), isdigit(c), isgraph(c), islower(c) - строчная буква, isprint(c), ispunct(c), isspace(c), isupper(c), isxdigit(c) - 16-ричная цифра

_tolower(c)-преобразует прописную в строчную, возвращает код строчной

toascii(c)-возвращает код ASCII от 0 до 127

tolower(c)- то же, что и _tolower, но с предварительной проверкой на принадлежность к прописной

toupper(c) - к верхнему регистру

5.8.4. Функции для работы с блоками памяти

(прототипы - в файлах mem.h и string.h)

void *memccpy(void *dest,const void *src,int c,size_t n) - копирует блок памяти из адреса src в адрес dest до встречи с символом c или до переноса n символов

void *memchr(const void *s,int c,size_t n) - возвращает указатель на байт в s, содержащий c или NULL

int memcmp(const void s1,const void s2,size_tn) - сравнивает первые n байтов буферов s1 и s2, возвращая 0 при совпадении отрицательное число если s1<s2 и положительное при s1>s2

void memcpy(void *dest,const void *src,size_t n) - копирует n байтов из src в dest

int memicmp(const void s1,const void s2, size_tn) - сравнивает первые n байтов буферов s1 и s2, возвращая 0 при совпадении, отрицательное число, если s1<s2 и положительное при s1>s2 с предварительным преобразованием в строчные

void memmove(void *dest,const void *src,size_t n) - то же что и memcpy(), но с корректным переносом для перекрывающихся буферов

void *memset(void *s,int c.size_t n) - заполняет n байтов буфера s символом c

void movedata(unsigned srcseg,unsigned srcOfs,unsigned dstseg, unsigned dstOfs,size_t n) - перемещение данных

void swab(char *from,char *to, int n) - копирует с заменой чет-нечет

5.8.5. Функции обработки текстовых строк

(прототипы в string.h)

char * strcat(char *dest,char *source) - конкатенирует (сцепляет) строки dest и source;

char *strncat (char *dest,char *souree,unsigned n) - конкатенирует n символов строки dest к строке souree;

char *strchr (char *souree,char ch) - поиск в строке source первого вхождения символа ch;

char *strcmp (char *s1,char *s2) - сравнивает строки s1 и s2, возвращает 0 (равны), < 0 (s1<s2), > 0 (s1 > s2).

char *strncmp (char *s1,char *s2,int n) - то же, что и предыдущая функция, но сравниваются только первые n символов обеих строк.

char *stricmp (char *s1,char *s2) - то же, что и предыдущая функция, но не проверяется регистр букв.

char *strnicmp(char *s1,char *s2,int n) - то же, для n сравниваемых символов без проверки регистров.

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

char * strncpy(char * dest, char * source, int n) - копирование n символов из source в dest.

int strlen(char * s) - длина строки, но без нуль-символа конца строки.

char * strlwr(char * s) - перевод в нижний регистр (только латинские символы)

char * strupr(char * s) - перевод в верхний регистр (только латиница)

char * strdup (char * s) - создает в куче копию строки s, возвращая ее указатель.

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

char * strnset (char * s, char ch, unsiqned n) -то же для первых n символов строки s.

char * strrev (char * s) - инвертирует все буквы s, делая из слова по данному указателю палиндром.

int strcspn (char * s1, char * s2) - длина начального сегмента s1,состоящая из символов, которых нет в s2.

char * strpbrk (char * s1, char * s2) - просмотр s1 до встречи с символом из s2.

char * strchr (char * s,char c) -последнее вхождение с в s.

int strspn (char * s1, char * s2) - длина начального сегмента s1, состоящая из символов, которые есть в s2.

char * strtok (char * s1, char * s2). В s2 содержатся разделители. Strtok возвращает указатели на слова.

В стандартной библиотеке Си stdio.h есть 2 функции ввода-вывода в строки с предварительным форматированием :

int sprintf(char * buffer, const char *format [, argument,,..]); принимает список аргументов, применяет к каждому из них спецификатор формата из строки формата и помещает сформатированные данные в строку

int sscanf(const char *buffer,const char* formаt[, address,...]); посимвольно читает последовательность входных полей из строки, каждое поле форматируется по спецификации формата и записывается по адресам, переданным в качестве аргументов

При использовании строковых функций вы должны быть уверены в том, что они заканчиваются символом ‘\0’. Для литералов этот символ добавит сам компилятор, но при посимвольном заполнении ответственность за символ окончания строки лежит на программисте.

5.9. Управление ресурсами - динамическое управление памятью в ДОС и Си.

5.9.1. Основные сведения.

При ориентации прикладного программирования на сидящего за компьютером пользователя - служащего обрабатываемые нашей прикладной программой данные чаще всего размещаются в подготовленных этим пользователем текстовых файлах. Формат текстового файла обычно свободный и размер размещенных там данных на стадии составления программы неизвестен. Поэтому в подавляющем большинстве реальных программ отвести память под обрабатываемые данные на стадии составления и компиляции программы невозможно - потребность в этой памяти программа узнает только на стадии выполнения, когда ей сообщат, например, имя обрабатываемого файла и количественные параметры содержащихся в нем данных.

Когда мы составляем нашу программу, то предполагаем, что она будет способна работать с данными самых различных размеров. Например, если мы составляем программу для решения систем линейных уравнений, то мы не знаем на стадии составления, какого порядка будет заданная пользователем система, т. е. сколько строк и столбцов будет в пользовательской матрице коэффициентов (пользователь скорее всего помесит ее в файл и сообщит его имя и размеры матрицы в командной строке при запуске программы на выполнение). Обычная схема работы нашей программы выглядит примерно так: разбирается командная строка для определения, например, имени файла с исходными данными, проверяется наличие этого файла, выполняется его открытие для чтения, определяется его длина - размер оперативной памяти для размещения подлежащих обработке данных - и мы встаем перед необходимостью запросить у операционной системы память вне программы, в так называемой куче для пересылки туда данных из файла, так как обрабатывать процессор может только данные, лежащие в оперативной памяти. Если алгоритм обработки данных позволяет, можно считывать данные по частям, а если нет - надо переписать весь файл в оперативную память. Часть памяти, выделяемая программе на стадии ее выполнения, называется динамически распределяемой, в отличие от памяти, резервируемой компилятором при размещении статических данных.

ФУНКЦИОНАЛЬНЫЕ ВЫЗОВЫ MS DOS ДЛЯ УПРАВЛЕНИЯ ПАМЯТЬЮ.

MS DOS - однопрограммная однопользовательская операционная система, способная оперировать адресным пространством не более 1Мбайт. В ее состав включены 3 подпрограммы 21h-го прерывания для управления памятью

AH=48H - выделение памяти

AH=49H - освобождение памяти

AH=4AH - изменение размера блока памяти

О возникновении ошибок при выполнении функций управления памятью сообщается установкой флага переноса, а в регистр AX записывается код ошибки:

AX=7 - разрушен блок управления памятью

AX=8 - недостаточно свободной памяти

AX=9 - недопустимый адрес блока памяти

Чтобы отслеживать распределение памяти, ОС ведет специальный связный список управляющих блоков управления памятью - **MCB *(Memory Control Block***). Каждый MCB занимает целый 16-байтовый параграф и непосредственно предшествует собственно блоку памяти непрерывной области, начинающейся на границе параграфа. Специальная внутренняя переменная ОС хранит указатель на 1-й MCB цепочки. Значимыми являются 5 первых байт MCB:

байт 0 - ASCII-символ ‘Z’ код 5A, если данный блок последний в цепочке MCB блоков или символ ‘M’ код 4DH в противном случае

байты 1-2 равны 0, если блок свободен или содержат PID программы, для которой распределен блок

байты 3-4 содержат размер блока в параграфах

Номер параграфа или, что то же самое, сегмент адреса, по которому располагается следующий MCB - блок в цепочке, определяется по формуле:

seg_mcb=seg_mcb_old+length_old+1

Новые блоки памяти и, следовательно, MCB-блоки в цепочке создаются и модифицируются ОС в случаях:

I. распределения нового блока памяти

II. изменения размера существующего блока

III. при загрузке и запуске программы на выполнение

IV. при резидентном завершении программы

При распределении нового блока AH=48H в регистр BX заносится только длина создаваемого блока в параграфах.

ОС отыскивает 1-й MCB-блок в цепочке, помеченный как свободный, размер которого превышает запрошенный. Если 2 или больше подряд расположенных блоков помечены как свободные, они рассматриваются как единый блок. MCB выбранного блока корректируется:

√ в байты 1-2 записывается номер параграфа текущего PSP (PID активной программы)
√ в байты 3-4 записывается размер созданного блока. Если размер запрошенной памяти меньше размера свободного блока, на границе остатка свободной памяти создается MCB и включается в цепочку MCB-блоков.

При загрузке и запуске программы на выполнение MS DOS отдает COM-программе всю свободную память (так как COM-файлы не имеют заголовков с данными о необходимой программе памяти, ОС вынужденно предполагает худший случай и отдает программе все что имеет) и запрос на выделение памяти из программы столкнется с отсутствием свободной памяти.

Для загрузки exe-файлов используются установленные компоновщиком поля в заголовке файла:

MIN_ALLOC - минимальное число параграфов, необходимое программе дополнительно к параграфам сегментов программы, данных и стека.

MAX_ALLOC - максимальное число параграфов, которые программа могла бы использовать при наличии

Если не удается выделить при загрузке MAX_ALLOC дополнительно к размеру самой программы, выделяется самый большой блок, если он не меньше размера программы и минимально допустимого дополнения.

В языках высокого уровня работу по освобождению ненужной программе памяти, как правило, выполняют специальные подпрограммы, присоединяемые к прикладной программе на стадии компоновки.

При выполнении AH=4AH в регистре ES задается сегмент начала модифицируемого блока, а в регистре BX - новый размер блока в параграфах.

Если размер блока уменьшается, на границе остатка создается новый свободный блок, а увеличение сработает, если следующий в цепочке блок свободный.

MS DOS имеет специальное средство создания резидентных программ - это так называемое TSR - завершение AH=31H 21-го прерывания или прерывание 27H (только для COM - файлов). При этом в регистре DX указывается размер

блока памяти от начала PSP, объявляемого резидентным, в байтах для 27H прерывания и в параграфах для 21-го функции 31H. Действия ОС при этом подобны выполняемым при изменении размера блока : блок, с которого начинается PSP, усекается до запрошенного размера, но остается занятым после завершения программы.

Функция освобождения блока памяти AH=49H 21-го прерывания требует задания в регистре ES номера параграфа начала блока. Этот вызов может выдаваться либо явно самой программой, либо ОС при завершении программы. В байты 1-2 освобождаемого блока записываются нули.

Многие профессиональные программы используют непосредственно MCB - блоки для определения наличия в памяти резидентных программ, их снятия, обнаружения и уничтожения вирусов и пр.

5.9.2. Динамическое управление памятью в Си.

ПРОГРАММА - ЗАГРУЗЧИК Си-ПРОГРАММ.

Во все скомпилированные и скомпонованные загрузочные модули, разработанные на языке Си, включается специальный модуль C0x.OBJ, содержащий 2 основные секции: загрузчик (или стартер) и секцию завершения. Структура и алгоритм работы загрузчика зависят от выбранной модели памяти - для каждой из них есть свой вариант загрузчика. Главная задача подключаемого модуля - сформировать загрузочный модуль в точном соответствии с представлениями программиста о его структуре, передать управление функции `main()`, а после ее завершения выполнить возврат в операционную систему.

Загрузчик выполняет следующие действия:

1. Сохраняет в кодовом сегменте адрес начала секции данных DGROUP.
2. Инициализирует в сегменте данных ряд внешних переменных: номер версии DOS, сегмент адреса начала PSP, сегмент адреса среды программы, сегмент адреса границы оперативной памяти и др.
3. Сохраняет в сегменте данных некоторые векторы прерываний (0,4,5,6 и др.) и устанавливает собственный обработчик прерывания 0 - он будет печатать «Divide error» и аварийно завершать программу при попытке деления на 0.
4. Формирует среду программы и аргументы, передаваемые в точку входа `main()` - эти аргументы доступны Си - программе, если точка входа описана как `main(int,char **,char **)`.
5. Освобождает неиспользуемую программой оперативную память.

6. Выполняет настройку программы на аппаратуру, если в программе есть код для математики с плавающей точкой и компиляция выполняется с опцией Emulation.
7. Настраивает регистры процессора в соответствии с моделью памяти.
8. Вызывает подпрограмму `main()`. После завершения `main()` модуль C0x.OBJ снова получает управление и выполняет действия по завершению программы:
9. Восстанавливает все сохраненные векторы прерываний.
10. Закрывает открытые при выполнении программы файлы.
11. Вызывает одну из функций MS-DOS завершения программы.

Программа располагается в 2-х блоках памяти: 1-й занимает среда программы, а 2-й - загрузочный модуль, построенный из сегментов с предопределенными именами. Весь программный код (1 или более сегментов) компонуется в секцию кода, все данные располагаются подряд в секцию данных. Ссылкой на начало секции является имя группы DGROUP. Кроме того, программа обязательно имеет стек в составе загрузочного модуля. И, наконец, в зависимости от модели памяти загрузчик создает пространство памяти под названием «куча» (heap) для динамического распределения. Для работы с кучей библиотека Си предоставляет следующий набор подпрограмм, подключаемых с помощью заголовочного файла alloc.h:

`void` * calloc(size_t nelem, size_t elsize) - выделяет место для nelem элементов по elsize каждый и обнуляет выделенную память, возвращая указатель типа `void` на выделенную память; тип size_t в настоящее время определен как `unsigned`.

`Unsigned` coreleft(`void`) (для моделей TINY, SMALL, MEDIUM)

`unsigned long` coreleft(`void`)

-возвращает размер свободной кучи (COMPACT, LARGE, HUGE)

`void far` *farcalloc(`unsigned long` nelem, `unsigned long` elsze) - выделяет из `far` - кучи место для nelem элементов по elsize каждый и обнуляет выделенную память, возвращая указатель типа `void far` * на выделенную память;

`unsigned long far`coreleft(`void`) - размер свободной дальней кучи

`void far`free(`void far` *block) - освобождает блок памяти в `far` - куче размером, который должен соответствовать выделенному ранее функциями `far`calloc, `far`malloc, `far`realloc для малых моделей памяти или calloc, malloc, realloc для старших моделей

void far *farmalloc(unsigned long nbytes)

void far * farrealloc(void far *block,unsigned long size) делает попытку перераспределить блок памяти block, в случае невозможности расширить блок до заданного размера возвращает NULL

void free(void* block) - освобождает выделенный malloc блок памяти

void *malloc(size_t size) -выделяет запрошенное в байтах количество памяти

void realloc(void *block, size_t size) -изменяет размер ранее выделенного блока памяти

5.10. Управление ресурсами компьютера из прикладных программ. Файловый обмен в языке С.

5.10.1. Общие сведения.

Как мы уже сообщали, устройства последовательного ввода-вывода трактуются операционной системой на логическом уровне как файлы последовательного доступа. Поэтому для написания относительно полнофункциональных программ нам придется освоить еще и методы управления ресурсами исполнителя наших программ и начнем мы с управления вводом-выводом, то есть с файлового обмена в Си.

Библиотечные подпрограммы Си для работы с файлами можно разделить на 2 группы - потоковые и префиксные. И те, и другие обращаются в принципе к одним и тем же вызовам MS DOS, но потоковые выполняют дополнительную буферизацию информации, что приводит к двойной буферизации - на уровне ОС и на уровне библиотечной подпрограммы.

Префиксные - это блок-ориентированные функции, обращающиеся к префиксным функциям ОС без дополнительной буферизации и их использование дает выигрыш в производительности при переносе сразу группы байтов за одно обращение к функции.

Для обеих групп функций файлового ввода-вывода возможны 2 режима доступа к файлу: текстовый и двоичный. В текстовом режиме производится трансляция символьных кодов 0DH, 0Ah - при чтении из файла в один символ новой строки ‘\n’, а при записи в файл этот символ преобразуется в пару CR LF; кроме того, при считывании символа Ctrl Z считается, что

достигнут конец файла и прочитать информацию после этого символа в текстовом режиме не удается.

В двоичном режиме никакого преобразования байтов не происходит и не делается никаких предположений об их смысле.

Режим доступа к файлу задается при его открытии через параметры библиотечной функции или присвоением значения специальной внешней переменной _fmode, описанной в <fcntl.h> или <stdlib.h> и могущей принимать 2 значения:

O_BINARY или O_TEXT

По умолчанию принято O_TEXT.

5.10.2. Потоковый ввод-вывод.

Поток (stream) - абстрактное понятие, связанное с переносом данных между устройством чтения-записи и потоком этих данных, «движущихся» мимо устройства (воображаемой головки). Потоком информации можно считать данные на диске или поступающие с клавиатуры или дисплей, или порты ввода-вывода - любое устройство с последовательной структурой информационных записей.

Для каждого файла, открытого для доступа через поток, Си создает внутреннюю структурную переменную по шаблону FILE:

```
typedef struct{
short level;            //флаг буфер пуст или полон
unsigned flags;         //флаги состояния файла
char fd;                //префикс файла
unsigned char hold;     //непереданный символ
short bsize;            //размер буфера
unsigned char *buffer;//начало буфера, строки
unsigned char *curp;    //текущая позиция для доступа в буфере
unsigned istemp;        //флаг временного файла
short token;            //маркер действительности файла
}FILE;
```

Флаги состояния:

_F_RDWR - для чтения- записи
_F_READ - только для чтения
_F_WRITE - только для записи
_F_BUF - имеет динамический буфер
_F_LBUF - построчно буферизуемый
_F_ERR - индикатор наличия ошибки
_F_EOF - индикатор наступления конца файла

_F_BIN - открыт в 2-ичном режиме
_F_IN - осуществляется чтение
_F_OUT - осуществляется запись
_F_TERM - файл является терминалом

Эти значения - битовые позиции в поле flags и использовать их следует как маску для операции побитового И:

```
if(ptr->flags & _F_READ)...
```

На таком непосредственном использовании описания файла f построены макро:

`#define` ferror(f) ((f)->flags & _F_ERR) - не 0, если при доступе возникла ошибка
`#define` feof(f) ((f)->flags & _F_EOF) - не 0 если достигнут конец файла
`#define` fileno(f) ((f)->fd - префикс файла
Чтобы эти макро давали свежую информацию, надо очищать поле flags с помощью:

```
void clearerr(FILE*fp)
```

Открытие файла:
FILE * fopen(`const char` * Filename, `const char` * mode);
Строка mode может принимать значения:
r - только для чтения
w - только для записи с усечением
a - для дополнения
r+ для обновления
w+ для обновления
a+ для обновления
b двоичный режим
t текстовый режим

Общепринятая схема:

```
#include <stdio.h>
FILE *f;
main(){ if((f=fopen(«myFile.txt»,»r»))!=NULL... else...}
```

Закрытие файла:

```
int fclose(FILE*fp)
int fcloseall(void)
```

Переоткрытие файла с другими правами доступа:
FILE *freopen(`const char` *Filename, `const char` *mode, FILE *stream)
Открытие безымянного временного файла, уничтожаемого при закрытии или завершении программы:

FILE *tmpfile(void)

Внимание! При задании имени файла с маршрутом доступа не забывайте ставить разделитель между именами каталогов в виде двойного обратного слэша:

c:\\dir1\\dir2\\Filem\\name

или

c:/dir1/dir2/Filem/name

Стандартные потоки: stdin, stdout, stderr, stdaux, stdprn являются указателями на переменные типа FILE и могут использоваться наравне с возвращаемыми функцией fopen.

Библиотеки Си содержат функции потокового посимвольного, построчного, блокового и форматированного ввода-вывода.

ВВОД - ВЫВОД СИМВОЛОВ.

int fgetc(FILE * stream)/* Возвращает следующий символ из файла с преобразованием в int.*/

int getc(FILE *__fp); - читает символ из потока

int fgetchar(void);/* Возвращает следующий символ из stdin; это всё равно. Что fgetc (stdin).*/

int getchar(void);//-из stdin

int fgetchar(void);//-получает символ из stdin

int ungetc(int ch, FILE *fp) - возвращает символ назад в поток

int fputc(int c, FILE *stream) - вывод символа _ в файл

int putc(const int c, FILE *fp) - в поток

int fputchar(int c) -вывод символа в stdout

int putchar (const int c) - в stdout

int getch(void)-с клавиатуры без отображения(в conio.h)

int putch(int c)-в текущее окно

ВВОД - ВЫВОД СТРОК

char * fgets (char s, int n, FILE * stream) - Читает символы из файла. Пока не наберёт n-1 или встретит символ новой строки. В конец всегда запишется нулевой байт конца строки. Возвращает указатель на строку или NULL.

char * gets(char *s) - со стандартного ввода до ‘\n’

int fputs(const char *s, FILE *stream) - вывод до нуль-байта int puts(const char *s) - то же в stdout

БЛОКОВЫЙ ВВОД - ВЫВОД.

Чаще всего используется при переносе структурных переменных или массивов. Файл следует открывать в 2-ичном режиме, а в качестве размера блока указывать `sizeof`(var)

`int` fread(`void` *ptr,`int` size,`int` n,FILE *fp) - чтение n элементов размером size байт каждый в буфер ptr из файла fp При ошибке возвращается EOF

`int` getw(FILE *fp) - чтение целого из файла fp

`int` fwrite(`void` *ptr,`int` size,`int` n,FILE *fp) - запись блока

`int` putw(`int` w,FILE*fp) - запись целого

ФОРМАТИРОВАННЫЙ ВВОД-ВЫВОД.

Является частным случаем строкового. Основная особенность состоит в преобразовании байтов строки и присваивание полученных значений специфицированным переменным. Одним из аргументов подпрограмм форматированного ввода - вывода всегда является форматная строка, задающая вид преобразования. Другие параметры - это указатели на переменные, которым присваиваются вводимые значения или это имена выводимых переменных при выводе.

Строка формата состоит из отдельных полей:

% [флаги][ширина][.точность][F|N|h]<тип>

Флаги:

- ■ - выравнивание вывод числа влево
- ■ + вывод знака числа
- ■ Пробел - вывод пробела перед положит числом
- ■ # вывод идентификатора системы счисления

Ширина(только для вывода):

n - минимальная ширина с пробелами в незаполненных позициях

0n - но с ведущими нулями в незаполненных слева целых

• - следующий аргумент из списка задает ширину

Модификатор:

F - `far` - указатель

N - `near` - указатель

h - `short int` для целых

l - `long int` для целых

Типы преобразований:

c - символьный ввод - вывод

d или i - десятичное целое со знаком

• - восьмеричное без знака

u - десятичное `int` без знака

x,X - 16-ричные целые без знака
f,e.E,g,G - форматы для типа `float`
s - строковый ввод - вывод

Перечень функций форматированного ввода - вывода (их прототипы и соответствующие заголовочные файлы для подключения библиотек вы получите через help - систему интегрированной среды)

`int` fprintf(FILE *stream, `const char` *`for`mat,...)
`int` printf(`const char` *`for`mat,...)
`int` fscanf(FILE *stream,`const char` *`for`mat,...)
`int` scanf(`const char` *`for`mat,...)
`int` sprintf(`char` *buffer, `const char` *`for`mat,...);
`int` sscanf(`const char` *buffer, `const char` *`for`mat,...);
`int` vfprintf(FILE *stream, `const char` *`for`mat,`void` *arglist)
`int` vfscanf(FILE *stream, `const char` *`for`mat,`void` *arglist)
`int` vprintf(`const char` *`for`mat, `void` *arglist)
`int` vscanf(`const char` *`for`mat, `void` *arglist)
`int` vsprintf(`char` *buffer,`const char`*`for`mat,`void`*arglist)
`int` vsscanf(`const char`*buffer,`const char` *`for`mat,`void` *arglist)

ПРОИЗВОЛЬНЫЙ ДОСТУП К ФАЙЛАМ.

Место в файле, куда ведется запись или откуда осуществляется считывание, определяется значением указателя чтения - записи. Стандартная Си - библиотека содержит подпрограммы для установки этого указателя в нужное место, т.е. позволяет доступ к любому байту в файле.

`void` rewind(FILE *fp) - «перемотка» указателя в начало файла
`int` fseek(FILE *fp, `long` Offset, `int` from_where) - сдвигает указатель на Offset байт к началу или концу файла от позиции from_where, которая может быть SEEK_SET (0) - от начала, SEEK_CUR (1) - от текущей позиции, SEEK_END (2) - от конца.
`long` ftell(FILE *fp) - возвращает текущее значение указателя
`int` fgetpos(FILE *fp, fpos_t *pos) - заносит в pos значение указателя чтения - записи
`int` fsetpos(FILE*fp, `const` fpos_t *pos) - устанавливает указатель в позицию pos

УПРАВЛЕНИЕ БУФЕРИЗАЦИЕЙ.

Размер встроенного буфера Borland C++ 512 байт для регулярных файлов и 128 байт для стандартных. При выполнении подпрограмм установки указателя чтения - записи через rewind() и fseek() и при закрытии файла производится флэширование буфера - очистка для буфера ввода и перенос данных из буфера в файл при выводе. Программа может явно вызвать флэширование одного или всех буферов :

```
int fflush(FILE *fp)
int flushall(void)
```

Библиотека языка Си имеет подпрограммы для либо полного отключения буферизации, либо задания необходимого размера буфера, либо использования в качестве буфера предоставляемую программой область памяти:

void setbuf(FILE *fp, char *buf) - если buf == NULL то буферизация отключена, иначе буфер в 512 байт размещается по адресу buf.

int setvbuf(FILE *fp,char *buf,int type,unsigned size) - type задает способ буферизации: _IONBF - выключена, _IOFBF - под буфер выделяется size байт по адресу buf, а если buf == NULL, то место под буфер будет в куче; _IOLBF - при каждой записи ‘\n’ буфер флэшируется.

5.10.3. Префиксный доступ к файлам

Базируется на группе подпрограмм нижнего уровня, сразу обращающихся к MS DOS:

```
#include<io.h>
int open(const char *pathname, int access [,unsigned mode])
```

- открывает файл в режиме чтения- записи и возвращает префикс открытого файла или -1; через access можно задать права доступа, а через mode режим создания файла: access:

O_RDONLY, OWRONLY, O_RDWR,O_CREAT использует параметр mode:

S_IFMT - маска файла (в ДОС не поддерживается),

S_IFDIR - файл является каталогом,

S_IFIFO,S_IFCHR - спец символьный файл,

S_IFREG - регулярный

O_TRUNC, O_EXCL, O_APPEND, O_TEXT, O_BINARY, O_NOINHERIT - понятны по написанию.

```
int _open(const char * Filename, int Oflags)
```

Значения Oflags формируются по ИЛИ из констант:

FA_RDONLY, FA_HIDDEN, FA_SYSTEM, FA_LABEL, FA_DIREC, FA_ARCH

int _creat(const char *path, int attrib)

int creat(const char *path, int amode) - эти функции создают файл

int createnew(const char *path, int attrib)- отличается тем, что при наличии файла с указанным именем возвращает ошибку -1

int creattemp(char *path,int attrib) - генерирует уникальное имя и создает файл в указанном каталоге.

int close(int handle), int _Close(int handle) - функции закрытия файлов

int read(int handle,void *buf,unsigned len). int _read(int handle,void *buf,unsigned len) - читать len байт из файла handle в буфер buf

int write(int handle,void *buf,unsigned len)

int _write(int handle,void *buf,unsigned len) - запись

long lseek(int handle, long Offset, int fromwhere) - установка указателя префиксным доступом

5.10.4. Переадресация файлового ввода - вывода.

В составе MS DOS есть подпрограммы для дублирования открытых файлов - вызов функции AH=45H прерывания 21H приводит к появлению в таблице открытых файлов 2-х копий элементов и на один и тот же файл можно будет ссылаться по 2-м разным значениям индексов элементов (префиксов). При выполнении вызова AH=46H копирование элемента происходит в заданное место таблицы и если это место занято, т.е. заданный префикс уже выделен открытому файлу, этот файл автоматически закрывается. При дублировании изменяется только префикс файла, а все другие реквизиты (положение указателя, режим открытия) сохраняются.

В Турбо Си эти вызовы отображены 2-мя библиотечными функциями:

int dup(int handle) - дублирует префикс в свободное место таблицы открытых файлов; при достижении предела на число открытых файлов errno=EMFILE,при ошибке возвращается -1 и в errno=EBADF.

int dup2(int old_handle,int new_handle) - выполняет вызов AH=46H

Как правило, эти функции используются при переадресации обмена со стандартными файлами ввода - вывода

5.10.5. Создание и уничтожение файла - каталога.

В ОС есть подпрограммы для создания, удаления, просмотра содержимого каталогов (директориев) в пределах текущего накопителя и эти подпрограммы дублируются в библиотеке Си с прототипами в <dir.h>:

```
#include<dir.h>
int mkdir(const char *path) - для создания каталога
int rmdir(const char *path) - для удаления пустого каталога
```

5.10.6. Управление текущим накопителем и каталогом.

Во внутренних переменных ДОС хранятся номера текущего накопителя и текущего каталога для каждого накопителя. Библиотека Си содержит функции для определения и установки текущих каталога и накопителя:

```
int getdisk(void) - возвращает номер накопителя 0 - A, 1- B...
int setdisk(int drive) - устанавливает drive текущим диском
int getcurdir(int drive,char *directory) - возвращает текущий каталог
char *getcwd(char *buf,int buflen) - запишет в buf маршрут текущего рабочего каталога
int chdir(const char *path) - установит текущим каталог
```

5.10.7. Чтение содержимого каталога и поиск файлов.

```
int findfirst(const char *Filename,struct ffblk *ffblk,int attrib)
```

- вызывает AH=4EH для получения информации о первом файле, имя которого Filename и атрибут attrib; допускается использование шаблонов имен файлов. При совпадении заполняется структурная переменная по шаблону ffblk:

```
struct ffblk{
char ff_reserved[21];
char ff_attrib;
```

```
unsigned ff_time;
unsigned ff_date;
long ff_Size;
char ff_name;}
```

int findnext(struct ffblk *ffblk) - продолжает поиск совпадений, начатый findfirst().

void fnmerge(char *panh,const char *drive, const char *dir, const char *name, const char *ext) - в path строится строка спецификации файла по заданным компонентам

int fsplit(const char *path,char *drive, char *dir,char *name, char *ext) - разборка спецификации на элементы

char *searchpath(const char *File) - выполняет поиск файла по имени и возвращает указатель на его полную спецификацию или NULL

5.10.8. Переход от префиксной к потоковой форме доступа.

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

FILE *fdopen(int handle, char*type) - создает описание потока и возвращает указатель на него, в поле fd префикса запишется handle уже открытого файла, в type задается право доступа к файлу - оно не должно противоречить правам, заданным при префиксном открытии

5.10.9. Удаление и переименование файлов.

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

int unlink(const char *Filename) - для удаления файла

```
#include<dir.h>
```

int rename(const char* oldname, const char *newname) - для переименования файла

char *mktemp(char *template) - генерирует уникальное имя файла в шаблон template, содержащий символы, обязательно включаемые в имя

char *tmpnam(char *sptr) - генерирует имя, не совпадающее ни с одним в текущем каталоге, помещает его в буфер sptr длиной 13 байт

5.10.10. Определение существования файла или каталога.

Определить существование регулярного файла можно с помощью функций открытия, задав права только для чтения, а проверку наличия каталога - попыткой установить этот каталог текущим. Другой способ определения существования файла или каталога - использование функции findfirst. Но более предпочтительным является использование функций, не выполняющих открытия файла и позволяющих получить дополнительную информацию:

```
#include <io.h>
int access(const char *Filename, int amode) - проверяет
```

существование или права доступа, параметр amode:

00 - проверка существования файла или каталога
02 - допускается ли запись
04 - допускается ли чтение
06 - проверка на чтение - запись
При утвердительном ответе возвращается 0, иначе -1

5.10.11. Определение и установка параметров файла.

Полезная информация о файле может быть получена функциями fstat и stat - они заполняют поля структурной переменной по шаблону, описанному в заголовочном файле <sys\stat.h>:

```
struct stat{
short st_dev;   номер накопителя    short st_ino;
short st_mode; биты прав доступа
short st_nlink; число привязок к файлу в MS DOS =1
int st_uid;
int st_gid;
short st_rdev; то же что и st_dev
long st_Size;   размер открытого файла
long st_atime;
}

#include<sys\stat.h>
int fstat(int handle, struct stat *statbuf)
int stat(char *path,struct stat *statbuf)
```

Наиболее часто обращаются к этим функциям для определения размера файла или прав доступа. Другой способ

получения длины файла - позиционирование указателя на конец файла (можно сразу открыть его для дополнения a+) и получение его позиции.

При префиксном доступе можно изменить длину файла функцией:

int chsize(int t handle,long size) - усекает или расширяет файл

Для получения некоторых параметров состояния открытого потока можно использовать следующие функции:

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

int feof(FILE *stream) - возвращает не 0, если достигнут конец файла

int ferror(FILE *strem) - не 0 при ошибках

int clearerr(FILE *fp) - сброс индикаторов ошибки и конца файла

Для префиксных файлов:

```
#include<io.h>
```

int eof(int handle) - 1 если конец файла

Изменение прав доступа к файлу может быть выполнено функция-
ми:

```
#include<sys/stat.h>
```

int chmod(const char *Filename,int amode) - изменение прав доступа (атрибута)

int _chmod(const char *Filename,int func,[,int attrib]) - позволяет либо запросить (func=0), либо установить (func=1) права доступа

```
#include<io.h>
```

int setmode(int handle,int amode) - устанавливает права доступа, amode либо O_TEXT, либо O_BYNARY

ПОЛУЧЕНИЕ И УСТАНОВКА ДАТЫ И ВРЕМЕНИ.

```
#include<io.h>
```

int getftime(int handle,struct ftime *ftimep)

int setftime(int handle, struct ftime *ftimep)

5.11. Управление ресурсами из прикладных программ. Ввод с клавиатуры средствами библиотек языка Си.

5.11.1. Общие сведения.

Основным средством ввода для пользователя является клавиатура - она служит для ввода командной строки при запуске программ из ОС, обеспечивает подготовку текстов в текстовых редакторах, позволяет управлять ходом программы нажатием всевозможных комбинаций клавиш, осуществлять выборы в меню программных услуг и многое другое.

Клавиатура ПК содержит встроенный микропроцессор; он при каждом нажатии и отпускании клавиши определяет ее порядковый номер, называемый скэн-кодом, и помещает его в порт 60H микросхемы связи с периферией. 1-е 7 битов скэн-кода - это номер клавиши, а 8-й сигнализирует нажатие (0) или отпускание (1). При удержании клавиши в нажатом состоянии дольше установленного времени задержки микропроцессор клавиатуры начнет генерировать коды нажатия с заданной частотой. Задержка и частота могут устанавливаться в нужные значения либо через порты клавиатуры, либо использованием функции AH=03H прерывания 16H BIOS.

Стандартный обработчик аппаратного прерывания от клавиатуры имеет номер 9 - по его вектору помещается BIOS - программа обработки, анализирующая и обрабатывающая скэн-коды клавиш. В зависимости от типа клавиши применяется свой алгоритм обработки скэн-кода:

1. Шифт - клавиши Shift,Alt,Ctrl
2. Триггерные клавиши NumLock,ScrollLock,CapsLock
3. Клавиши с буферизацией расширенного кода 4. Спецклавиши

За каждой шифт- и триггерной клавишей закреплен свой бит в ячейках памяти по адресам 40:17H и 40:18H; при каждом нажатии и отпускании эти биты инвертируются и состояние бита рассказывает о том, нажата или отпущена клавиша. За триггерными клавишами закреплены 2 бита, 1 из них инвертируется при нажатии, а 2-й при нажатии и отпускании. Текущее состояние этих клавиш используется при выборе правил преобразования скэн-кодов от других клавиш.

При нажатии большинства клавиш и их комбинаций в специальный буфер помещается 2-байтовый BIOS-код, у которого младший байт равен ASCII-коду либо 0, а старший - скэн-коду или так называемому расширенному скэн-коду.

2- байтовый 0 + расширенный скэн-код записывается в буфер клавиатуры при нажатии функциональных клавиш, Ins, Del, клавиш управления курсором и их комбинаций с Alt, Ctrl, Shift и прочих комбинациях Alt с ASCII - клавишами.

Клавишная комбинация PrtScr вызывает прерывание 5, Alt+Ctrl+Del передает управление программе начальной

загрузки BIOS, а Ctrl+C записывает по адресу 00471H значение 80H, используемое как флаг желания пользователя остановить выполнение операций ввода - вывода.

5.11.2. Буфер клавиатуры.

Буфер BIOS для кодов клавиш занимает 32 байта с адреса 40:1EH до 40: 3EH, запись в буфер выполняет BIOS - обработчик 9-го прерывания, а чтение - функции BIOS прерывания 16H. Буфер рассчитан на 15 нажатий клавиш + 2 холостых байта для Enter.

Буфер организован как циклическая очередь, доступ к которой осуществляется с помощью указателя «головы» по адресу 40:1AH и указателя хвоста по адресу 40:1CH. Оба указателя содержат смещение от сегмента 40H.

При каждом нажатии по указателю головы записывается 2-байтовый код и указатель смещается на 2 байта, перепрыгивая при необходимости через холостую пару байтов и может «обогнать» содержимое указателя хвоста. При чтении из буфера смещается указатель хвоста. Если содержимое указателей равно, буфер пуст.

Буфер клавиатуры - классический пример использования кольцевого буфера для организации асинхронного взаимодействия 2-х программ по схеме «производитель - потребитель» - запись данных в буфер и считывание из буфера происходят в случайные, не связанные между собой моменты времени. При переполнении буфера производитель блокируется, пока потребитель не освободит чтением одно или более мест в буфере.

Прикладные программы, выполняющиеся под управлением MS DOS, могут использовать различные средства организации ввода с клавиатуры:

- Функции типа дескрипторов;
- Традиционные символьно-ориентированные функции ;
- Функции драйвера ПЗУ BIOS.
- Непосредственный доступ к буферу клавиатуры.

Перечисленные средства обеспечивают различную степень гибкости, переносимости и аппаратной независимости.

Для использования потокоориентированных функций типа дескрипторов программа должна предоставить дескриптор выбранного устройства ввода, адрес и длину приемного буфера памяти. Когда программа начинает выполняться, MS DOS предоставляет в ее распоряжение набор заранее определенных

дескрипторов символьно-ориентированных устройств, включая и клавиатуру:

Дескриптор	Устройство	Обозначение
0	Клавиатура stdin	CON
1	Дисплей stdout	CON
2	Дисплей stderr	CON
3	Последовательный порт stdaux	AUX
4	Принтер stdprn	PRN

Эти дескрипторы могут использоваться в операциях чтения записи без всякой предварительной подготовки. Кроме того, программа может получить дескриптор символьно-ориентированного устройства операцией его открытия как файла для ввода или вывода с использованием его логического имени. Дескрипторные функции поддерживают операцию перенаправления ввода - вывода, позволяя программе брать входные данные вместо например клавиатуры с другого устройства или с дискового файла. Дескрипторные функции ввода с клавиатуры обслуживаются прерыванием 21H с функцией 3FH.

Функции MS DOS для ввода с клавиатуры могут вызываться напрямую из Си - программ через библиотечные подпрограммы gen`interrupt()`, `int`86`()`, `intr()` и т. П. Либо неявно подпрограммами ввода С.

Большая группа функций С для ввода с клавиатуры - это функции потокового и префиксного ввода. Они используют функцию MS DOS AH=3FH с префиксом стандартного ввода 0. Следовательно, используя функции ввода из файла stdio.h, невозможно выполнить неотображаемый на экране ввод символов, обойти обработку Ctrl Break, определить нажатие спецклавиш. Для выполнения таких действий используются прототипы функций из файла conio.h (консольного ввода - вывода). Они рассчитаны на построение простейших оконных интерфейсов, поэтому при любом выводе на экран с их помощью корректируются значения переменных, хранящих текущие координаты курсора активного окна. Кроме того, предоставляется возможность управления цветом вывода. Итак:

`#include` <conio.h>

`char` *cgets(`char` *str) помещает в буфер str строку символов со стандартного ввода; запись символов начинается с str[2]; str[0] должен содержать максимальную длину строки - используется функция 0AH, но символ CR преобразуется в ‘\0’ с

порождением корректной ASCIIZ - строки. По сравнению с gets() функция позволяет задать длину вводимой строки, защититься при вводе от лишних символов, возможность ввода строк длиннее 128 символов, установленных для стандартного ввода.

int cscanf(char *format[,arg,...]) выполняет форматированный ввод с клавиатуры без буферизации ввода, все введенные символы доступны в программе немедленно. Ввод пробела рассматривается как завершение ввода, работает она через функцию AH=07H, а эхо выполняет библиотека C.

int getch(void) - вводит символ без эха.

int getche(void) - то же но с эхо, перевод строки происходит при достижении правой границы активного окна.

char * getpass(const char *prompt) - выводит строку prompt, а затем принимает с клавиатуры без эха строку символов не длиннее 7; введенные символы помещаются во внутреннюю память, указатель на введенную строку возвращается. Основное назначение - ввод паролей.

int kbhit(void) - проверяет буфер клавиатуры на пустоту, возвращая не 0, если там что-то есть.

int ungetch(int ch) - записывает символ ch в буфер клавиатуры, он будет доступен при следующей операции чтения с консоли функциями файла conio.h.

5.11.3. Ввод в (Турбо, Борланд) C (C++) средствами BIOS.

Работа функции BIOS AH=02H прерывания 16H моделируется библиотечной функцией:

#include<bios.h>

int bioskey(int cmd) в зависимости от cmd обращается к AH=00 или 02H прерывания 16H и возвращает 2-хбайтовый код клавиши, что бывает удобно использовать при смешанной работе с символьными и управляющими клавишами.

Очистка буфера клавиатуры может быть выполнена несколькими способами: приравниванием указателей головы и хвоста, использованием функции AH=0CH с операцией ввода в AL, циклическим чтением через bioskey(0) или getch().

Очистка приравниванием указателей:

```
#include <dos.h>
void clear_kb(void)
{char register _es *tail=0x1c, _es *head = 0x1a;
_es=0x40; disable(); *tail=*head;enable();
```

```
}
```

Ввод с предварительной очисткой:

```
#include<dos.h>
unsigned int getc_kb(void)
{register ret_AX=0;
_AX=0x0c01; geninterrupt(0x21);
ret_AX=_AL;
if(!ret_AX) {_AX=0x0c01;geninterrupt(0x21);
ret_AX=_AL;return ret_AX<<8; }
return ret_AX;
}
```

Чтение символов из буфера клавиатуры можно выполнить доступом к ячейке памяти, на которую ссылается указатель головы:

```
#include<dos.h>
unsigned key_kb(void)
{register char _es *tail =(char _es*)0x1c;
register char _es *head =(char _es*)0x1a;
_es=0x40;
if(*head == *tail) return 0;
disable(); tail=(char _es*)(*head);enable();
return *(unsigned _es*)tail;
}
```

5.12. Управление ресурсами компьютера из прикладных программ. Видеосистема.

5.12.1. Краткие сведения по архитектуре видеосистем. Основные типы дисплеев.

КОМПОЗИТНЫЙ ДИСПЛЕЙ. Имеет 1 аналоговую входную линию, может быть цветным или монохромным. Видеосигнал поступает в дисплей в стандарте NTSC, который используется и в телевидении.

Применяется совместно с видеоадаптером CGA.Сам дисплей может отображать до 16 цветов, но с CGA обеспечивает 4 цвета для графики и 8 цветов для текста. В монохромном варианте имеет разрешающую способность 720x350 и используется в основном для систем подготовки текстов с адаптером улучшенного типа Hercules. В цветном варианте имеет разрешение 640x200 и в настоящее время не используется.

ЦИФРОВОЙ ДИСПЛЕЙ. Имеет от 1 до 6 входных линий, может отображать до 2^n цветов (любые 16 из 64 при 6 линиях), может использоваться с адаптерами EGA, CGA. Разрешающая способность 640x350, размер символов 8x14 пикселов.

АНАЛОГОВЫЙ RGB-ДИСПЛЕЙ с 3-мя аналоговыми входными линиями для красного, зеленого и синего цветов, количество цветов ограничено только возможностями адаптера. Используется с VGA, Super VGA, XGA.

ВИДЕОАДАПТЕРЫ.

Это устройство на специальной электронной плате, с собственным микропроцессором, предназначено для управления выводом на дисплей. В самом общем виде видеоадаптер состоит из контроллера электронно-лучевой трубки (CRT-контроллера) и видеопамяти (видеобуфера), а в более совершенных системах могут иметь дополнительные узлы, например, контроллеры быстрой графики для манипуляций содержимым видеопамяти (акселераторы 2- и 3хмерной графики, Windows, MPEG-декодеры). Видеобуфер хранит образ экранной информации, а видеоадаптер 25-30 раз в секунду освежает (заново рисует) изображение на экране. Изображение на экране строится из небольших телевизионных точек прямоугольной формы - так называемых пикселов и разрешающая способность указывается числом пикселов в строке и числом строк.

Память, необходимая для хранения одного полного образа экрана, называется видеостраницей и при наличии достаточного объема видеопамяти можно хранить в видеобуфере несколько страниц - одна из них будет отображаемой или текущей.

ВИДЕОРЕЖИМЫ.

Интегральной характеристикой видеоадаптера является совокупность поддерживаемых им режимов. Режимы принято нумеровать, начиная с 0 - чем совершеннее адаптер, тем больше режимов он поддерживает - как правило, более совершенные адаптеры поддерживают режимы своих менее совершенных предков.

При всем разнообразии режимов их можно объединить в 2 группы: текстовые и графические - переключение из одного режима в другой полностью изменяет логику взаимодействия адаптера с видеобуфером.

Всего насчитывается около 20 стандартных и 200 нестандартных режимов и с появлением новых типов

видеосистем этот список расширяется. Мы рассмотрим только основные:

Стандартные:

0,1-текстовые цветные 40x25, 8 основных и 8 интенсивных цветов для CGA, 16 из 64 для EGA, 16 из 262144 для VGA, видеостраница 8 Кбайт, EGA и VGA -до 8 страниц

2,3-текстовые цветные 80x25,16 цветов, видеостраница 16 Кбайт

7-текстовый монохром 80x25,2 цвета

4,5-цветная графика 320x200,8 цветов (по 4),1 страница, 2 бита на пиксел

6 -2-цветная графика 640x200,2 цвета 1 страница

0Dh - цветная графика, 320x200.16 цветов, страница 32 Кбайт

0Eh - цветная графика 640x200.16 цветов, страница 64 Кбайт

0Fh - монохромная графика 640x350 2 страницы по 64 Кбайт

10h - цветная графика 640x350 16 цветов (при 64 кб-4цвета)

11h - цветная графика 640x480 2 цвета

12h - цветная графика 640x480 16 цветов, 1 видеостраница (только VGA)

13h - цветная графика 640x200 256 цветов

В текстовых режимах экран рассматривается как совокупность «текселов» - прямоугольных областей для размещения символов. Каждому текселу соответствуют 2 байта в видеобуфере - четный хранит ASCII - код символа, а следующий за ним нечетный байт кодирует цвет пикселов для очертания символа, цвет фона, необходимость включения режима мерцания и повышенной яркости - этот байт называют байтом атрибута. При побитной расшифровке байт атрибута выглядит следующим образом:

Биты 0,1,2 - Foreground Color - код цвета символа
3 - Яркость символа
4,5,6 - Цвет фона символа
7 - Режим мерцания

Присваивая различные значения байтам атрибута в видеобуфере, можно управлять цветом символов и фона, на котором изображается их рисунок. Например, если байт атрибута равен

$$128 + 64 + 32 + 16 = 240,$$

то выведется мерцающий черный символ на сером фоне (смешивание в одинаковой пропорции синего, зеленого и красного на RGB-мониторах дает серый цвет). Включение бита интенсивности «осветляет» цвет, светло-серый - это белый. При атрибуте

$$8 + 4 + 2 + 1 = 15$$

будут белые символы на черном фоне. Если задать одинаковые цвета фона и символов, то символы будут невидимы, например, на красном фоне с атрибутом 44H.

Видеопамять в текстовых режимах доступна для непосредственного доступа из прикладных программ, как и обычная оперативная память - из нее можно читать и в нее можно записывать. Если адаптер работает в текстовых режимах «40 столбцов x 25 строк», то для хранения полного образа экрана (видеостраницы) необходимо 25x40x2 = 2000 байт памяти, а в режимах 80x25 уже 4000 байт. Адреса начала видеобуферов для различных типов адаптеров:

CGA,EGA,VGA	B8000H для режимов 0..6
Hercules	B0000H для режима 7
EGA,VGA	A0000H для режимов DH..13H

«Освежение» видеобуфера происходит, начиная с некоторого начального адреса - смещения до видеостраницы. Страница 0 имеет 0-е смещение, страница 1 в режиме 80x25 начинается со смещения 4096 (1000H) относительно начального адреса видеопамяти и т д. Если изменить смещение для операции «освежения» видеобуфера, произойдет переключение страницы и на экране возникнет образ другой страницы видеопамяти, а образы других страниц останутся в памяти невредимыми и пригодными для последующего отображения.

Байт кода символа используется видеоадаптером как индекс для входа в специальную таблицу знакогенератора - она задает 2-ичные коды строк рисунков символов ; число строк для рисунка символа зависит от разрешающей способности (количества телевизионных линий умещающихся на экране) и может быть 8, 14, 16, а число пикселов в строке символьного изображения обычно 8 или 9.

При работе в **ГРАФИЧЕСКОМ** режиме обеспечивается возможность управлять цветом любого пиксела на экране - экран в графическом режиме представляет собой матрицу пикселов. Коды цветов пикселов хранятся в видеобуфере, при этом в режимах 4..6 (CGA - режимы) возможен непосредственный доступ к видеопамяти из программы столь же простой, как и в текстовых режимах - записывая по нужным

адресам коды цветов пикселов. Для кодирования цветов в 4 - цветных режимах 4 и 5 используется 2 бита на пиксел и водном байте можно записать цвета 4-х пикселов. В 2 - цветном режиме 6 для кодирования цвета пиксела достаточно одного бита. В режимах 4..6 коды четных и нечетных строк в видеобуфере сгруппированы в 2 отдельных массива - по смещению 0 от начала страницы располагаются коды пикселов четных строк, а со смещения 2000H - нечетных.

При работе в графических EGA - режимах 0Dh..10h логика использования видеопамяти существенно меняется. Для каждого пиксела в видеобуфере отводится 4 бита и количество одновременно отображаемых цветов равно 16. Одному и тому же адресу видеопамяти соответствуют сразу 4 байта, расположенных на так называемых битовых планах и кодирующих в совокупности 8 соседних пикселов телевизионной строки экрана. EGA - адаптер имеет 4 битовых плана (I, R, G, B) и любому пикселу соответствует по 1 биту на каждом плане. Код цвета в EGA - это номер одного из 16-ти специальных внутренних 6-тиразрядных регистров палитры, в которых и находится код отображаемого на экране цвета. Т.о., EGA позволяет выбрать 16 цветов из 64 возможных. Возможные 64 цвета задаются форматом

r g b R G B

где строчные буквы задают половинную интенсивность, а прописные - нормальную соответствующего цвета. Результирующий цветовой сигнал образуется сложением сигналов цвета, управляемых каждым из 6 бит кодов цвета.

VGA использует подобную схему преобразования кода цвета пиксела в реальный цвет на экране, но значение в регистре палитры задает не код цвета, а номер DAC - регистра (регистра цифро-аналогового преобразователя), т. К. На вход монитора необходимо послать аналоговый сигнал (напряжение), соответствующий яркости основных цветов. ЦАП имеет 256 3- байтовых регистров (по байту для красного, зеленого, синего), при этом 6 битов байта кодируют интенсивность цвета (64 оттенка яркости) и число возможных цветов получается 64x64x64=262144. Значения в DAC - регистры можно записать с помощью подпрограмм BIOS или непосредственным доступом к портам адаптера. При работе VGA в режимах EGA используются только первые 64 регистра, задающие 64 возможных цвета палитры EGA. Память VGA в режимах 0Dh..10h тоже делится на битовые планы и код цвета пиксела задает номер регистра палитры, в качестве регистров палитры выступают 4 блока DAC - регистров по 64 регистра в

блоке. В текущий момент времени может быть активным только 1 блок.

В режиме 13H на экране могут отображаться сразу 256 различных цветов и для кодирования цвета каждого пиксела нужен целый байт, так как разрешение в этом режиме 320x200, то всего необходимо 64000 байт. В этом режиме коды всех пикселов располагаются в видеопамяти подряд, начиная с адреса A000:0000H, без деления на четные и нечетные строки. Код цвета задает номер одного из 256 DAC - регистров. Изменением значений в DAC - регистрах можно менять цвет пиксела на экране для заданного кода.

В текстовых режимах видеоадаптер отображает курсор, а BIOS имеет средства отслеживания и установки текущей позиции курсора. В графических режимах курсор не отображается, хотя вывод через BIOS в текущую позицию невидимого курсора по прежнему возможен, как и задание курсорной позиции.

5.12.2. Управление видеосистемой в Borland C++. Функции консольного ввода - вывода.

Их прототипы размещены в файле <conio.h>, они используют понятие активного окна. Установку параметров активного окна выполняет функция

```
#include<conio.h>
void window(int ltcol, int ltrow, int rbcol, int rbrow)
```

В качестве параметров - координаты левого верхнего и правого нижнего углов окна, строки и столбцы нумеруются от 1. Описание активного окна (фрейм) хранится во внутренней структурной переменной:

```
struct text_info{
        unsigned char winleft,wintop,
        winright,winbottom,
        attribute,normattr,
        currmode,
        screenhight,/*высота экрана */
        screenwidth,
        curx,cury};
```

Информация об активном окне доступна через функцию

```
void gettextinfo(struct text_info * r)
```

Функции textcolor(), textbackground(), textattr() и другие управляют цветом отображаемых символов окна.

УПРАВЛЕНИЕ КУРСОРОМ.

Включение и выключение курсора и изменение формы..

Функция AH=01H задает толщину курсора, при этом регистр CH определяет номер верхней линии, CL - нижней. Биты 5 и 6 кода CH управляют мерцанием и могут выключить курсор.

Текущая форма хранится BIOS по адресу 0040:0060H с номером нижней строки в младшем байте и верхней в старшем.

Пример функции, выключающей курсор или восстанавливающей его форму в зависимости от параметра mode:

```
#include<<dos.h>
#include<screen.h>
void cursor(int mode)
{static unsigned shape;
if(mode==ON) {if(shape) {_CX=shape; shape =0;}
else return;}
else
{
 shape=*(unsigned far *)MK_FP(0x40,0x60);
_CX=0x2000; /*Бит 6=0,бит 5=1*/
}
_AH=1; geninterrupt(0x10);
}
```

Текущая позиция курсора может быть установлена функцией консольного ввода - вывода:

```
#include<conio.h>
void gotoxy(int x,int y)
```

а текущие координаты курсора могут быть получены с помощью:

```
int wherex(void)
int wherey(void)
```

Кроме того, позиция курсора в окне возвращается в полях curx, cury структурной переменной text_info после вызова gettextinfo().

ВЫВОД ТЕКСТА НЕПОСРЕДСТВЕННО В ВИДЕОБУФЕР.

Если адаптер работает в одном из текстовых режимов или в графических CGA- режимах или 13-м EGA-VGA. То возможен непосредственный доступ к ячейкам видеопамяти. Это наиболее быстрый способ вывода, но он не сопровождается

автоматическим смещением курсора. Для доступа к видеобуферу используется far-указатель :

```
char far *vid_buf=(char far*)0xb80000000;
...
vid_buf+=5*160+40*2; /*5-я строка 40-й столбец */
*vid_buf++='A';
vid_buf=atr;
```

Для повышения скорости можно осуществлять запись не побайтно, а сразу словом:

```
unsigned word;
...
word=(unsigned)((symbol<<8)|attribut);
...
```

Еще большую скорость можно получить, записав перед доступом к видеопамяти ее адрес в регистр ES и используя специальный тип near - указателя unsigned _es * work.

И самая высокая скорость переноса достигается при использовании библиотечных функций с прототипами в <mem.h>.

При записи в видеобуфер для вычисления смещения необходимо знать максимальное число столбцов текста в текущем режиме, номер текущей видеостраницы и сегмент адреса начала видеопамяти - эти переменные разумно объявить как внешние и инициализировать при первой записи в видеопамять и при каждой смене режима адаптера.

5.12.3. Вывод в окна средствами Borland C++.

Прототипы функций - в файле <conio.h>. В отличие от функций стандартного ввода - вывода они позволяют управлять цветом и не пересекают границ активного окна - при достижении правой границы курсор переходит на следующую строку, а при достижении нижней границы осуществляется скроллинг вверх.

void clreol(void) - стирает строку от позиции курсора;
void clrscr(void) - очистка текстового окна;
void delline(void) - стирает строку в которой курсор
void insline(void) - вставка пустой строки со сдвигом остальных вниз и потерей самой нижней
int cprintf(const char * format,...) - форматированный вывод в окно, возвращает число выведенных байтов

int cputs(const char * str) - вывод строки в окно с текущей позиции курсора, возвращает код последнего выведенного символа
int movetext(int left,int top, int right, int bottom, int destleft, int desttop) - перенос окна в другое место
int putch(int ch) -вывод символа
int puttext(int left,int top, int right, int bottom,void*source) - вывод окна с содержимым из буфера source, который обычно заполняется функцией gettext().
void highvideo(void), void lowvideo(void), void normvideo(void) - задание яркости вывода
void textattr(int newattr) - устанавливает атрибут; параметр может задаваться либо числом, либо формироваться из символических констант: BLINK | (BLACK<<4) |LIGHTRED
void textcolor(int newcolor) - установка цвета символов
void textbackground(int newcolor) - установка цвета фона символов

5.12.4. Чтение информации с экрана.

Потребность в этом возникает при организации оконного интерфейса, при построении резидентных HELP-систем, выдающих подсказку по слову в позиции курсора.

Для получения высокой производительности надо читать прямо из видеобуфера; классическая задача - сохранение окна экрана, для выполнения которой в библиотеке C есть функция gettext():

int gettext(int left,int top, int right, int bottom,void* destin)

5.12.4. Особенности вывода текстов в графических режимах.

Для вывода пиксельных представлений символов в графических режимах можно использовать функцию AH=9H прерывания 10H. При выводе этой функцией во всех графических режимах меньше 13 учитывается значение бита 7 кода цвета в регистре BL. Если он 0, то символы имеют цвет, заданный битами 0..6; если он 1, цвет пикселов определяется исключающим ИЛИ битов 0..6 с кодом текущего цвета на экране без изменения кода цвета фона 0. Это гарантирует видимость символов с отличными от 00 кодами цвета и для стирания символа с экрана достаточно вывести символ в той же позиции с тем же цветом.

В частности для вывода и восстановления текстового окна в графическом режиме средствами С поступают так:

- содержимое выводимого окна заранее размещают в отдельном буфере, отводя на каждое знакоместо 2 байта - для символа и атрибута; все атрибуты должны иметь взведенный старший бит, а 2 первых слова должны задавать число столбцов и строк окна;
- переменную directvideo обнуляют;
- окно «открывают» функцией puttext();
- восстановление части экрана, измененной выведенным окном, выполняют повторным выводом на том же месте функцией puttext().

5.12.5. Управление знакогенератором EGA и VGA.

В этих адаптерах аппаратно реализован стандартизованный способ установки таблицы знакогенераторов, предусмотрена возможность изменять битовые карты для любых символов.

Управление знакогенераторами EGA-VGA выполняет функция 11H прерывания 10H, имеющая ряд подфункций, выбираемых значением регистра AL.

В текстовых режимах эти адаптеры позволяют использовать несколько полных таблиц знакогенератора - блоков. На каждые 64 Кбайт видеопамяти имеется по 1 блоку. В текстовом режиме можно сделать активными сразу 2 блока с выбранными номерами, в результате на экране можно будет отображать до 512 различных очертаний символов. Выбор таблицы при этом будет выполнять бит 3 байта атрибута символа. В графических режимах можно использовать только 1 таблицу для 256 символов.

Таблицы знакогенераторов для графических режимов располагаются либо в ОЗУ либо в ПЗУ на плате адаптера, а в текстовых режимах - на плане 2 видеопамяти. Каждый раз при переключении в текстовый режим в память должна загружаться таблица знакогенератора. Очертание символа задается битовой картой 8x8, 8x14 или 8x16 в зависимости от режима работы адаптера. Битовые карты ложатся последовательно друг за другом начиная с карты для символа с ASCII - кодом 0 и до 255.

Таблицы знакогенераторов могут быть переопределены полностью или частично, но все изменения в таблице будут потеряны при любом переключении из режима в режим. Поэтому после каждого переключения режима необходимо вновь загружать определенную пользователем таблицу или активизировать 2 таблицы. По такому принципу работают

программы русификации экрана - после запуска они загружают пользовательскую таблицу, подменяют вектор 10H и завершаются резидентно. При каждом обращении к прерыванию 10H управление получает резидентный русификатор, он вызывает старый обработчик прерывания 10H и если вызывалась функция переключения режима - выполняет повторную загрузку пользовательской таблицы знакогенератора.

При выводе текстов в графических режимах 0Dh - 13h вектор 43H указывает адрес начала используемой таблицы. Поэтому, подменив этот вектор на адрес собственной таблицы, можно выводить «заказные» символы.

5.12.6. Вывод графической информации.

Непосредственный доступ к аппаратуре видеоадаптера обеспечивает максимальную скорость. Наиболее просто это решается для режимов 4,5,6,13H - видеопамять для 4,5,6 начинается с B800:0000H, для 13H - с A000:0000 и организована линейно. В 4,5 режимах каждый байт кодирует 4 пиксела, в 6-ом - 8 пикселов, в 13H - один пиксел. Видеобуфер в 4-6 режимах поделен на 2 части - для пикселов четных строк с адреса B800:000H и нечетных с B800:2000H, в 13H деления на чет-нечет нет.

БИБЛИОТЕКА ГРАФИКИ ТУРБО С.

Использование графической библиотеки Турбо С - многошаговый процесс, включающий:

-определение типа видеоадаптера;

-установка режима работы видеоадаптера;

-инициализация графической системы в заданном режиме;

Только после этого можно воспользоваться библиотечными подпрограммами для построения основных графических примитивов и вывода текста с использованием различных шрифтов.

В состав библиотеки входит набор так называемых BGI - драйверов, каждый из которых является обработчиком прерывания 10H и выполняет установку и обновление ряда внешних переменных, которые могут изменяться как подпрограммами операционной системы, так и подпрограммами библиотеки графики Турбо С.

ИНИЦИАЛИЗАЦИЯ И ЗАКРЫТИЕ СИСТЕМЫ ГРАФИКИ ТУРБО С.

Состоит в загрузке BGI - драйвера, соответствующего адаптеру или режиму, установке начальных значений внешних

переменных, выборе шрифта и т. Д. Поддерживаемые библиотекой графические режимы задаются символьными константами, описанными в файле <graphics.h> в перечислимом типе gtaphics_modes. Некоторые из этих констант:

Константа	Разрешение	Палитра	Режим
CGAC0	320x200	0	4,5
CGAC1		1	
CGAC2		2	
CGAC3		3	
CGAHI	640X200		6
EGALO	640X200	16цветов	0EH
EGAHI	640X350	16цветов	10H
VGALO	640X200	16цветов	0EH
VGAMED	640X350	16цветов	10H
VGAHI	640X480	16цветов	12H

Инициализацию графики выполняет функция

`void far` initgraph(`int far` *graphdriver,`int far`* graphmode,`char far` *pathtodriver) - инициализирует графическую систему, загружая BGI - драйвер graphdriver, устанавливает режим graphmode, используя путь доступа к драйверу, заданный строкой pathtodriver.

BGI - драйвер может тоже задаваться символьной константой из набора (приведены некоторые):

Константа	Значение	Описание
DETECT	0	Автоматическое определение типа драйвера
CGA	1	
EGA	3	
VGA	9	

Если нашей программе больше не нужна графическая библиотека, следует обратиться к функции

`void far` closegraph(`void`) - освобождает память под драйверами графики, шрифтами и пр., восстанавливает предыдущий режим адаптера.

Скелет программы, выполняющей всю необходимую подготовку для использования графической библиотеки:

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

```
int graph_driver,graph_mode,graph_error_code;
graph_driver=DETECT;
initgraph(&graph_driver,&graph_mode,»c:\\bc\\bgi»);
graph_error_code=graphresult();
if(graph_error_code!=grOk){/*Обработка ошибки*/
return 255;}
...
closegraph();
}
```

Наиболее защищенный способ использования функции инициализации требует предварительного уточнения типа адаптера - для этого либо вызывается initgraph() с драйвером, определенным через DETECT, либо явно вызывается функция:

`void far` detectgraph(`int far` *graphdriver,`int far` * graphmode)

Обработка ошибок

После любого обращения к detectgraph() или initgraph() желательно вызвать функцию

`int far` graphresult(`void`), возвращающую код ошибки, а его текстовую расшифровку можно получить с помощью функции

`char *far` grapherrormsg(`int` errorcode)

ОПРЕДЕЛЕНИЕ И УСТАНОВКА ГРАФИЧЕСКОГО РЕЖИМА.

После инициализации графической системы можно установить другой, не превышающий максимального, режим видеоадаптера и выбрать цвета для пикселов.

Установку режима выполняет функция setgraphmode().

Целая группа функций - getgraphmode(), getmaxmode(), getmodename(), getmoderange() - упрощает работу по определению текущего установленного видеорежима;

две функции - getmaxx(), getmaxy() - позволяют определить высоту и ширину экрана в пикселах для текущего режима;

функция restorecrtmode() возвращает адаптер в текстовый режим.

Синтаксис и параметры этих функций всегда можно уточнить через help - систему IDE Borland C++.

УПРАВЛЕНИЕ ЦВЕТАМИ И ПАЛИТРАМИ.

CGA - адаптеры в режимах 4,5 320x200 аппаратно поддерживают задание цвета фона и выбор одной из 4-х предопределенных палитр. Код цвета пиксела 0 соответствует

цвету фона и может быть выбран равным любому из 16 возможных цветов:

Константа	Значение	Цвет
BLACK	0	ЧЕРНЫЙ
BLUE	1	СИНИЙ
GREEN	2	ЗЕЛЕНЫЙ
CYAN	3	СИНЕ-ЗЕЛЕНЫЙ
RED	4	КРАСНЫЙ
MAGENTA	5	КРАСНО-СИНИЙ
BROWN	6	КОРИЧНЕВЫЙ
LIGHTGRAY	7	СВЕТЛО СЕРЫЙ
DARKGRAY	8	ТЕМНО-СЕРЫЙ
LIGHTBLUE	9	ЯРКО-СИНИЙ
LIGHTGREEN	10	ЯРКО-ЗЕЛЕНЫЙ
LIGHTCYAN	11	ЯРКИЙ СИНЕ-ЗЕЛЕНЫЙ
LIGHTRED	12	ЯРКО-КРАСНЫЙ
LIGHTMAGENTA	13	ЯРКИЙ КРАСНО-СИНИЙ
YELLOW	14	ЖЕЛТЫЙ
WHITE	15	БЕЛЫЙ

Установку кода цвета фона выполняет функция setbkcolor(), а функция getbkcolor() позволяет узнать код цвета фона в текущий момент. Выбор же палитры осуществляется по символьной константе, задающей режим работы.

У EGA, VGA - адаптеров возможности управления палитрами значительно шире - библиотека графики Турбо С тоже содержит группу функций для определения текущей и установки новой палитры EGA,VGA - адаптеров: getdefaultpalette(), gemaxcolor(), getpalette(), getpalettesize(), setallpalette(), setpalette().

УПРАВЛЕНИЕ ВИДЕОСТРАНИЦЕЙ.

Как уже отмечалось, многие адаптеры позволяют хранить в видеопамяти несколько страниц. Отображаемая в текущий момент называется видимой, а та, на которую выполняется вывод - активной. Первоначально активная и видимая страница имеют номер 0, но могут и не совпадать, в этом случае картинка формируется «за кулисами». Для управления страницами в С есть 2 функции:

```
void far setactivepage(int page)
void far setvisualpage(int page)
```

ЗАДАНИЕ ЭКРАННОГО ОКНА (VIEWPORT), ОПРЕДЕЛЕНИЕ И УСТАНОВКА ГРАФИЧЕСКИХ КООРДИНАТ.

Графическое окно может иметь отличающиеся от других участков экрана цвета фона и пикселов, маску заполнения и другие характеристики. Для описания окна используется функция

```
void far setviewport(int left,int top,int right,int bottom,int clip)
```

Получить текущие характеристики окна можно через обращение к

```
void far getviewsettings(struct viewporttype far *viewport),
```

которая заполнит поля структурной переменной.

В графическом окне определены относительные координаты, измеряемые в пикселах относительно левого верхнего угла окна. Текущие координаты доступны через функции getx(),gety(), а установить нужные координаты можно функциями moveto() и moverel().

В Borland C++ есть возможность встроить в .EXE - файл программы необходимые BGI - драйверы. Для этого с помощью утилиты BGIOBJ.EXE (при большом количестве драйверов - с ключом /F) файл драйвера преобразуется в .OBJ - файл, который компонуется с программой после регистрации с помощью функций:

```
int registerbgidriver(void(*driver)(void))
int far registerfarbgidriver(void far *driver)
```

Пример: `registerfarbgidriver(EGAVGA_driver)`

ВЫВОД ТЕКСТА.

Во-первых, вывод текста можно осуществить всеми функциями стандартного вывода - при этом будет задействована функция AH=9 прерывания 10H со всеми вытекающими последствиями, которые обсуждались ранее.

Библиотека графики позволяет выводить текст на экран различными шрифтами двух типов - битовым и сегментированным (или векторным). Для битового шрифта используется таблица знакогенератора 8x8. Сегментированные шрифты задают правило рисования каждого символа, описываемого как совокупность отрезков прямых линий или сегментов. Программа может задать масштаб для каждого символа.

В Турбо С доступны 4 сегментированных шрифта: Triplex, Small, SansSerif, Gothic. Файлы сегментированных шрифтов с расширением .CHR, как и .BGI - драйверы, загружаются как

оверлеи во время исполнения программы. Выводить в графике текст можно слева направо и снизу вверх, масштаб задается по отношению к знакоместу шрифта 8x8, а для сегментированных шрифтов - и в относительных единицах масштаба по вертикали и по горизонтали. Кроме того, выводимые строки могут по разному выравниваться - влево и вверх, влево и вниз и т. П. Относительно текущей точки отсчета. Поведение графической системы при выводе текста характеризуется набором внутренних переменных, значения которых доступны через функцию :
`void` gettextsettings(`struct` textsettingstype `far` *texttypeinfo), заполняющую поля структурной переменной по шаблону textsettingstype

Интерпретация отдельных полей этой структуры приводится в описании функции:

`void` settextstyle(`int` font, `int` direction, `int char`size)

Доступные шрифты задаются символьными константами:

Константа	Значение	Файл шрифта
DEFAULT_FONT	0	-
TRIPLEX_FONT	1	TRIP.CHR
SMALL_FONT	2	SMAL.CHR
SANS_SERIF_FONT	3	SANS.CHR
GOTHIC_FONT	4	GOTH.CHR

Направление может задаваться HORIZ_DIR, VERT_DIR. Функция:
void far setusercharsize(int multx, int divx, int multy,int divy)
позволяет установить масштабы по горизонтали и вертикали раздельно значениями числителя и знаменателя.

Установленные размеры символов можно узнать через функции textheight`()`,textwidth`()`.

Способ выравнивания задается функцией settextjust`ify`(`int` horiz,`int` vert), horiz может быть LEFT_TEXT, CENTER_TEXT, RIGHT_TEXT, vert может быть BOTTOM_TEXT, CENTER_TEXT, TOP_TEXT.

Собственно вывод осуществляется функциями:
`void far` outtext(`char far` *textstring)
`void far` outtextxy(`int` x, `int` y, `char far` *textstring)

Включение .CHR-файлов в выполняемый файл выполняют аналогично драйверам после регистрации функциями

registerbgifont(),
registerfarbgifont().

Имена шрифтов при регистрации
triplex_font, triplex_font_far
small_font. small_font_far
sansserif_font, sansserif_font_far
gothic_font.gothic_font_far

5.12.7. Вывод графической информации. Параметры и атрибуты графического вывода.

После успешной инициализации графической системы становятся доступными функции графической библиотеки для построения основных графических примитивов: отрезков прямых, дуг, окружностей, эллипсов, прямоугольников, секторных и столбцовых диаграмм и т.д. Многие из замкнутых фигур могут быть залиты текущим цветом с использованием текущего шаблона или маски заполнения. Есть также функции для запоминания прямоугольных областей экрана и восстановления их в заданном месте. Все функции работают в пределах текущего графического окна с использованием текущего цвета пиксела, установленного функцией setcolor().

При выводе отрезков прямых можно определить такой параметр как стиль линий:
void far setlinestyle(int linestyle, unsigned upattern, int thickness)

Аргумент upattern используется только если
linestyle = = USERBIT_LINE - тогда 16 бит upattern задают маску линии.

Линия с определенным пользователем стилем может иметь толщину только в 1 пиксел.

Поддерживается ряд предопределенных стилей и можно описать собственный. Задается стиль целым числом или символьной константой:

Константа	Значение	Стиль
SOLID_LINE	0	Сплошная
DOTTED_LINE	1	Точечная
CENTER_LINE	2	Штрих-пунктир
DASHED_LINE	3	Штриховая
USERBIT_LINE	4	Определяется пользователем

Толщина может быть 1 или 3 пиксела.

При выводе отрезков прямых можно дополнительно задать режим вывода линий:

void far setwritemode(int mode), при этом mode может быть либо COPY_PUT (0) либо XOR_PUT (1).

Следующие параметры - это маска и стиль заполнения. Маска определяется 8-мибайтовым шаблоном и рассматривается как битовая карта 8x8, вся заполняемая область разбивается на блоки по 8x8 пикселов. Наложение маски осуществляется так: если бит в маске 1, то пиксел имеет код текущего цвета, иначе не меняется. Маска и цвет пикселов задается функцией:

void far setfillpattern(char far * upattern, int color)

По умолчанию - белый цвет и маска с единицами во всех битах. Для удобства в библиотеке предусмотрен ряд предопределенных масок в комбинации с цветами, называемую стилем заполнения и устанавливаемую функцией:

void far setfillstyle(int pattern, int color)

Константа	Значение	Описание
EMPTY_FILL	0	Цветом фона
SOLID_FILL	1	Текущим цветом
LINE_FILL	2	Символами –
LTSLASH_FILL	3	// нормальной толщины
SLASH_FILL	4	//удвоенной толщины
BKSLASH_FILL	5	\\
LTBKSLASH_FILL	6	\\
HATCH_FILL	7	вертикально-горизонтальная штриховка
XHATCH_FILL	8	редкая штриховка крест-накрест
INTERLEAVE_FILL	9	густая штриховка крест-накрест
WIDE_DOT_FILL	10	редкими точками
CLOSE_DOT_FILL	11	частыми точками
USER_FILL	12	определенная пользователем маска

Для коррекции неквадратности пиксела на некоторых мониторах можно воспользоваться функцией установки коэффициента сжатия:

void far setaspectratio(int far xasp, int far yasp)

ЧТЕНИЕ - ЗАПИСЬ ОТДЕЛЬНЫХ ПИКСЕЛОВ.

Для манипуляции пик селами есть 2 функции:

unsigned far getpixel(int x,int y)
voi far putpixel(int x, int y, int pixelcolor)

ВЫВОД ОТРЕЗКОВ ПРЯМЫХ.

line(int x1,int y1,int x2,int y2)-линия между 2-мя точками

linerel(int dx, int dy);-линия из текущей позиции

lineto(int x, int y);-линия из текущей позиции в заданную точку.

6. Практическое процедурное программирование на Си.

6.1. Извлечение аргументов из командной строки.

Обычно первой программой выбирают вывод на экран строки "Hello World! - Привет Мир!". Вы можете это проделать - программа выглядит так:

```
#include<stdio.h>
main() {printf(\n"Hello World! - Привет Мир!");}
```

В файле stdio.h находятся прототипы ряда библиотечных функций ввода - вывода и мы включаем этот файл в текст нашей программы с помощью директивы #include препроцессору - программе предкомпиляционной обработки текста.

В теле главной функции main(), ограниченном парой фигурных скобок { }, вызывается на выполнение библиотечная функция форматированного вывода на экран - в нашем случае мы не используем ее способность преобразовывать переменные различных типов в строковую форму, а даем в качестве аргументов строку - константу (литерал).

Решим попутно еще одну столь же простую задачу - выведем на экран один столбец 16-ричной таблицы умножения, например, на число b:

```
//вывести 1 столбец таблицы умножения в 16-ричной системе на 0xb
#include<conio.h>
void main()
{
 clrscr();
 for(int i=0x1;i<0x11;i++)
  cprintf("b x %x = %x\n\r",i,0xb*i);
}
```

Здесь мы уже используем для отображения числа в 16-ричной системе счисления формат преобразования в строку %x - эти форматы приведены в одном из последующих разделов

Но основной нашей задачей в начале обучения программированию будет научиться извлекать из командной строки DOS, сформированной при вызове нашей программы на выполнение, имеющиеся в ней слова-аргументы. Дело в том, что подавляющее большинство программ всегда знают "что

делать", но, начиная работу, хотят узнать, с какими объектами надо это делать. Только программа, реализующая некоторый алгоритм для целого класса однотипных объектов обработки, имеет какую-то потребительскую ценность. Поэтому большинство программ требуют после начала выполнения уточняющих задачу аргументов и эти аргументы удобнее всего задать сразу после имени программы в командной строке.

Как мы уже сообщали, Си предоставляет 2 таких возможности - либо оформлением главной функции main() с параметрами, в которые специально прикомпонованным к любой Си-программе загрузчиком будут помещены количество слов в командной строке, массив указателей на эти слова и массив указателей на строки среды окружения, либо глобальные переменные из модуля dos.h _argc, _argv, environ, содержащие ту же информацию.

Итак, первая программа, в которой мы учимся получать слова из командной строки. Для вывода мы воспользуемся библиотечной функцией консольного форматированного вывода cprintf() с прототипом в conio.h, там же находится прототип подпрограммы очистки экрана clrscr(). Наберите эту программу и испытайте ее, четко уясните себе понятия "командная строка", "слова командной строки", "строки окружения" - без этого трудно будет осознанно программировать что бы то ни было.

```
#include<conio.h>
main(int arg_count, char* word[], char* env[])
{
//Очистим экран, чтобы вывод был на "чистую поверхность"
clrscr();
//Выводим количество слов в командной строке
cprintf("В командной строке %d слов, разделенных пробелами
\r\n", arg_count);
//Выводим сами слова командной строки
cprintf("Слова командной строки: \r\n");//Что-то вроде заголовка
for(int i=0;i<arg_count;i++)          //Цикл вывода слов
    cprintf("%s\r\n",word[i]);
//Выводим строки среды, сформированные командами DOS
//path, set
cprintf("Строки среды окружения: \r\n");//Опять заголовок
for(i=0;env[i];i++)          //Опять цикл вывода строк
    cprintf("%s\r\n",env[i]);
getch(); //Ожидаем нажатия произвольной клавиши
         //без отображения ее на экране
}
```

Другой способ получения той же информации состоит в использовании глобальных переменных с именами _argc (количество слов в командной строке), _argv (массив указателей на слова в командной строке), environ (массив указателей на строки окружения) из файла dos.h. В этом случае функция main() может оформляться без параметров, но становится обязательной директива включения #include<dos.h>:

```
#include<dos.h>
#include<conio.h>
main()
{
//Очистим экран, чтобы вывод был на "чистую поверхность"
clrscr();
//Выводим количество слов в командной строке
cprintf("В командной строке %d слов, разделенных пробелами \r\n",_argc);
//Выводим сами слова командной строки
cprintf("Слова командной строки: \r\n");
for(int i=0;i<_argc;i++)
     cprintf("%s\r\n",_argv[i]);
//Выводим строки среды, сформированные командами DOS
//prompt, path, set
cprintf("Строки среды окружения: \r\n");
for(i=0;environ[i];i++)
     cprintf("%s\r\n",environ[i]);
getch(); //Ожидаем нажатия произвольной клавиши
         //без отображения ее на экране
}
```

Испытайте и этот вариант – мы будем использовать их поочередно практически во всех дальнейших примерах.

6.2. Предостережения о синтаксических ошибках, которых вы должны старательно избегать.

В первых программах они неизбежны и поэтому мы сообщаем вам заранее наиболее распространенные типы ошибок начинающих программистов.

1. Не ставьте точку с запятой после директив препроцессору
   #include<имя_файла>
   #define имя_определения текст_подстановки

2. Всегда ставьте круглые скобки после имени функции (даже если у нее пустой список аргументов), после названий условных операторов и операторов циклов if(выражение), switch(выражение), while(выражение), for(операторы; выражение; операторы)

3. Не забывайте точку с запятой после каждого оператора в теле функции, в том числе после оператора вызова на выполнение любой функции.

```
y=x; printf("%d",x); z++;
```

   В круглых скобках цикла for в первой и третьей секциях операторы разделяйте запятыми:

```
for(i=0,j=1,k=2;i<N;i++,j+=2,k*=3)
```

4. Не забывайте, что точка с запятой ';' - это пустой оператор, эквивалентный команде "нет операции" и его установка после закрытой круглой скобки заголовка цикла или условного оператора означает отсутствие в теле операции каких-либо действий - не ставьте ее по ошибке.

5. Не забывайте ограничить тело в определении функции фигурными скобками - открывающая скобка { должна стоять сразу за круглой скобкой списка аргументов.

6. Не путайте операции присваивания = и сравнения на равенство ==.

7. Не присваивайте отрицательных значений целым беззнаковым переменным.

8. Среда разработки Си, C++ не проверяет выходы за границы массивов - будьте внимательны с индексами и пределами в циклах.

9. При обработке символьных массивов как строк следите, чтобы в конце строки был завершающий нуль-байт.

10. Избегайте работы с указателями, хранящими адреса локальных переменных - после выхода из зоны существования локальной переменной этот адрес будет адресовать бог знает что - память под локальные переменные освобождается после завершения работы функции - владельца.

А теперь программируем дальше - нет другого способа научиться этой работе.

6.3. Программа, решающая квадратные уравнения с вещественными коэффициентами, заданными в командной строке.

Здесь нет незнакомого вам материала и мы ограничимся комментариями по тексту программы; единственное, о чем напомним - это о необходимости преобразования цифровых слов командной строки в вещественные числа - мы это делаем использованием библиотечной функции `atof()`.

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

//Подпрограмма для вывода на экран решения квадратных
//уравнений

void quadr(double a,double b, double c)
{
//Дискриминант и квадратный корень из его абсолютного значения
double d=b*b-4*a*c,sqrd=sqrt(abs(d)),r=-b/(2*a),i=sqrd/(2*a);

//Если дискриминант положителен или равен нулю
if(d>=0)
    cprintf("Корни вещественные:\r\nx1 = %lf\r\nx2=
%lf", r+i,r-i);
else
cprintf("Корни комплексные:\r\nx1 = %lf+%lfj\r\nx2 = %lf-
%lfj", r,i,r,i);
}

void main(int count,char* w[])
{
 if(count<4)
 {cprintf("Неполадки с коэффициентами уравнения");return;}
 double a=atof(w[1]),b=atof(w[2]),c=atof(w[3]);
 clrscr();quadr(a,b,c);

}
```

6.4. Простые числа.

Задача: вывести на экран все простые числа, не превышающие заданного в командной строке.

Алгоритм описан в разделе 4.9.1.

Прежде всего нам придется проверить, задал ли пользователь нашей программы предел для последовательности простых, и если да - "достать" из командной строки цифровое слово, преобразовать его в целое число и далее использовать в программе, которая выглядит при этом следующим образом:

```
#include <conio.h> //Очистка экрана и вывод
#include <stdlib.h> //atol - преобразование строки в число
#include <math.h> //sqrt() - извлечение корня

long limit;    //Для предела простых
const beg=5; //С этого числа начнем список простых

void main(int c,char* w[])
{
//Если не задано предельное число
if(c<2) {cprintf("Не задано предельное число");return;}

/*Цифровое слово в 1-м элементе массива слов командной строки преобразуем в длинное целое с помощью библиотечной функции atol()*/
limit=atol(w[1]);

char flag; //Для признака простое =1 или не простое =0
clrscr();      //Очистим экран
//Пока не превышен предел для делимого i с шагом 2
for(long i=beg;i<=limit;i+=2)
{
  flag=1;       //Вначале предполагаем простое

//Пробуем делители j пока не дойдем до квадратного корня делимого
for(long j=beg-2;j<=sqrt(i);j+=2)
if(!(i%j)) {flag=0;break;}//Если разделилось без остатка
//Если ни одного безостаточного деления - печатаем
if(flag) cprintf("%li ",i);
}
}
```

6.5. Определение длины строки, заданной указателем на начало и "закрытой" нулем.

Мы уже рассматривали эту и последующие задачи на модели оперативной памяти, в библиотеке такие подпрограммы

тоже есть, но для понимания процессов обработки строк этот материал важен и мы его приведем.

Пусть для начала исследуемая строка задана литералом (в реальных программах она читается из файла или вводится с клавиатуры – одним словом, поступает извне на стадии выполнения программы).

Алгоритм очень прост: выделим память для счетчика символов в строке, обнулим его, и, наращивая указатель на строку, пока не встретим завершающий нуль – байт, будем наращивать счетчик.

```
#include<conio.h>
unsigned char* str="Надо определить длину этой строки";
void main()
{
     for(unsigned char len=0;*str!=0;str++,len++);
     clrscr();
     cprintf("Длина строки = %d ",len);
}
```

6.6. Копирование строки, заданной указателем на начало и заканчивающейся двоичным нулем.

Алгоритм очевиден – наращиваем индекс, пока не скопируем нулевое значение в буфер назначения.

```
#include<conio.h>
unsigned char* str="Надо скопировать эту строку ";
unsigned char dest[256];//Буфер назначения – копируем сюда
void main()
{int i=0;
clrscr();
for(dest[i]=str[i];dest[i];) {i++;dest[i]=str[i];}
cprintf("Строка – оригинал: %s\n\rСтрока – копия:
         %s",str,dest);
}
```

6.7. Сравнение 2-х строк.

Алгоритм описан в разделе 4.9.2.

```
#include<conio.h>
unsigned char* str1="Наш дядя самых честных правил",
          * str2="Наш дядя самых честных правил",
```

```
        * str3="Наш дядя не самых честных правил";
//Подпрограмма сравнения строк
int compstr(unsigned char*s1,unsigned char*s2)
{for(int r=*s1-*s2;(*s1+*s2) && !r;r=*++s1-*++s2);
return r;}

//Главная функция тестирует подпрограмму сравнением
//str1 с str2  и str3

void main(){clrscr();
if(!compstr(str1,str2)) cprintf("\r\nСтроки 1 и 2 равны");
if(compstr(str1,str2)>0)cprintf("\r\nСтрока 1 больше 2-й ");
if(compstr(str1,str2)<0)cprintf("\r\nСтрока 1 меньше 2-й ");
if(compstr(str1,str3)<0)cprintf("\r\nСтрока 1 меньше 3-й ");
if(compstr(str1,str3)>0)cprintf("\r\nСтрока 1 больше 3-й ");
if(!compstr(str1,str3)) cprintf("\r\nСтрока 1 равна 3-й ");
}
```

6.8. Выделение и преобразование в целое цифровой подстроки в строке, заданной указателем на начало и завершающейся нулем.

Алгоритм в разделе 4.9.3.

```
#include<conio.h>

int str_digit(char*s){
int d=0;//Здесь будет формироваться число из цифровой подстроки
while((*s<'0') || (*s>'9') && *s) s++; //Пока не цифра и не нуль
while((*s>='0') && (*s<='9'))        //Пока цифра
{//Преобразуем в число с учетом позиции цифры
d=(*s-'0')+10*d;s++;}
return d;
}
//Теперь тестирование в главной функции
unsigned char* str="Цена этого изделия 1860 рублей";
void main()
{clrscr();
cprintf("\r\nВыделенная подстрока содержит \
        число %d",str_digit(str));
}
```

6.9. Печать отрезка ряда чисел Фибоначчи с длиной, заданной в командной строке.

Каждый член этого ряда - натуральное число, равное сумме 2-х предшествующих членов ряда. Очевидно, что 2 первых члена придется определить "волевым порядком" - например, как 1 и 1.

```
#include<conio.h>
#include<dos.h>
#include<stdlib.h>

//Функция, печатающая заданное количество членов ряда Фибоначчи
void fib(unsigned n)
{unsigned long sum;
unsigned long fa[2]={1,1};//Два первых члена ряда Фибоначчи
cprintf("\r\n%-5lu%\r\n-5lu",fa[0],fa[1]);
for(unsigned i=2;i<n;i++)
{sum=fa[0]+fa[1];      //Вычисляем следующее
cprintf("\r\n%-5lu ",sum); //Печатаем
fa[0]=fa[1];fa[1]=sum;}    //Сдвигаем влево на 1 число
}

void main()
{
if(_argc<2)
{cprintf("Не задано факториальное число");return;}
unsigned n=atoi(_argv[1]); //Определяем количество чисел
clrscr();fib(n);} //Вызываем специалиста по Фибоначчи
```

6.10. Наибольший общий делитель.

Подпрограмма обмена значениями между двумя областями памяти, заданными указателями на их начало и размером (одинаковым для обеих в нашем примере) и ее использование в алгоритме Евклида для определения наибольшего общего делителя.

Подпрограмма обмена выполнена универсальной, пригодной для любых типов любых размеров - обратите внимание, что для реализации этого служит указатель на неопределенный тип void.

Сам алгоритм Евклида выполнен в вызываемой из main() функции nod() в классическом варианте последовательным вычитанием до тех пор, пока числа не сравняются, а для упорядочения чисел по убыванию nod(), в свою очередь, вызывает swapobj().

```
//Для использования библиотечных функций работы с областями памяти
#include<mem.h>
#include<stdlib.h>
#include<stdio.h>
//Подпрограмма, меняющая местами содержимое 2-х объектов
//в памяти, заданных своими указателями и размером
void swapobj(void *obj1,void*obj2,int size)
{//выделим память для временного сохранения одного из объектов
 void * tmp= malloc(size);
 //Если память не выделена
 if(tmp==NULL) {puts("Нехватка памяти в куче");exit(1);}
 memmove(tmp,obj1,size);//перегрузим в "запасник" один объект
 memmove(obj1,obj2,size);//второй на место первого
 memmove(obj2,tmp,size);//из запасника в область памяти второго
 free(tmp);          //освободим арендованную память
}

//Подпрограмма, определяющая наибольший общий делитель
//двух целых чисел, использующая swapobj()

long nod(long d1,long d2)
{
  long r;
  for(;d1!=d2;)
  {
   if(d1<d2) swapobj(&d1,&d2,sizeof(d1));
   d1=d1-d2;
  }
  return d1;
}

void main()
{
 clrscr();
 if(_argc!=3) {puts("Непорядок с аргументами");return;}
 long n1=atol(_argv[1]),n2=atol(_argv[2]),r;
 cprintf("НОД чисел %ld и %ld равен %ld",d1,d2,nod(n1,n2);
}
```

6.11. Программирование рекурсивных алгоритмов.

Язык Си поддерживает программирование рекурсивных алгоритмов, любая подпрограмма может вызвать любую и быть вызванной из любой подпрограммы, включая главную функцию `main()`.

При рекурсивных вызовах все происходит так, как будто обращение осуществляется к другому экземпляру этой же подпрограммы. Рекурсий в программной реализации следует избегать – они приводят к большим затратам времени на вызовы и возвраты и усиленно расходуют память стека. Но рекурсия часто позволяет с меньшими затратами усилий составить алгоритм решения, а потом, после уяснения его особенностей, переписать с помощью циклов – это всегда возможно, но не всегда легко.

6.11.1. Простейший пример использования рекурсии – вычисление факториала натурального числа.

Он равен произведению числа на факториал числа, уменьшенного на 1, и просто напрашивается вызов из подпрограммы вычисления факториала N самой себя для вычисления факториала N-1.

В приведенной ниже программе есть 2 подпрограммы – вычисление факториала через рекурсивные вызовы и вторая – без рекурсии. Обе вызываются из функции main() прямо из подпрограммы печати.

```
#include<conio.h>
#include<dos.h>
#include<stdlib.h>

//Подпрограмма рекурсивного алгоритма вычисления факториала
long rfact(unsigned n)
{if(n==1) return 1l; else return n*rfact(n-1);}

//Подпрограмма нерекурсивного алгоритма вычисления факториала
long nrfact(unsigned n)
{long f=1; for(;n>1;n--) f*=n;return f;}

void main(){
if(_argc<2)
{cprintf("Не задано факториальное число");return;}
unsigned n=atoi(_argv[1]);
clrscr();
cprintf("Факториал числа %d равен %ld",n,rfact(n));
cprintf("\r\nФакториал числа %d равен %ld",n,nrfact(n));
}
```

6.11.2. Рекурсии - задача о Ханойских башнях.

На одном из 3-х стержней надето N дисков с убывающими к вершине радиусами; необходимо переложить диски на другой стержень в том же порядке с использованием 3-го как промежуточного, но нельзя класть больший диск на меньший.

Рекурсивный и нерекурсивный алгоритмы - в разделе 4.9.4.

```
#include<conio.h>
#include<dos.h>
#include<stdlib.h>
#include<math.h>
//Рекурсивный алгоритм "Ханойские башни"
void hanojrec(unsigned n, char a, char b, char c)
{if(n==1) cprintf("%c->%c ",a,b);
 else
 {hanojrec(n-1,a,c,b);
 hanojrec(1,a,b,c);
 hanojrec(n-1,c,b,a);}
}

//Нерекурсивный алгоритм "Ханойские башни"
void hanoj(unsigned char n)
{
//Выделим память под массивы дисков на всех стержнях сразу
//как для утроенного массива a
unsigned char* a=(unsigned char*)malloc(3*(n+1));
//Проверим выделена ли память
if(a==NULL)
{cprintf("Не удалось получить память в куче");return;}
//Определим указатели на массивы b и c
unsigned char* b=a+n+1,*c=b+n+1,
//Начальные присвоения переменным
atop=0,btop=n,ctop=n,min='a';b[btop]=n,c[ctop]=n;
for(int i=0;i<(n+1);i++) a[i]=i;
//Реализуем описанный алгоритм перекладывания дисков
 //Пока все диски не окажутся на стержне 'c' или 'b'
for(;btop!=0 && ctop!=0;)
{
 if(btop==0 || ctop==0) break;
 //В зависимости от того, где сейчас самый маленький диск
 //перекладываем его по часовой стрелке
 switch(min){
 case 'a':
 cprintf("%c->%c ",min,min+1);min++;b[--btop]=a[atop++];break;
 case 'b':
 cprintf("%c->%c ",min,min+1);min++;c[--ctop]=b[btop++];break;
 case 'c':
 cprintf("%c->%c ",min,min-2);min-=2;a[--atop]=c[ctop++];break;
```

```
}
/*Минимальный диск изменил позицию и мы занимаемся парой стержней, на которых его нет */

switch(min){
case 'a':if(b[btop]<c[ctop])
{cprintf("%c->%c ",'b','c');c[--ctop]=b[btop++];break;}
if(c[ctop]<b[btop])
{cprintf("%c->%c ",'c','b');b[--btop]=c[ctop++];}break;

case 'b':if(a[atop]<c[ctop])
{cprintf("%c->%c ",'a','c');c[--ctop]=a[atop++];break;}
if(c[ctop]<a[atop])
{cprintf("%c->%c ",'c','a');a[--atop]=c[ctop++];}break;
case 'c':if(a[atop]<b[btop])
{cprintf("%c->%c ",'a','b');b[--btop]=a[atop++];break;}
if(b[btop]<a[atop])
{cprintf("%c->%c ",'b','a');a[--atop]=b[btop++];}break;
}
}
free(a);
}

/*В главной функции вызываются на выполнение по очереди рекурсивный и нерекурсивный варианты для сравнения результатов */
void main(){
if(_argc<2) {cprintf("Не задано количество дисков");return;}
unsigned n=atoi(_argv[1]);
clrscr();
hanojrec(n,'a','b','c');
cprintf("\r\n");
hanoj(n);
}
```

6.11.3. Рекурсии – программный генератор перестановок N попарно различных чисел.

Описание алгоритма – в разделе 4.9.5.

Получение количества переставляемых элементов, выделение памяти под массив их индексов и начальное присвоение значений элементам этого массива выполнено в главной функции, оттуда же вызываются на выполнение оба варианта генератора – рекурсивный и нерекурсивный.

```
#include<conio.h>
#include<stdlib.h>
#include<dos.h>
```

```
unsigned N,*AN; //Для количества чисел и указателя на начало
                //массива индексов.

//Рекурсивный генератор перестановок
void permute_rec(unsigned * a,unsigned n)
{
unsigned temp,number;int j;
if(n>1){ permute_rec(a,n-1);
//Переставляем значения элементов a[n-1] и a[i-1]
for(int i=n-1;i>=1;i--)
{temp=a[n-1];a[n-1]=a[i-1];a[i-1]=temp;permute_rec(a,n-1);
//Переставляем значения элементов a[n-1] и a[i-1]
temp=a[n-1];a[n-1]=a[i-1];a[i-1]=temp;
}
}
else {for(j=1;j<=N;j++) cprintf("%u",a[j-1]); cprintf("\r\n");}
}

unsigned p; //Для счетчика количества перестановок;

//Генератор перестановок без использования рекурсии
void permute(unsigned* a,unsigned n)
{
unsigned tmp,j,k,l;

for(;;)
{
//Ищем правую границу, упорядоч. по возрастанию
for(int i=n-2;(a[i]>a[i+1]) && (i>=0);i--);
if(i<0) break;                  //Закончили - перестановки все
//Ищем самый правый больший граничного
for(j=n-1;(j>i) && a[j]<a[i];j--);
tmp=a[i];a[i]=a[j];a[j]=tmp;    //Переставляем
for(j=i+1,k=1;j<(i+1+n)/2;j++,k++)//Реверсируем хвост массива
{tmp=a[j];a[j]=a[n-k];a[n-k]=tmp;}
for(l=0;l<n;l++)
cprintf("%d",a[l]);cprintf("\n\r");p++;
}
gotoxy(20,12);cprintf("Перестановки без рекурсии всего =%d",p);
}

void main()
{
 if(_argc<2) {cprintf("\r\nНе задан размер массива");return;}
 //Берем из командной строки количество элементов
 N=atoi(_argv[1]);
//Просим память под массив для индексов переставляемых объектов
 AN=(unsigned*)malloc(sizeof(unsigned)*N);
```

```
 if(AN==NULL) {cprintf("Проблемы с выделением памяти");return;}
 clrscr();
//заполняем числами натурального ряда
//и печатаем стартовую последовательность
for(int i=0;i<N;i++)
{AN[i]=i;cprintf("%d",i);} cprintf("\r\n");p++;
permute(AN,N);getch(); //Пауза до нажатия клавиши
clrscr();
permute_rec(AN,N);
gotoxy(20,12);cprintf("Рекурсивные перестановки");getch();
free(AN);
}
```

*6.11.4. Задача: найти все возможные варианты расстановок N не бьющих друг друга ферзей на шахматной доске размером N*N.*

Описание рекурсивного алгоритма решения в разделе 4.9.6.

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

unsigned * F,//Адрес массива для хранения варианта расстановки
*COL,          //то же для индикации занятости столбца
*DS,       //для индикации занятости диагонали, у которой
               //постоянна сумма индексов столбцов и строк
*DR,       //то же для постоянной разности индексов
N,             //для размерности матрицы
ND,        //для количества диагоналей
row,       //для текущего индекса строки
number;

    /*Нам будет удобно составить предварительно несколько вспомогательных подпрограмм - это улучшит читабельность программы */

//Функция, определяющая, свободно ли поле для установки
int isfree(unsigned row, unsigned col)
{if(COL[col] && DS[row+col] && DR[N-1+row-col]) return 1;
else return 0;}

//Функция, занимающая поле
void setfild(unsigned row,unsigned col)
```

```
{F[row]=col;COL[col]=0;DS[row+col]=0;DR[N-1+row-col]=0;}

//Функция, освобождающая поле
void freefild(unsigned row,unsigned col)
{COL[col]=1;DS[row+col]=1;DR[N-1+row-col]=1;}

//Функция печати очередной расстановки
void printconf(void)
{
printf("Номер расстановки:%-6d",number);
for(char i = 0;i<N; i++) printf("%d",F[i]); putchar('\n');}

//Рекурсивная функция генерации расстановок
void generation(void){
unsigned col=0;
do{
//Если поле[row][col] свободно
if(isfree(row,col))
//займем его занеся номер столбца в массив расстановки и
//обозначив нулями занятость столбца и диагоналей
{
setfild(row,col);
row++;
//Если все строки заняты
if(row==N)
//печатаем очередную конфигурацию искомых элементов
{number++;printconf();}
/*иначе рекурсивно вызываем саму себя, но уже с новой исследуемой строкой */
else generation();
//Возвращаемся на строку назад для исследования в ней
//следующего столбца
row--;
//Освобождаем ранее занятое в ней поле[row][col]
freefild(row,col);
}
//Исследуем следующий столбец в строке row
col++;
}while(col<N);
}


void main(int c,char* w[]){
clrscr();
if(c!=2)
{puts("Неполадки с параметрами - задайте только размер доски");
return;}
N=atoi(w[1]);//Размер матрицы
```

```
ND=2*N-1;       //Количество диагоналей

//Выделим память под все массивы сразу как под массив результата
F=(unsigned *)malloc((2*N+2*ND)*sizeof(unsigned));
if(F==NULL) {puts("Что-то с памятью ");return;}

//Определим указатели на другие массивы в выделенной памяти
COL=F+N;DS=COL+N;DR=DS+ND;

//"Объединичим" все массивы сразу -
//т.е. обозначим свободными все поля
for(int i=0;i<N+2*ND;i++) COL[i]=1;
/*Начальная подготовка закончена - вызываем функцию генерации расстановок */
     generation(); free(F); }
```

Подпрограмма решения этой же задачи без использования рекурсивных вызовов приведена ниже и мы надеемся, что вы разберетесь в ней без труда.

```
void generation(void){
unsigned col=0,col_temp;
for(;;)
{
do{
//Если поле[row][col] свободно
if(isfree(row,col))
//займем его занеся номер столбца в массив расстановки и
//обозначив нулями занятость столбца и диагоналей
{
setfild(row,col);row++;col=0;
//Если все строки заняты
if(row==N)
{//печатаем очередную конфигурацию искомых элементов
{number++;printconf();}
//возвращаемся на строку с неисчерпанным списком столбцов
while(row>0){row--;col_temp=F[row];freefild(row,col_temp);
if(col_temp<(N-1)) break;}
//Если такой строки нет - все конфигурации отпечатаны
if(row==0 && col_temp==N-1) return;
col=col_temp+1;
}//if по все строки заняты
}//if по если поле свободно
//Исследуем следующий столбец в строке row
else col++;
}while(col<N);
//возвращаемся на строку с неисчерпанным списком столбцов
```

```
while(row>0){row--
;col_temp=F[row];freefild(row,col_temp);
if(col_temp<(N-1)) break;}
//Если такой строки нет - все конфигурации отпечатаны
if(row==0 && col_temp==N-1) return;
col=col_temp+1;
}//for(;;)
}
```

6.12. Простые числа - решето Эратосфена и битовая упаковка булевских переменных.

Задача: вывести на экран все простые числа, меньшие заданного пользователем, используя алгоритм Эратосфена "просеивания" нечетных чисел.

Алгоритм решения в разделе 4.9.7.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <mem.h>

unsigned limit, //количество чисел
candidat,nonprost; //индексы кандидатов в простые и непростые
char*bit;     //адрес массива для побитовой работы
int d;         //для значения числа
int nbit;     //для номера бита в активном байте
char *byte_ptr; //для адреса активного байта
const first=3;     //первое простое


//подпрограмма обнуления бита с заданным номером i в массиве
//char-элементов с указателем bit

inline void sbros_bit(long i,char*b)
{char*bptr=b+(i>>3);//номер байта с заданным битом char
nbit=7-(i&7); //номер бита в байте
*bptr&=~(1<<nbit);
}
//Собственно алгоритм Эратосфена
void eratosfen()
{
//необходимая память в байтах для хранения битовой модели
//массива нечетных - примерно вдвое меньше заданного пре-
//дельного числа, т.к. нечетных - половина
```

```
unsigned lbyte=limit>>2; //адрес выделенного массива
bit=(char*)malloc(lbyte); //выделим память
setmem(bit,lbyte,'\xff');//заполним массив единицами

//цикл перебора битовых элементов массива
for(candidat=0;candidat<=limit;candidat++)
{
byte_ptr=bit+(candidat>>3); //адрес активного байта
nbit=7-(candidat & 7); //номер бита счетом справа налево
if(*byte_ptr & (1<<nbit)) //если число простое
{d=2*candidat+first;   //вычисляем его
printf("%d ",d);        //и печатаем
nonprost=d+candidat;/*находим индекс непростого (кратного простому) */
for(;nonprost<=limit;)     /*в цикле обнулим все кратные текущему простому */
{sbros_bit(nonprost,bit);
nonprost+=d;
} } //if }
free(bit);              //освободим память
}
//И теперь главная функция main()
void main(int c,char*w[])
{
if(c<2) {cprintf("Не задан предел для простых");return;}
//Получаем количество нечетных чисел - их половина
limit=atoi(w[1])>>1;
clrscr();
eratosfen(); //Вызываем подпрограмму печати простых чисел
}
```

6.13. Примеры простых программ, работающих с файлами.

6.13.1. Запись в файл.

Вывести в файл с заданным в командной строке именем текущую таблицу кодировки символов.

План решения в разделе 4.9.8.

А вот так это будет выглядеть на языке Си:

```
#include<stdio.h> //Для работы с файлом и вывода на экран
#include<dos.h>   //Для работы с параметрами командной
                  //строки _argc,_argv
```

```
void main()
{
 int i,j;
 char kod;
 if(_argc!=2)
 {printf("Не задано имя файла под таблицу кодировки"); return;}
 FILE*f=fopen(_argv[1],"w");
 if(f==NULL)
 {printf("Неудача с открытием указанного файла");return;}
 //Вначале выведем строку с номерами столбцов таблицы
 fprintf(f," ");
 for(i=0;i<16;i++)
 fprintf(f,"%-4x",i);fprintf(f,"\n");
 //А теперь строки с символами
 for(i=0;i<16;i++)
 {
 fprintf(f,"%-4x",i); //Выводим номер строки
 for(j=0;j<16;j++)     //и символы
 {kod=i*16+j;
 if(kod==9 || kod==10 || kod==13)
 fprintf(f,"%-4s","УС");
 else fprintf(f,"%-4c",kod);}
 fprintf(f,"%c",'\n');
}
 fclose(f);
 }
```

Наберите эту программу в редакторе интегрированной среды, консультируясь с преподавателем по поводу каждого непонятного вам оператора. Имя файла для таблицы, например `char.tbl`, задайте в подменю `Argument` меню `Run`. С помощью `F3` вызовите на экран полученный файл и посмотрите на результат. Усовершенствуйте таблицу добавлением горизонтальных и вертикальных разделительных линий между столбцами и строками – коды соответствующих символов вы уже можете взять из полученной файловой таблицы. Выйдите из интегрированной среды и вызовите на выполнение вашу программу из командной строки ОС.

6.13.2. Чтение из файла.

Составить программу, подсчитывающую относительные частоты всех символов (частное от деления общего количества появлений каждого символа на суммарное число символов в анализируемом тексте) латинского алфавита в тексте из произвольного, предположительно англоязычного, файла, имя которого задано в командной строке ОС.

Алгоритм в разделе `4.9.9`.

Вот как это выглядит на Си:

```
#include<stdio.h>
#include<conio.h> //Для очистки экрана
#include<ctype.h>
#include<dos.h>
long freq[26];     //Массив счетчиков
void main()
{
 char ch;
 long count=0;//Общий счетчик букв
 if(_argc!=2)
 {printf("Не задано имя файла под таблицу кодировки"); return;}
 FILE*f=fopen(_argv[1],"r");
 if(f==NULL)
 {printf("Неудача с открытием указанного файла");return;}
 //Посимвольная обработка содержимого файла
 for(ch=fgetc(f);!feof(f);ch=fgetc(f))
 if(ch>='A' && ch<='z') //Если прочитанный символ буква
 {count++;              //Наращиваем счетчик букв
 if(ch>='a' && ch <='z') //Если буква маленькая
 ch=ch+'A'-'a';         //делаем ее большой
 freq[ch-'A']++;   //Наращиваем соответствующий счетчик
 }
 //Вывод результата на экран в 2 колонки
 clrscr();
 for(int i=0;i<26;i++)
 {
 gotoxy(i/13+40*(i/13),i-12*(i/13)); //Установка курсора
 printf("Частота символа %c - %4f\n",
 i+'A',(float)freq[i]/(float)count);
 } }
```

6.14. Простые примеры непосредственной работы с видеобуфером.

Рассмотрим некоторые варианты "бегущих" строк, выполняемых прямой записью в видеобуфер:

1. Горизонтальная строка движется горизонтально.

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

/*Инициализируем указатель vb на видеобуфер в 3-м текстовом режиме значением числового адреса видеобуфера - и дальше обращаемся с ним как с обычным одномерным массивом. */

```
char far*vb=(char far*)0xb8000000;
char* fam="Фамилия"; //Эта строка будет двигаться

void main()
{int i,j;clrscr();
/*По видеопамяти будем двигаться с шагом 2 (индекс i) по чет-
ным байтам кодов символов, а по строке с шагом 1 (индекс j)*/

     for(i=0;i<4000;i+=2)
     {
for(j=0;j<strlen(fam);j++) vb[i+2*j]=fam[j];//Выводим строку
delay(200);                          //Задержка 200 мс
vb[i]=' ';                           //Стираем букву
} }
```

Остальные варианты не будем подробно комментировать – ничем, кроме формирования индексов и процедур стирания "следа" от предыдущего вывода они не отличаются – разберитесь и составьте другие варианты движения строк, например, по диагоналям.

2. Вертикальная строка движется вертикально

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

char far*vb=(char far*)0xb8000000;
char* fam="Фамилия";

void main()
{int i,j;clrscr();
for(i=0;i<4000;i+=2*80)
{for(j=0;j<strlen(fam);j++) vb[i+80*2*j]=fam[j];
delay(200);vb[i]=' ';}
}
```

3. Горизонтальная строка бежит вертикально.

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

char far*vb=(char far*)0xb8000000;
char* fam="Фамилия";

void main()
{int i,j;clrscr();
for(i=0;i<4000;i+=2*80)
{for(j=0;j<strlen(fam);j++) vb[i+2*j]=fam[j];
delay(200);//vb[i]=' ';
for(j=0;j<strlen(fam);j++) vb[i+2*j]=' ';
```

```
}
}
```

4. Вертикальная строка бежит горизонтально.

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

char far*vb=(char far*)0xb8000000;
char* fam="Фамилия";

void main()
{int i,j;clrscr();
for(i=0;i<160;i+=2)
{for(j=0;j<strlen(fam);j++) vb[i+2*80*j]=fam[j];
delay(200);//vb[i]=' ';
for(j=0;j<strlen(fam);j++) vb[i+2*j*80]=' ';
}
}
```

6.15. Пример работы с таблицей знакогенератора.

Задача: подменить таблицу знакогенератора своей, которая будет отличаться только изображением одного из символов псевдографики и вывести этот символ на экран для демонстрации, а после нажатия произвольной клавиши восстановить стандартную таблицу и вывести символ с тем же кодом.

Необходимая справочная информация в разделе 4.9.10.:

```
#include<dos.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
char far*tz;   //указатель на таблицу знакогенератора будет здесь
char s;        //число строк в рисунке символа
struct REGPACK r; //Структура для работы с регистрами через функцию
                  //генерации прерывания intr()
char far* buf;//Для указателя на буфер самодельной таблицы
//Массив байтов, описывающий рисунок шахматного ферзя
char ch[]= {0x00,0x00,0x22,0x66,0xe7,0x24,0x24,0x24,
           0xa5,0xa5,0xa5,0xff,0xff,0xff,0xff,0x00};
void main()
{ int i,j;
//Заполняем регистровую структуру и вызываем прерывание
r.r_ax=0x1130;r.r_bx=0x0600;intr(0x10,&r);
tz=(char far*)MK_FP(r.r_es,r.r_bp);//Получили адрес таблицы
```

```
s=r.r_cx;                          //Получили высоту символа
//Выделим память в куче под свою таблицу того же размера,
//что и стандартная и s байтов для своего рисунка
buf=(char far *)malloc(256*s);
//Копируем стандартную таблицу в свой буфер
for(i=0;i<256;i++)
for(j=0;j<16;j++)
buf[i*s+j]=tz[i*s+j];
//Меняем рисунок символа с кодом 0xd4 на свой
for(i=0;i<s;i++) buf[s*0xd4+i]=ch[i];
//Очищаем экран и выводим символ стандартным рисунком
clrscr();
for(i=1;i<39;i++) cprintf("%c ",0xd4);cprintf("\r\n");
//Подставляем адрес своей таблицы вместо стандартной
r.r_ax=0x1100;
r.r_es=FP_SEG(buf);r.r_bp=FP_OFF(buf);
r.r_cx=256;r.r_dx=0;r.r_bx=0x1000;intr(0x10,&r);
getch();//Любуемся проделанным
//Возвращаем старую таблицу
r.r_ax=0x1104;r.r_bx=0;intr(0x10,&r);
free(buf);
}
```

6.16. Прикладное программирование в среде (Турбо, Борланд) C, C++ на более сложных примерах.

6.16.1. Обработка одномерных числовых массивов (векторов).

Общие сведения о задачах этого класса приведены в 4.9.11.

1. Мы начнем с одной простой задачи обработки файлового массива, затем создадим многофункциональную программу обработки и попытаемся постепенно совершенствовать диалог с пользователем для повышения комфортности в его работе с нашей программой.

```
//Программа заполнения файла N случайными числами
#include<stdlib.h>
#include<stdio.h>
const N=200,M=100;
void main(){
FILE* f=fopen("rndvect.txt","w");//Открываем файл для записи
```

```
if(f==NULL)
{printf("\r\nНеудача при открытии файла ");return;}
for(int i=0; i<N;i++)
fprintf(f,"%lf ",
M*(double)(rand()-(RAND_MAX>>1))/(double)RAND_MAX);
fclose(f);
}
```

Теперь у нас есть файл для обработки с именем rndvect.txt - это имя мы и будем задавать в командной строке.

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

void func0(char*, unsigned);
```

/*Это прототип функции обработки – ее определение мы составим позже. Функция не будет возвращать значение, а аргументами ее будут указатель на строку с именем файла и размер массива (количество чисел в нем). В объявлении прототипа должны присутствовать возвращаемый тип, имя функции и в круглых скобках список типов аргументов – имена аргументов (формальных параметров) не обязательны в прототипе, т.к. у прототипа нет тела, нет операторов обработки – сразу за круглой скобкой ставится точка с запятой или запятая (при объявлении списка прототипов функций, отличающихся только именами).*/

```
void main(int c,char* w[])
{
if(c<2)
{cprintf("\r\nНет имени файла в командной строке");return;}
unsigned r; //Для размера массива
//Можно начинать диалог с пользователем
for(;;)
{clrscr();
cprintf("Введите размер массива (0 – для выхода) и Enter");
scanf("%u",&r); //Принимаем размер массива в r
if(!r) break; //При 0-м размере прекращаем работу
fflush(stdin);
func0(w[1],r);    //Иначе вызываем функцию обработки
getch();
}
}
```

Пусть подпрограмма обработки определяет среднее арифметическое элементов массива – для этого мы можем читать по одному элементу и тут же его подсуммировать.

```
void func0(char* fname,unsigned n)
{
//Откроем файл для чтения
FILE* f=fopen(fname,"r");
if(f==NULL)
{cprintf("\r\nНеудача при открытии файла %s",fname);return;}
double d,sum=0; //Для приема числа и подсуммирования
//Читаем слова из файла с преобразование к double и суммируем
for(int i=0;i<n;i++) {fscanf(f,"%lf",&d);sum+=(d/n);}
//Закрываем ненужный больше файл
fclose(f);
//Выводим результат обработки
clrscr(); cprintf("Среднее арифметическое = %lf",sum);
}
```

2. Многофункциональная программа обработки векторов.

Составлять отдельные программы на каждый вид обработки массивов неудобно, рациональнее предоставить весь комплекс услуг в одной программе. Вначале создадим массив текстовых строк - перечень выполняемых программой обработок массива, каждой такой строке будет поставлена в соответствие подпрограмма обработки.

Пусть таких различных функций в нашей программе будет, например, 15 - этот список вы можете расширить или сократить при необходимости. Мы не будем приводить решения всех перечисленных задач, они достаточно однообразны и вам предстоит для приобретения навыков решить их самостоятельно или с помощью преподавателя. Мы приведем реализацию 2-х функций - под номером 0 и 15, отличающихся наличием и отсутствием фрагмента выделения памяти под обрабатываемый массив; кроме того, 15-я функция демонстрирует использование указателя на функцию в качестве параметра другой функции.

Список директив включения файлов накапливается постепенно, по мере роста программы, их можно ставить в любом месте текста, но удобнее сводить в начало, что мы и делали.

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<conio.h>
#include<dir.h>
#include<bios.h>
#include<dos.h>
#include<stdlib.h>
```

/*Прототипы 2-х функций обработки, которые мы определим после текста главной функции main() */

```
void func0(char*, unsigned),func14(char*, unsigned);
```

/*Для остальных подпрограмм мы сразу приведем определения - "пустышки" для последующей самостоятельной проработки */

```
void func1(char*fname, unsigned n){};
void func2(char*fname, unsigned n){};
void func3(char*fname, unsigned n){};
void func4(char*fname, unsigned n){};
void func5(char*fname, unsigned n){};
void func6(char*fname, unsigned n){};
void func7(char*fname, unsigned n){};
void func8(char*fname, unsigned n){};
void func9(char*fname, unsigned n){};
void func10(char*fname,unsigned n){};
void func11(char*fname,unsigned n){};
void func12(char*fname,unsigned n){};
void func13(char*fname,unsigned n){};
void func14(char*fname,unsigned n){};
```

/* Для каждой задачи у нас есть ее изложение в виде строки текста - позже мы собираемся использовать его при выводе списка предоставляемых программой услуг по обработке одномерных массивов. К этим строкам надо "привязать" функции решения поставленных задач - и эту привязку мы осуществляем организацией массива структур с 2-мя полями - комментариями и и указателями на функции. */

```
struct list_task
{char * str; //Поле-указатель на строку названия задачи
void (*func)(char*,unsigned);//Поле-указатель на функцию решения
}task[]=
```

/*Каждому полю мы сразу присваиваем значение, а для удобства последующего вывода конец списка обозначаем нуль-указателями на строку и на функцию */

```
{
"Среднее арифметическое элементов массива",func0,
"Центрировать массив относительно среднего",func1,
"Среднее квадратов элементов центрированного массива",func2,
"Среднее поэлементного произведения \
2-х центрированных массивов",func3,
"Скалярное произведение 2-х векторов",func4,
```

```
"Модуль массива как вектора и направляющие косинусы",func5,
"Сумма и разность 2-х векторов",func6,
"Проекция 1-го вектора на направление 2-го",func7,
"Угол между 2-мя векторами в градусах",func8,
"Максимальное, минимальное значение и их индексы",func9,
"Максимальное и минимальное абсолютных значений",func10,
"Суммы положит и отрицат значений и их количества",func11,
"Произведение отличных от 0 положит и отрицат",func12,
"Отсортировать массив по неубыванию методом пузырька",func13,
"Отсортировать массив быстрой сортировкой",func14,
NULL,NULL
};
```

/*В многофункциональной программе придется дополнить диалог с пользователем возможностью выбора выполняемой функции обработки. Для начала мы сделаем это так же, как принимали размер массива, а затем попытаемся улучшить сервис нашей программы */

```
void
main(int c,char* w[]) {
if(c<2) {cprintf("\r\nНет имени файла в командной строке");
return;}
unsigned r; //Для размера массива int
nf; //Для номера функции
//Можно начинать диалог с пользователем
for(;;)
{clrscr(); cprintf("Введите размер массива (0 - для выхода) и Enter");
scanf("%u",&r); //Принимаем размер массива в r
if(!r) break; //При 0-м размере прекращаем работу
fflush(stdin); //Очищаем буфер ввода (клавиатурный)
cprintf("Введите N функции (от 0 до 17, -1 для выхода) и Enter");
scanf("%u",&nf); //Принимаем номер функции в nf
if(nf<0) break; //При отриц. номере прекращаем работу
fflush(stdin); //Очищаем буфер ввода (клавиатурный)
if(nf==0) //Временный вариант - если это уже решенная задача 0
task[nf].func(w[1],r); //Вызываем функцию обработки func0
else
task[1].func(w[1],r);//Иначе вызываем "заглушку" func1
getch(); }
}
```

/*Теперь нам придется определить две функции обработки, у которых пока есть только прототипы - это функция определения среднего арифметического всех элементов массива и функция сортировки массива. */

```
//Среднее арифметическое элементов массива
void func0(char* fname,unsigned n)
{
//Откроем файл для чтения
FILE* f=fopen(fname,"r");
if(f==NULL)
{window(42,10,80,25);
cprintf("\r\nНеудача при открытии файла %s",fname);return;}
double d,sum=0; //Для приема числа и подсуммирования
/*Читаем слова из файла с преобразование к double и суммируем с попутным делением на размер массива*/
for(int i=0;i<n;i++) {fscanf(f,"%lf",&d);sum+=(d/n);}
//Закрываем ненужный больше файл
fclose(f);
//Выводим результат обработки
clrscr(); cprintf("Среднее арифметическое = %lf",sum);
}

    /*Функция сравнения элементов типа double - указатель на нее надо дать аргументом библиотечной функции быстрой сортировки qsort().*/
    cmpf(const void* a, const void* b)
    {if(*(double*)a==*(double*)b) return 0;
    if(*(double*)a<*(double*)b) return -1;
    if(*(double*)a>*(double*)b) return 1;}

//Отсортировать массив быстрой сортировкой
void func15(char* fname,unsigned n)
{
//Откроем файл для чтения
FILE* f=fopen(fname,"r");
if(f==NULL)
{window(42,10,80,23); //Окно для вывода результирующего массива
cprintf("\r\nНеудача при открытии файла %s",fname);return;}
//Выделим память для массива
double* a=(double*)malloc(n*sizeof(double));
if(a==NULL)
{window(42,10,80,23);//Окно для сообщения об ошибке
cprintf("\r\nНеудача выделения памяти %s",fname);return;}
//Читаем слова из файла с преобразование к double в массив a
for(int i=0;i<n;i++) fscanf(f,"%lf",a+i);
//Закрываем ненужный больше файл
fclose(f);
//Сортируем с помощью библиотечной функции qsort()
qsort((void *)a, n, sizeof(double), cmpf );
//Выводим результат обработки
window(42,10,80,23); clrscr();
for(i=0;i<n;i++) cprintf("%.2lf ",a[i]);
```

```
free(a); //Освобождаем арендованную память
}
```

2.1. Фильтрованный посимвольный прием ввода пользователя.

Человеку свойственно ошибаться – и пользователь вашей программы будет это делать при вводе запрашиваемых у него данных. Если, например, при наборе номера задачи или размера массива он введет нецифровой символ, функция форматированного приема строки не сможет преобразовать ее в целое и программа аварийно завершится.

В диалоговых программах это недопустимо – всегда надо давать возможность пользователю исправить свою ошибку. Это можно осуществить по разному – принимать, например строку вначале без преобразования в целое с форматом %s, анализировать наличие недопустимых (в нашем случае нецифровых) символов и возвращаться при необходимости к повторению ввода сначала.

В этом разделе мы познакомим вас с другим простым подходом – контролируемым или фильтрованным посимвольным приемом вводимых данных, при котором каждый символ принимается с клавиатуры отдельно без эха на экран, анализируется на допустимость и при положительном результате отображается на экране и заносится в строку. Если он ошибочен – то просто игнорируется и так до нажатия Enter, после которого строка преобразуется в число и далее все как и раньше. Так мы просто отсекаем попытки ввести недопустимый символ. Разумеется, это работает в простейших случаях, а в более сложных приходится комбинировать фильтрацию с анализом.

```
const Enter=13; //Фиксируем код клавиши Enter

//Перепишем main() с учетом сказанного

void main(int c,char* w[])
{
if(c<2)
{cprintf("\r\nНет имени файла в командной строке");return;}
char ch;  //Для приема символа без отображения
char s[10]; //Массив для промежуточного приема строки
unsigned r; //Для размера массива
int nf;   //Для номера функции
//Можно начинать диалог с пользователем
for(;;)
```

```
{clrscr();
cprintf("\r\nВведите размер массива (0 - для выхода)\
     и Enter: ");
//Принимаем размер массива в s посимвольно до Enter
for(int i=0,ch=getch();ch!=Enter;)
{
if(ch>='0' && ch <='9') //Если принятый символ цифра
{s[i]=ch;                //Заносим его в строку s
putch(ch);               //Выводим его на экран
i++;                 //Наращиваем счетчик символов
}
ch=getch();          //Читаем следующий
}
s[i]=0;              //Закрываем строку нулем
r=atoi(s);               //Преобразуем в целое
if(!r) break; //При 0-м размере прекращаем работу
fflush(stdin);     //Очищаем буфер ввода (клавиатурный)
cprintf("\r\nВведите N функции (от 0 до 17,\
     -1 для выхода) и Enter: ");
//Повторяем прием в s номера задачи до Enter
for(i=0,ch=getch();ch!=Enter;)
{
if(ch>='0' && ch <='9') //Если принятый символ цифра
{s[i]ch;             //Заносим его в строку s
putch(ch);               //Выводим его на экран
i++;                 //Наращиваем счетчик символов
}
ch=getch();          //Читаем следующий
}
s[i]=0;              //Закрываем строку нулем
nf=atoi(s);              //Преобразуем в целое
if(nf<0) break; //При отриц. номере прекращаем работу
fflush(stdin);     //Очищаем буфер ввода (клавиатурный)
if(nf==0)          //Временный вариант -
                 //если это уже решенная задача 0
task[nf].func(w[1],r);     //Вызываем функцию обработки
func0
else task[1].func(w[1],r);//Иначе вызываем "заглушку"
func1
getch();
}
}
```

2.2.Выбор услуг программы из списка услуг (меню услуг).

Для набора номера задачи в рассмотренном варианте программы пользователю придется иметь допустимый перечень задач на бумаге - это не очень удобно; было бы желательно предварительно вывести ему на экран перечень видов обработки массивов и дать возможность выбора с помощью, например, курсорных клавиш. Для этого нам придется составить соответствующую подпрограмму, которая будет возвращать номер выбранной строки и этот номер будет использоваться основной программой для запуска соответствующей функции обработки. Этим и займемся.

/*Подпрограмма, возвращающая номер выбранной пользователем строки из экранного списка строк.

Может использоваться как модель меню при запуске на выполнение различных подпрограмм, в том числе по индексу из массива подпрограмм (массива указателей на функции)*/

/*Нам понадобится работа со спецклавишами управления курсором - чтобы не осуществлять анализ типа нажатой клавиши и повторного чтения для получения старшего байта, мы будем работать с двухбайтовыми кодами и определим их в виде констант */

```
const
UP=18432,DOWN=20480,HOME=18176,END=20224,
ENTER=7181,ESC=283,
ROW_COUNT=4, //Количество строк в окне
MAX_ELEM=50, //Предельное количество строк
STR_LEN=72, //Длина строк в окне
WNDYBEG=2; //Начальная строка окна на экране
```

/*Шаблон оконных строк - вы можете создать его в любом текстовом редакторе, оснащенном макросами для рисования рамок*/

```
    char* strwnd[]=
{"╔══════════════════════════════════════════════════════════╗",
 "║                                                          ║",
 "╚══════════════════════════════════════════════════════════╝"
};
```

```
    //Атрибуты для нормальной и выделенной строк и строка
    //пробелов
    unsigned char norm_attr=0x1f, spec_attr=0x3f,
*space_str='                                                  ',
    count;  //Для общего количества подлежащих выводу
строк
```

```
                //Функция вывода окна для списка строк
     void str_wnd()
{
window(1,WNDYBEG,STR_LEN,WNDYBEG+ROW_COUNT+3);
textattr(norm_attr);cputs(*strwnd); //Выводим верхнюю рамочную
for(int i=0;i<=ROW_COUNT;cputs(strwnd[1]),i++);//Содержательные
cputs(strwnd[2]);                       //Нижнюю рамочную
window(2,WNDYBEG+1,STR_LEN-1,WNDYBEG+ROW_COUNT+1);
}
unsigned index=0,offs=0,temp; /*index+offs - индекс в массиве структур задач task; считаем, что весь список задач не может поместиться по высоте окна и при "наезде" на его нижнюю границу будет наращиваться offs, если список не исчерпан */


     /*Чтобы без проблем отслеживать одновременно индекс задачи, положение курсора и номер рабочей строки? мы свяжем их с помощью директив макроподстановки */

#define y (1+index%ROW_COUNT)
#define row (index%ROW_COUNT)
#define x 1

//функция вывода массива строк
int strput()
{
int i,key;
int string_number; //для номера выбранной строки
//Необходимо подсчитать общее количество строк в массиве задач
for(i=0;task[i].str!=NULL;i++);count=i;
//Выводим массив строк меню
for(index=0;index<count && index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(task[index].str);}

index=temp=offs=0;              //Начальные позиции
//Перекрашиваем текущую строку
textattr(spec_attr);
gotoxy(x,y);cputs(space_str);gotoxy(x,y);
cputs(task[index].str);gotoxy(x,y);
textattr(norm_attr);

//Теперь двигаемся по строкам в окне
 for(;;)
 {
 key=bioskey(0); //Получаем двухбайтовый код нажатой клавиши
 switch(key){ //Анализируем его сравнением с константами
```

```
      case ESC:return -1; //Клавиша ESC вызывает выход

      case UP:       //Клавиша стрелка вверх
//Если строка 0 и двигаться вверх некуда, но индекс массива не 0
if((index+offs)>0 && row==0 && offs>0)
{
//Вернем окраску текущей строке
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
offs--;
gotoxy(x,y);
temp=index;
//Выводим - вначале строки пробелов для очистки прошлого
for(index=0;index<ROW_COUNT;index++)
{gotoxy(x,y); cputs(space_str);}

//а затем строки
for(index=0;(index+offs)<count && index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(task[index+offs].str);}

//Перекрасим строку без изменения позиции курсора
index=temp;gotoxy(x,y);textattr(spec_attr);
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
textattr(norm_attr);
}
else if((index+offs)>0){ //В средней части окна и списка
//Вернем окраску текущей строке
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
index--;gotoxy(x,y);
//Выделим цветом новую строку
textattr(spec_attr);
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
textattr(norm_attr);
}
break;
         case DOWN:
//Если достигнута нижняя граница окна,
//но список вывода не исчерпан
if((index+offs)<count-1 && row==(ROW_COUNT-1))
{
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
offs++;
gotoxy(x,y);
textattr(spec_attr);
cputs(space_str);gotoxy(x,y);
```

```
cputs(task[index+offs].str);gotoxy(x,y);
textattr(norm_attr);

//Выводим -вначале строки пробелов для очистки прошлого
temp=index;
for(index=0;index<ROW_COUNT;index++)
{gotoxy(x,y); cputs(space_str);}

//а затем массив строк
for(index=0;(index+offs)<count && index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(task[index+offs].str);}

//Перекрасим строку без изменения позиции курсора
index=temp;gotoxy(x,y);
textattr(spec_attr);
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
textattr(norm_attr);
}
else if((index+offs+1)<count && (ROW_COUNT+row+1)<count){
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
index++;gotoxy(x,y);
textattr(spec_attr);
cputs(space_str);
gotoxy(x,y);
cputs(task[index+offs].str);
gotoxy(x,y);
textattr(norm_attr);
}
break;
          case HOME:
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
if(offs>0) {
offs=0;
//Выводим -вначале строки пробелов для очистки прошлого
for(index=0;index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(space_str);}
//а затем массив str
for(index=0;index<count && index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(task[index+offs].str);}
}
index=0;
//Перекрашиваем текущую строку
textattr(spec_attr);
gotoxy(x,y);cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
textattr(norm_attr);
```

```
break;//case HOME
          case END:
cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
if(offs>0 || (offs==0 && count>ROW_COUNT))
{offs=count-ROW_COUNT;
//Выводим -вначале строки пробелов для очистки прошлого
for(index=0;index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(space_str);}
//а затем массив str
for(index=0;index<count && index<ROW_COUNT;index++)
{gotoxy(x,y);cputs(task[index+offs].str);}
index=ROW_COUNT-1;
}
if(count<=ROW_COUNT) index=count-1;
//Перекрашиваем текущую строку
textattr(spec_attr);
gotoxy(x,y);cputs(space_str);gotoxy(x,y);
cputs(task[index+offs].str);gotoxy(x,y);
textattr(norm_attr);
break;
          case ENTER:
//Записываем номер текущей строки в string_number и выходим
string_number=index+offs;
return string_number;
          default : break;
}//switch
}
}

//Главная функция для этого варианта имеет вид:
void main(int c,char* w[])
{
if(c<2)
{cprintf("\r\nНет имени файла в командной строке");return;}
char ch;  //Для приема символа без отображения
char s[10]; //Массив для промежуточного приема строки
unsigned r; //Для размера массива
int nf;   //Для номера функции
//Можно начинать диалог с пользователем
for(;;)
{window(1,1,80,25);textattr(0x07);clrscr();
cprintf("Введите размер массива (0 - для выхода) и Enter: ");
//Принимаем размер массива в s посимвольно до Enter
for(int i=0,ch=getch();ch!=Enter;)
{
if(ch>='0' && ch <='9') //Если принятый символ цифра
{s[i]=ch;               //Заносим его в строку s
```

```
putch(ch);                  //Выводим его на экран
i++;                      //Наращиваем счетчик символов
}
ch=getch();               //Читаем следующий
}
s[i]=0;                   //Закрываем строку нулем
r=atoi(s);                  //Преобразуем в целое
if(!r) break; //При 0-м размере прекращаем работу
fflush(stdin);     //Очищаем буфер ввода (клавиатурный)
str_wnd();     //Шаблон окна для выбора задач
nf=strput();  //Получение выбранной строки
if(nf<0) break; //При отриц. номере прекращаем работу
fflush(stdin);     //Очищаем буфер ввода (клавиатурный)
if(nf==0)           //Временный вариант -
                  // если это уже решенная задача 0
task[nf].func(w[1],r);//Вызываем функцию обработки func0
else task[1].func(w[1],r);//Иначе вызываем "заглушку" func1
getch();
}
}
```

2.3. Выбор файла из списка файлов.

Аналогичную списку функций услугу мы можем оказать пользователю, предоставив ему такой же метод выбора файла - для этого придется составить функцию, осуществляющую навигацию по каталогам и файлам в оконной таблице с помощью клавиш управления курсором. Количество файлов в каталоге может значительно превышать количество задач, но их имена короче - 12 символов максимум вместе с разделяющей имя и расширение точкой. Поэтому рационально сделать окно многоколонковым - мы сделаем его на 3 колонки, это как раз позволит занять половину экрана. Для перемещения по колонкам нам понадобятся горизонтальные клавиши-стрелки и мы добавим определение их кодов соответствующими константами LEFT, RIGHT. Имена остальных констант, имеющих тот же смысл, что и в предыдущей задаче, мы просто несколько изменим, чтобы избежать путаницы.

```
const
LEFT=19200,RIGHT=19712,
COL_COUNT=3,ROW_CNT=12,MAX_EL=500,NAME_LEN=14;

char* okno[]={
{"╔═══════════╦═══════════════════════╦═══════════════════════╗",
 "║           ║                       ║                       ║",
 "╚═══════════╩═══════════════════════╩═══════════════════════╝"
```

```
};
unsigned char *space="                    ";
char fpath_name[128]; //для имени выбранного файла


//Вывод окна для списка файлов
void file_wnd()
{
window(1,10,40,25);textattr(norm_attr);
cputs(*okno);
for(int i=0;i<ROW_CNT;cputs(okno[1]),i++);
cputs(okno[2]);gotoxy(2,2);
}


char files[MAX_EL][NAME_LEN]; //Для списка элементов каталога
struct ffblk ffblk;     //Структура для параметров файла
char s[80];             //Для названия текущего каталога
int dcount;         //Для количества элементов в каталоге

/*Нам придется отслеживать при перемещении по элементам в окне
-текущий индекс в массиве files;
-текущую строку и колонку в окне
-позицию курсора
     Поэтому, как и в предыдущей задаче, мы используем макроподстановку. */

unsigned findex=0,foffs=0,ftemp; //findex+foffs -индекс в files
#define fx (2+(NAME_LEN-1)*(findex/ROW_CNT))
#define fy (2+findex%ROW_CNT)
#define col findex/ROW_CNT
#define frow (findex%ROW_CNT)

     //функция вывода содержимого текущей директории
void read_dir()
{
int done,k,i,key;
for(;;){
int chd=0;              //признак смены текущего каталога
gotoxy(1,1);cputs(*okno);//Восстанавливаем верхнюю рамку
     //определяем тек. директорию в s и выводим в рамку
getcwd(s,MAXPATH);
strcat(s,"\\");gotoxy(15,1); cputs(s);

// занесем каталоги текущего каталога в files
```

```
for(done=findfirst("*",&ffblk,FA_DIREC|FA_RDONLY),
findex=0;  !done;  strcpy(files[findex],ffblk.ff_name),
files[findex][NAME_LEN-1]=ffblk.ff_attrib,    findex++,
done = findnext(&ffblk));

// занесем файлы текущего каталога в files
for(done=findfirst("*.*",&ffblk,0);!done;findex++)
{
//Попутно имена файлов - в нижний регистр
for(i=0;i<strlen(ffblk.ff_name);i++)
ffblk.ff_name[i]=tolower(ffblk.ff_name[i]);
strcpy(files[findex+foffs],ffblk.ff_name);
files[findex+foffs][NAME_LEN-1]=ffblk.ff_attrib;
done = findnext(&ffblk);
}
dcount=findex; //Общее количество элементов в каталоге
//Остальное обнуляем
for(;findex<MAX_EL;findex++) files[findex][0]=0;

//Для удобства использования сортируем по именам
char tmp[NAME_LEN];
for(i=0;i<dcount-1;i++)
for(k=i+1;k<dcount;k++)
if(strcmp(files[i],files[k])>0)
{strcpy(tmp,files[i]);
strcpy(files[i],files[k]);
strcpy(files[k],tmp);}

//Выводим -вначале строки пробелов для очистки прошлого
for(findex=0;findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(space);}
//а затем массив files
for(findex=0;findex<dcount &&
findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(files[findex]);}

findex=ftemp=foffs=0;       //Начальные позиции
//Перекрашиваем текущую строку
textattr(spec_attr);
gotoxy(fx,fy);cputs(space);gotoxy(fx,fy);
cputs(files[findex]);gotoxy(fx,fy);
textattr(norm_attr);

//Теперь двигаемся по именам в окне
 for(;;)
 {
 key=bioskey(0);
 switch(key){
          case ESC:return;
```

```
          case LEFT:
if(col>0)
{//Вернем окраску текущей строке
cputs(space);gotoxy(fx,fy);cputs(files[findex]);gotoxy(fx,fy);

//Переместимся в колонку левее
 findex-=ROW_CNT;gotoxy(fx,fy);

 //Перекрашиваем новую строку
textattr(spec_attr);
cputs(space);gotoxy(fx,fy);cputs(files[findex]);gotoxy(fx,fy);
textattr(norm_attr);
}//if case LEFT
break;

          case RIGHT:
if(col<2 && ((col+1)*ROW_CNT+frow)<dcount)
{//Вернем окраску текущей строке
cputs(space);gotoxy(fx,fy);cputs(files[findex]);gotoxy(fx,fy);
//Переместимся в колонку правее
findex+=ROW_CNT;gotoxy(fx,fy);
//Перекрашиваем новую строку
textattr(spec_attr);
cputs(space);gotoxy(fx,fy);
cputs(files[findex]);gotoxy(fx,fy);
textattr(norm_attr);}//if case RIGHT
break;
          case UP:

if((findex+foffs)>0 && col==0 && frow==0 && foffs>0)
{
//Вернем окраску текущей строке
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
foffs--;
gotoxy(fx,fy);
ftemp=findex;
//Выводим - вначале строки пробелов для очистки прошлого
for(findex=0;findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy); cputs(space);}

//а затем массив files
for(findex=0;(findex+foffs)<dcount &&
findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(files[findex+foffs]);}

//Перекрасим строку без изменения позиции курсора
findex=ftemp;gotoxy(fx,fy);textattr(spec_attr);
```

```
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);
}
else if((findex+foffs)>0){
//Вернем окраску текущей строке
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
findex--;gotoxy(fx,fy);
textattr(spec_attr);
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);
}
break;
          case DOWN:
if((findex+foffs)<dcount-1 && col==2 && frow==(ROW_CNT-1))
{
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
foffs++;
gotoxy(fx,fy);
textattr(spec_attr);
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);

//Выводим - вначале строки пробелов для очистки прошлого
ftemp=findex;
for(findex=0;findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy); cputs(space);}

//а затем массив files
for(findex=0;(findex+foffs)<dcount &&
findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(files[findex+foffs]);}

//Перекрасим строку без изменения позиции курсора
findex=ftemp;gotoxy(fx,fy);
textattr(spec_attr);
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);
}
else if((findex+foffs+1)<dcount &&
(col*ROW_CNT+frow+1)<dcount){
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
findex++;gotoxy(fx,fy);
textattr(spec_attr);
cputs(space);gotoxy(fx,fy);
```

```
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);
}
break;
          case HOME:
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
if(foffs>0) {
foffs=0;
//Выводим - вначале строки пробелов для очистки прошлого
for(findex=0;findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(space);}
//а затем массив files
for(findex=0;findex<dcount &&
findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(files[findex+foffs]);}
}
findex=0;
//Перекрашиваем текущую строку
textattr(spec_attr);
gotoxy(fx,fy);cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);

break;//case HOME
          case END:
cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
if(foffs>0 || (foffs==0 && dcount>COL_COUNT*ROW_CNT))
{foffs=dcount-COL_COUNT*ROW_CNT;
//Выводим -вначале строки пробелов для очистки прошлого
for(findex=0;findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(space);}
//а затем массив files
for(findex=0;findex<dcount &&
findex<COL_COUNT*ROW_CNT;findex++)
{gotoxy(fx,fy);cputs(files[findex+foffs]);}
findex=COL_COUNT*ROW_CNT-1;
}
if(dcount<=COL_COUNT*ROW_CNT) findex=dcount-1;
//Перекрашиваем текущую строку
textattr(spec_attr);
gotoxy(fx,fy);cputs(space);gotoxy(fx,fy);
cputs(files[findex+foffs]);gotoxy(fx,fy);
textattr(norm_attr);
break;
          case ENTER:
//Если стоим не на имени каталога то считаем файлом
//и копируем его в fpath_name
if(!(files[findex+foffs][13] & FA_DIREC))
```

```
{strcpy(fpath_name,files[findex+foffs]);
//и выходим
return;}
//иначе делаем каталог текущим
if(!strcmp(files[findex+foffs],".")) chdir("\\");
else chdir(files[findex+foffs]);
chd=1;break;
          default : break;
}//switch
if(chd) break;
}
}//for(;;)
}
```

При выборе имени файла из предоставляемого списка функция `main` может выглядеть так:

```
     void main()
{
char ch;  //Для приема символа без отображения
char s[10]; //Массив для промежуточного приема строки
unsigned r; //Для размера массива
int nf;   //Для номера функции
//Можно начинать диалог с пользователем
for(;;)
{window(1,1,80,25);textattr(0x07);clrscr();
cprintf("Введите размер массива (0 - для выхода) и Enter: ");
//Принимаем размер массива в s посимвольно до Enter
for(int i=0,ch=getch();ch!=Enter;)
{
if(ch>='0' && ch <='9') //Если принятый символ цифра
{s[i]=ch;                //Заносим его в строку s
putch(ch);               //Выводим его на экран
i++;                  //Наращиваем счетчик символов
}
ch=getch();           //Читаем следующий
}
s[i]=0;               //Закрываем строку нулем
r=atoi(s);                //Преобразуем в целое
if(!r) break; //При 0-м размере прекращаем работу
fflush(stdin);    //Очищаем буфер ввода (клавиатурный)
str_wnd();    //Шаблон окна для выбора задач
nf=strput();  //Получение выбранной строки
if(nf<0) break; //При отриц. номере прекращаем работу
fflush(stdin);    //Очищаем буфер ввода (клавиатурный)

file_wnd();//рисуем окно для вывода файлов
read_dir();//Выводим текущий каталог в окно
```

```
if(nf==0 || nf==15) //Временный вариант
task[nf].func(fpath_name,r); //Вызываем функцию обработки func0
else task[2].func(fpath_name,r);//Иначе вызываем "заглушку"
getch();
}
}

//Подпрограмма - заглушка
void func2()
{window(42,15,80,18);clrscr();
cprintf("Эту задачу вы должны программировать сами");}
```

2.4. Стандартный набор функций для работы с векторами.

Мы приведем определения наиболее употребительных общего характера функций для работы с одномерными массивами (векторами). Вы можете записать их в один файл, например, fvector.def, включать этот файл в свои программы директивой

```
#include "fvector.def"
```

и вызывать эти функции на выполнение с соответствующими значениями параметров.

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

//Определения наиболее общих функций работы с векторами.

/*Эта функция пытается создать пустой вектор заданной размерности, попутно очищая его */
double *CreateEmptyVector(long n)
{
 if(n<=0)//при некорректном размере вектора
 {
 printf("%d не может быть размерностью вектора \n",n);
 return 0;//возвращаем нулевой указатель
 }
 //пытаемся выделить память под запрашиваемое число
 //элементов двойной точности
 double *vec=(double *)farcalloc(n,sizeof(double));
 if(vec==NULL)//если не удалось выделить запрошенный блок
 {
  printf("Не хватает памяти под %d элементов вектора типа double\n",n);
 return vec;//в этом случае тоже возвращаем нулевой указатель
 }
 for(long i=0;i<n;i++)
```

```
 vec[i]=0;//обнуляем n составляющих вектора
 return vec;          //и возвращаем указатель на него
}

/*Эта функция пытается создать пустой вектор заданной размерности, а затем заполнить его данными из другого вектора */
double *CreateFilledVector(long n, double *whatcopy)
{
 double *vec=CreateEmptyVector(n);
 if(vec==NULL)//если не удалось выделить запрошенный блок
 return vec;//возвращаем нулевой указатель
 for(long i=0;i<n;i++)
 vec[i]=whatcopy[i];//копируем n составляющих вектора
 return vec;          //и возвращаем указатель на него
}

//эта функция пытается загрузить вектор, лежащий в файле
//в виде v1 v2 ... vn, где vi - его составляющие
double *LoadVector(long n, char *f)//имя файла с вектором
{
 double *vec=CreateEmptyVector(n);
 if(vec==NULL)//если не удалось выделить запрошенный блок
 return vec;//возвращаем нулевой указатель
 FILE *fp=fopen(f,"r");//пытаемся открыть файл
 if(!fp)//если не удалось открыть файл
 {
 printf("Не могу открыть файл %s\n",f);
 farfree(vec);
 return NULL;//возвращаем NULL-вектор
 }
 for(long i=0;i<n;i++)
 fscanf(fp,"%lf",vec+i);//считываем n составляющих вектора
 fclose(fp);//закрываем ненужный больше файл
 return vec;//возвращаем указатель на считанный вектор
}

//эта функция записывает вектор в файл с заданным именем
void WriteVector(long n, double *vec, char *f)
{
 if(n<=0)//при некорректном размере вектора
 {
 printf("%d не может быть размерностью вектора \n",n);
 return;
 }
 FILE *fp=fopen(f,"w");//пытаемся открыть файл
 if(!fp)//если не удалось открыть файл
 {
```

```
 printf("Не могу открыть файл %s\n",f);
 return;
 }
 for(long i=0;i<n;i++)
 fprintf(fp,"\t%g",vec[i]);//записываем n составляющих вектора
 fclose(fp);//закрываем ненужный больше файл
}

//Печать вектора на экран - это не что иное, как запись
//его в файл, соответствующий экрану
void PrintVector(long n, double *vec)
{
 WriteVector(n,vec,"con");
}
```

6.16.2. Обработка двумерных числовых массивов (таблиц, или матриц, или массивов векторов).

Общие сведения о задачах этого класса в 4.9.12.

При выделении памяти в куче работа с элементами матрицы может вестись как с элементами одномерного массива с вычисляемым индексом - доступ к j-му элементу i-й строки, очевидно, возможен в этом случае только так : `matrix[i*M+j]=8.23;` где M - длина строки - в отличие от статического 2-мерного массива, размещенного объявлением вида

```
double matrix1[N][M];
```

доступ к элементам которого возможен по 2-м индексам записью `matrix1[i][j]=c;` так как компилятор знает оба размера матрицы.

Если вам более по душе такое же обращение к элементам матрицы, размещенной в куче, то выделение памяти придется осуществлять в 2 этапа: сначала выделить память под массив указателей на строки матрицы, а затем в цикле - под сами строки и присвоить адреса выделенных под каждую строку участков памяти указателям из массива указателей на строки. Выглядит это так (если количество строк равно например N, число столбцов M, а тип элементов double):

```
double** matr=(double**)malloc(N*sizeof(void*));
if(matr==NULL){printf("Нет памяти в куче");exit(3);}
for(unsigned i=0;i<N;i++)
{matr[i]=(double*)malloc(M*sizeof(double));
if(matr==NULL){
for(unsigned j=i-1;j>=0;j--)
//освобождаем память из-под всех предыдущих строк
farfree(mtr[j]);
```

```
    printf("Нет памяти в куче");exit(3);}
    }
```

Освобождение арендованной на время памяти осуществляется в обратном порядке:

```
    for(unsigned i=0;i<N;i++) free(matr[i]);
    free(matr);
```

При таком выделении памяти вы можете индексировать матричные элементы в той же форме, что и для статических матриц:

```
    matr[i][j]=6.345;
```

Теперь мы приведем ряд функций общего характера по конструированию матриц в оперативной памяти при различных исходных данных. Эти функции вы можете собрать в одном файле и затем компилировать и компоновать вашу программу с этим файлом, присоединив его, например, через файл проекта.

1. Функция создания в памяти "пустой" матрицы заданной размерности, с попутным обнулением её элементов.

```
double **CreateEmptyMatrix(long m, long n)
{
  if(m<=0||n<=0)//при некорректном размере матрицы
  {
    printf("%dx%d не может быть размерностью матрицы \n",m,n);
   return 0;//возвращаем нулевой указатель
  }

  //пытаемся выделить память под m строк матрицы -
  //указателей на вещественные числа двойной точности
  double **mtr=(double **)farmalloc(m*sizeof(double*));
if(mtr==NULL)//если не удалось выделить запрошенный блок
{
    printf("Не хватает памяти под %d строк матрицы типа
    double*\n",m);
    return mtr;
}//в этом случае тоже возвращаем нулевой указатель
    //выделяем память под каждую строку матрицы
 for(long i=0;i<m;i++)
  if(!(mtr[i]=(double *)farmalloc(n*sizeof(double))))
    //если не удалось выделить память под очередную строку
    {printf("Не хватает памяти под %d строку матрицы \n",i);
    for(long j=i-1;j>=0;j--)
    //освобождаем память из-под всех предыдущих строк
    farfree(mtr[j]);
    //освобождаем память из-под массива указателей на строки
    farfree(mtr);
```

```
     return 0;//возвращаем нулевой указатель
     }
      for(i=0;i<m;i++)
      for(long j=0;j<n;j++)
      mtr[i][j]=0;  //обнуляем mxn составляющих матрицы
return mtr;         //и возвращаем указатель на неё
}
```

2. Эта функция пытается создать пустую матрицу заданной размерности, а затем заполнить её данными из другой матрицы

```
double **CreateFilledMatrix(long m, long n, double **whatcopy)
{
double **mtr=CreateEmptyMatrix(m,n);
if(mtr==NULL)//если не удалось выделить запрошенный блок
return mtr;//возвращаем нулевой указатель
for(long i=0;i<m;i++)
for(long j=0;j<n;j++)
mtr[i][j]=whatcopy[i][j];//копируем mxn составляющих матрицы
return mtr;             //и возвращаем указатель на неё
}
```

3. Эта функция пытается загрузить матрицу, лежащую в файле в виде m11 m12... m1n m21... mmn, где mij – её составляющие

```
double **LoadMatrix(long m, long n, char *f)
//char *f - имя файла с матрицей
{
double **mtr=CreateEmptyMatrix(m,n);
if(mtr==NULL)//если не удалось выделить запрошенный блок
return mtr;//возвращаем нулевой указатель
FILE *fp=fopen(f,"r");//пытаемся открыть файл
if(!fp)//если не удалось открыть файл
{
printf("Не могу открыть файл %s\n",f);
ReleaseMatrix(m,n,mtr);
return NULL;//возвращаем NULL-указатель
}
for(long i=0;i<m;i++)
for(long j=0;j<n;j++)
{
     double *dptr=mtr[i]+j;
     //считываем mxn составляющих матрицы
     fscanf(fp,"%lf",dptr);
}
fclose(fp);//закрываем ненужный больше файл
```

```
return mtr;//возвращаем указатель на считанную матрицу
}
```

4. Эта функция записывает матрицу в файл с заданным именем

```
void WriteMatrix(long m, long n, double **mtr, char *f)
{
if(m<=0||n<=0)//при некорректном размере матрицы
{
printf("%dx%d не может быть размерностью матрицы \n",m,n);
return;
}
FILE *fp=fopen(f,"w");//пытаемся открыть файл
if(!fp)//если не удалось открыть файл
{
printf("Не могу открыть файл %s\n",f);
return;
}
for(long i=0;i<m;i++)
{
for(long j=0;j<n;j++)
    //записываем mxn составляющих матрицы
     fprintf(fp,"\t%g",*(mtr[i]+j));
     fprintf(fp,"\n");//добавляем в конце каждой строки
}
fclose(fp);//закрываем ненужный больше файл
}
```

5. Вывод матрицы на экран - это не что иное, как запись её в файл, соответствующий экрану

```
void PrintMatrix(long m, long n, double **mtr)
{
  WriteMatrix(m,n,mtr,"con");
}
```

6. Для освобождения памяти из-под матрицы, представляющей собой массив указателей, мы должны вначале освободить память из-под каждой строки с указателем на тип double, и лишь затем - из-под указателя на double*

```
void ReleaseMatrix(long m, long /*n*/, double **mtr)
{
  for(long i=0;i<m;i++)
    farfree(mtr[i]);
  farfree(mtr);
}
```

Теперь мы приведем серию примеров прикладных программ, использующих вышеприведенные подпрограммы создания и разрушения матриц в оперативной памяти. Будет целесообразно прототипы подпрограмм общего пользования собрать в заголовочном файле, например cmatrix.h и включать его во все программы:

Файл cmatrix.h

```
#ifndef __CMATRIX_H
#define __CMATRIX_H

double **CreateEmptyMatrix(long m, long n);
double **CreateFilledMatrix(long m, long n, double
**whatcopy);
double **LoadMatrix(long m, long n, char *f);
void WriteMatrix(long m, long n, double **mtr, char *f);
void PrintMatrix(long m, long n, double **mtr);
void ReleaseMatrix(long m, long n, double **mtr);

#endif
```

1. Программа сложения двух матриц

Необходимые сведения в 4.9.12.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include "cmatrix.h"
void main(int argc, char *argv[])
{
 //при недостаточном количестве аргументов в командной строке
 if(argc<6)
 {
  printf("Используйте : %s m n file1.txt file2.txt fileres.txt,\n\trge mxn -
размерность матриц, лежащих в указанных файлах \n",*argv);
return;}//выход из программы при недостатке параметров
     long m=atol(argv[1]),
     n=atol(*(argv+2));//определяем размерность матриц
     if(m<=0||n<=0)//при некорректном размере матрицы
     {
 printf("%dx%d не может быть размерностью матрицы \n",m,n);
 return;//выходим в ОС
      }
     //загружаем первую матрицу
 double **mtr1=LoadMatrix(m,n,argv[3]);
 if(mtr1==NULL)//если не удалось загрузить, выходим
 exit(0);
     //загружаем вторую матрицу
 double **mtr2=LoadMatrix(m,n,*(argv+4));
```

```
if(mtr2==0)//если не удалось загрузить
{
     //освобождаем память из под первой матрицы и выходим
ReleaseMatrix(m,n,mtr1); return;
}
double **mtr3=CreateEmptyMatrix(m,n);
if(!mtr3)//если не хватило памяти
{ //освобождаем память из под первой матрицы
ReleaseMatrix(m,n,mtr1);
//освобождаем память из под второй матрицы
ReleaseMatrix(m,n,mtr2);
return;
}
printf("Первая матрица: \n");
PrintMatrix(m,n,mtr1);
printf("Вторая матрица: \n");
PrintMatrix(m,n,mtr2);
printf("Сумма матриц: \n");
for(long i=0;i<m;i++)
for(long j=0;j<n;j++)
     //поэлементно суммируем матрицы
     mtr3[i][j]=(*(mtr1+i))[j]+*(j+mtr2[i]);
PrintMatrix(m,n,mtr3);
//записываем матрицу-результат в соответствующий файл
WriteMatrix(m,n,mtr3,argv[5]);
ReleaseMatrix(m,n,mtr1);//освобождаем память из под матриц
ReleaseMatrix(m,n,mtr2);
ReleaseMatrix(m,n,mtr3);
}
```

Абсолютно аналогичной будет программа вычитания - меняется только знак.

2.Программа умножения двух матриц

Необходимые сведения в 4.9.12.2.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include "cmatrix.h"

void main(int argc, char *argv[])
{
 //при недостаточном количестве аргументов в командной строке
 if(argc<8)
 {
printf("Используйте : %s m1 n1 file1.txt m2 n2 file2.txt fileres.txt,\n\tгде mxn - размерность матриц, лежащих в указанных файлах \n",*argv);
return;//выход из программы при недостатке параметров
```

```
}
long m1=atol(argv[1]),
n1=atol(*(argv+2)),//определяем размерность первой матрицы
m2=atol(argv[4]),
n2=atol(*(argv+5));//определяем размерность второй матрицы
if(m1<=0||n1<=0||m2<=0||n2<=0)//при некорректном размере
{
printf("Некорректная размерность матрицы \n");
return;//выходим в ОС
}
if(n1!=m2)
{
printf("Условие умножения матриц - равенство количества \
строк второй матрицы"
" количеству столбцов первой - не выполняется \n");
exit(0);
}
    //загружаем первую матрицу
double **mtr1=LoadMatrix(m1,n1,argv[3]);
if(mtr1==NULL)//если не удалось загрузить, выходим
exit(0);
    //загружаем вторую матрицу
double **mtr2=LoadMatrix(m2,n2,*(argv+6));
if(mtr2==0)//если не удалось загрузить
{
    //освобождаем память из под первой матрицы
ReleaseMatrix(m1,n1,mtr1);
return;                    //и выходим
}
double **mtr3=CreateEmptyMatrix(m1,n2);
if(!mtr3)//если не хватило памяти
{
ReleaseMatrix(m1,n1,mtr1);//освобождаем память
ReleaseMatrix(m2,n2,mtr2);
return;
}
printf("Первая матрица: \n");
PrintMatrix(m1,n1,mtr1);
printf("Вторая матрица: \n");
PrintMatrix(m2,n2,mtr2);
for(long j=0;j<n2;j++)
for(long i=0;i<m1;i++)
    for(long k=0;k<n1;k++)
    mtr3[i][j]+=mtr1[i][k]*mtr2[k][j];
printf("Произведение матриц: \n");
PrintMatrix(m1,n2,mtr3);
//записываем матрицу-результат в соответствующий файл
```

```
WriteMatrix(m1,n2,mtr3,argv[7]);
ReleaseMatrix(m1,n1,mtr1);//освобождаем память из под матриц
ReleaseMatrix(m2,n2,mtr2);
ReleaseMatrix(m1,n2,mtr3);
}
```

3. Операции над одной матрицей, традиционные для массивов: поиск минимума и максимума и их координат, а также абсолютного минимума (максимума).

```
#include <stdio.h>
#include <math.h>
#include <alloc.h>
#include "cmatrix.h"

int main()
{
double min,max,amin,amax;
long xmin,xmax,xamin,xamax,ymin,ymax,yamin,yamax;

//Для разнообразия введём элементы матрицы с клавиатуры
long m,n;
scanf("%li%li",&m,&n);//определяем размерность матрицы
if(m<=0||n<=0)//при некорректном размере матрицы
{
printf("%dx%d не может быть размерностью матрицы \n",m,n);
return 0;//выходим в ОС
}
double **mtr=LoadMatrix(m,n,"con");//вводим матрицу с консоли
if(!mtr)//если не удалось загрузить, выходим
return 1;
printf("\nВведённая матрица: \n");
PrintMatrix(m,n,mtr);
```

/*устанавливаем значения минимумов и максимумов (обычных и абсолютных [т.е. по модулю]) в значение первого элемента вектора векторов – матрицы действительных чисел, а их соответствующие координаты в его номер, т.е. 0*/

```
min=max=mtr[0][0];
amin=amax=fabs(mtr[0][0]);
xmin=xmax=xamin=xamax=ymin=ymax=yamin=yamax=0;
for(long i=0;i<m;i++)//в циклах по элементам матрицы
for(long j=0;j<n;j++)
{
     if(mtr[i][j]<min)
     {
     min=mtr[i][j];
     xmin=j;
```

```
      ymin=i;
      }
      if(*(mtr[i]+j)>max)
      max=mtr[i][j],
      xmax=j,ymax=i;
      if(fabs(*(*(mtr+i)+j))<amin)
      {
      amin=fabs(mtr[i][j]);
      xamin=j;
      yamin=i;
      }
      if(fabs((*(mtr+i))[j])>amax)
      amax=fabs(mtr[i][j]), xamax=j, yamax=i;
}

printf("\nmin(mtr) =
mtr[%ld][%ld]=%g\nmax(mtr)=mtr[%ld][%ld]=%g\n"
     "min(|mtr|)=|mtr[%ld][%ld]|=%g\nmax(|mtr|) =
     |mtr[%ld][%ld]|=%g\n",
ymin,xmin,min,ymax,xmax,max,yamin,xamin,amin,yamax,xamax,amax);
ReleaseMatrix(m,n,mtr);//освобождаем память из-под матрицы
return -1;//вернём -1 как код завершения программы
}
```

4. Программа решения системы линейных алгебраических методом Гаусса

Изложение алгоритма в 4.9.12.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include "cmatrix.h"
#include "cvector.h"

      void main(int argc, char *argv[])
      {
 long i,j,k,num;
 //при недостаточном количестве аргументов в командной строке
 if(argc-5)
 {
printf("Используйте : %s m n file.txt fileres.txt,\n"
"\tгде mxn - размерность матриц, лежащих в указанных файлах \n",
argv[0]);
return;//выход из программы при недостатке параметров
 }
 long m=atol(argv[1]),
      n=atol(*(argv+2));//определяем размерность матрицы
 if(m<=0||n<=0)//при некорректном размере матрицы
 {
```

```
 printf("%dx%d не может быть размерностью матрицы \n",m,n);
 return;//выходим в ОС
 }
 if(n-m!=1)
 {
printf("Количество неизвестных отличается от \
количества уравнений !\n");
exit(-1);
 }
 double **mtr=LoadMatrix(m,n,argv[3]);//загружаем матрицу
 if(mtr==NULL)//если не удалось загрузить, выходим
 exit(0);
/*Результатом решения системы уравнений является вектор неизвестных */

 double *res=CreateEmptyVector(m);
 if(!res)//если не хватило памяти
 {
 ReleaseMatrix(m,n,mtr);//освобождаем память из под матрицы
 return;  //выходим
 }
 printf("Исходная матрица СЛАУ: \n");
 PrintMatrix(m,n,mtr);
 printf("\nВектор решений: \n");
 //Выбор главного элемента
 for(i=0;i<m;i++)//col
 {
 double max=mtr[0][i];
 for(j=i,num=i;j<m;j++) //row
      if(fabs(mtr[j][i])>max)
      max=fabs(mtr[j][i]),num=j;
 if(num!=i)
      //Обмен num-той и i-той строк местами
      for(k=0;k<n;k++)
      {
      double temp=mtr[num][k];
      mtr[num][k]=mtr[i][k];
      mtr[i][k]=temp;
      }
 }
 for(i=0;i<m;i++)
 if(!mtr[i][i])
 {
printf("Решение системы при вырожденной матрице невозможно \n");
      ReleaseMatrix(m,n,mtr);
      farfree(res);
      exit(0);
 }
```

```
//Прямой ход Гаусса
for(i=0;i<m;i++)//
{
double sw=mtr[i][i];
for(j=0;j<n;j++)
     mtr[i][j]/=sw;
for(k=i+1;k<m;k++)
{
     double c=mtr[k][i];
     for(j=0;j<n;j++)
     mtr[k][j]-=mtr[i][j]*c;
}
}
//Обратный ход Гаусса
for(i=m-2;i>=0;i--)//row
{
double s=0;
for(j=i+1;j<m;j++) //col
     s+=mtr[i][j]*mtr[j][n-1];
mtr[i][n-1]-=s;
}
//Переписываем из последнего столбца вектор-результат
for(i=0;i<m;i++)
res[i]=mtr[i][n-1];
PrintVector(m,res);
//записываем вектор-результат в соответствующий файл
WriteVector(m,res,argv[4]);
ReleaseMatrix(m,n,mtr);
farfree(res);
}
```

5. Программа нахождения детерминанта матрицы с помощью метода Гаусса. Алгоритм решения в 4.9.12.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include "cmatrix.h"

void main(int argc, char *argv[])
{
 double sw,c,det,max;
 long i,j,k,how,num;
 //при недостаточном количестве аргументов в командной строке
 if(argc<4)
 {
printf("Используйте : %s m n file.txt,\n"
"\tгде mxn - размерность матрицы, лежащей в указанном файле \n",
argv[0]);
return;//выход из программы при недостатке параметров
```

```
 }
 long m=atol(argv[1]),
      n=atol(*(argv+2));//определяем размерность матрицы
 if(m<=0||n<=0||m!=n)//при некорректном размере матрицы
 {
printf("%dx%d не может быть размерностью квадратной \
матрицы \n",m,n);
return;//выходим в ОС
 }
 double **mtr=LoadMatrix(m,n,argv[3]);//загружаем матрицу
 if(mtr==NULL)//если не удалось загрузить, выходим
 exit(0);
 printf("Исходная матрица: \n");
 PrintMatrix(m,n,mtr);
 switch(m)
 {
 case 1:
      det=mtr[0][0];
      break;
 case 2:
      det=mtr[0][0]*mtr[1][1]-mtr[0][1]*mtr[1][0];
      break;
 default:
 //количество перестановок строк определяет знак детерминанта
      for(how=i=0;i<m;i++)//col
      {
      max=mtr[0][i];
      for(j=i,num=i;j<m;j++) //row
          if(fabs(mtr[j][i])>max)
           max=fabs(mtr[j][i]),num=j;
      if(num!=i)
      {
          for(k=0;k<m;k++)//num & i
          {
           double temp=mtr[num][k];
           mtr[num][k]=mtr[i][k];
           mtr[i][k]=temp;
          }
          how++;//наращиваем количество перестановок
      }
      }
      //Прямой ход Гаусса
      for(i=0;i<m;i++)
      {
      for(sw=mtr[i][i],j=i+1;j<m;j++)
          mtr[i][j]/=sw;
      for(k=i+1;k<m;k++)
          for(c=mtr[k][i],j=0;j<m;j++)
           mtr[k][j]-=mtr[i][j]*c;
      }
```

```
      //После прямого хода нам остаётся только перемножить
      //элементы главной диагонали
      for(det=1,i=0;i<m;i++)
      det*=mtr[i][i];
      //и учесть количество проведенных перестановок строк
      det*=powl(-1,how);
 }
 printf("\nДетерминант: %g\n",det);
 ReleaseMatrix(m,n,mtr);
}
```

6.Программа ортогонализации матрицы (построения из строк матрицы ортогональной системы векторов)

Алгоритм решения в 4.9.12.5.

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <math.h>
#include "cmatrix.h"

void main(int argc, char *argv[])
{
//при недостаточном количестве аргументов в командной строке
 if(argc<5)
 {
 printf("Используйте : %s m n file.txt fileres.txt,\n"
 "\tгде mxn - размерность матриц, лежащих в \
      указанных файлах \n",*argv);
      return;//выход из программы при недостатке параметров
 }
 long m=atol(argv[1]),
      n=atol(*(argv+2));//определяем размерность матрицы
 if(m<=0||n<=0)//при некорректном размере матрицы
 {
 printf("%dx%d не может быть размерностью матрицы\n",m,n);
 return;//выходим в ОС
 }
 double **mtr=LoadMatrix(m,n,argv[3]);//загружаем матрицу
 if(mtr==NULL)//если не удалось загрузить, выходим
 exit(0);
 double **orto=CreateFilledMatrix(m,n,mtr);
 if(!orto)//если не хватило памяти
 {
 ReleaseMatrix(m,n,mtr);//освобождаем память
 return;  //выходим
 }
 printf("Исходная матрица: \n");
 PrintMatrix(m,n,mtr);
```

```
printf("Ортонормированная матрица: \n");
//Вычисляем сумму квадратов элементов первой строки матрицы
for(double i=0,mod=0;i<n;i++)
mod+=orto[0][i]*orto[0][i];
//находим корень из неё, то есть модуль первой строки
mod=sqrt(mod);
//нормируем первую строку, деля каждый её элемент на модуль
for(i=0;i<n;i++)
orto[0][i]/=mod;
//Для надёжности повторяем процесс ортогонализации 3 раза
for(long dd=0;dd<3;dd++)
{
for(long l=1;l<m;l++)
{
     for(i=0;i<l;i++)
     {
     //Вычисляем скалярное произведение l-той строки на i-ую
     for(double p=0,s=0;s<n;s++)
         p+=orto[l][s]*orto[i][s];
     //Ортогонализируем l-тую строку к i-ой
     for(long j=0;j<n;j++)
         orto[l][j]-=p*orto[i][j];
     }
     //Нормируем l-тую строку
     for(double f=0,mod=0;f<n;f++)
     mod+=orto[l][f]*orto[l][f];
     mod=sqrt(mod);
     for(f=0;f<n;f++)
     orto[l][f]/=mod;
}
}
PrintMatrix(m,n,orto);
//После вывода на печать матрицы для очистки совести выводим
//попарно скалярные произведения всех строк, дабы убедиться,
//что построенная нами система векторов ортогональна
for(i=0;i<m;i++)
for(long j=i+1;j<m;j++)
{
     for(double p=0,s=0;s<n;s++)
     p+=orto[i][s]*orto[j][s];
     printf("orto[%li]*orto[%li]=%g\n",(long)i,j,p);
}
//записываем матрицу-результат в соответствующий файл
WriteMatrix(m,n,orto,argv[4]);
ReleaseMatrix(m,n,mtr);//освобождаем память из под матриц
ReleaseMatrix(m,n,orto);
}
}
```

7. Программа транспонирования матрицы (замены строк столбцами)

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include "cmatrix.h"

     void main(int argc, char *argv[])
     {
//при недостаточном количестве аргументов в командной строке
if(argc<5)
{
printf("Используйте : %s m n file.txt fileres.txt,\n"
"\tгде mxn - размерность матриц, лежащих в \
     указанных файлах \n",*argv);
     return;//выход из программы при недостатке параметров
}
long m=atol(argv[1]),
      n=atol(*(argv+2));//определяем размерность матрицы
if(m<=0||n<=0)//при некорректном размере матрицы
{
printf("%dx%d не может быть размерностью матрицы \n",m,n);
return;//выходим в ОС
}
double **mtr=LoadMatrix(m,n,argv[3]);//загружаем матрицу
if(mtr==NULL)//если не удалось загрузить, выходим
exit(0);
double **tran=CreateEmptyMatrix(n,m);
//размерность транспонированной матрицы
//обратна к размерности исходной
if(!tran)//если не хватило памяти
{
ReleaseMatrix(m,n,mtr);
return;
}
printf("Исходная матрица: \n");
PrintMatrix(m,n,mtr);
printf("Транспонированная матрица: \n");
for(long i=0;i<m;i++)
for(long j=0;j<n;j++)
      tran[j][i]=mtr[i][j];
PrintMatrix(n,m,tran);
//записываем матрицу-результат в соответствующий файл
WriteMatrix(n,m,tran,argv[4]);
ReleaseMatrix(m,n,mtr);//освобождаем память из под матриц
ReleaseMatrix(n,m,tran);
}
```

6.17. Работа с текстовыми строками.

Текстовые строки – это массивы однобайтовых целочисленных кодов символов и их посимвольная обработка в принципе не отличается от обработки векторов. Как и любые числовые массивы в С, строки можно сравнивать только поэлементно, для выполнения этой работы в библиотеке есть функция `strcmp()` с прототипом в заголовочном файле `string.h` – там же вы найдете прототипы многих других функций обработки строк. Некоторые мы научились программировать в самом начале настоящего курса. Библиотечный сервис для обработки строк в С очень мощный, несмотря на отсутствие абстрактного строкового типа.

Операции ввода-вывода строк могут осуществляться как посимвольно, так и строками в целом – например, с помощью функций форматированного ввода-вывода `fprintf()`, `printf()`, `cprintf()`, `fscanf()`, `scanf()`, `cscanf` с форматом преобразования `%s`.

При работе со строками надо помнить о принятом соглашении завершать строку нулевым байтом и если вы помещаете ее в массив, объявленный как, например, `char c[20]` и заполняете его посимвольно, то позаботьтесь сами о записи в него в конце нулевого значения – что-то вроде `c[i]=0;` или `c[i]='0';`

Наиболее сложной задачей в работе со строками является видимо их разбивка на слова из-за множества возможных вариантов используемых в стоках совокупностей разделителей между словами. Для выделения отдельных слов из строки текста при заданном списке разделителей в библиотеке есть функция

```
char *strtok(char *s1, const char *s2);
```

возвращающая указатель на очередное слово из строки `s1`, отделяемое от остальных разделителями, входящими в определяемую вами строку `s2`. Это функция с возможностью повторного вызова – при первом вы задаете указатель на разбираемую строку, а при всех остальных для этой же строки – нуль-указатель со значением `NULL`. Когда анализируемая строка исчерпается функция вернет `NULL`. Фрагмент программы печати слов из строки `s1` с набором принятых разделителей в `s2` может выглядеть, например, так:

```
char* s2=",;!?";
char* t=strtok(s1,s2);
for(;t!=NULL;) {printf("\n%s",t);t=strtok(NULL,s2);}
```

В настоящем разделе мы приведем 2 примера работы со строками - сортировку строк и синтаксическую разборку строки на лексемы - условные неделимые единицы некоторого языка, принятого в контексте текущей задачи. Остальные случаи будут встречаться по текстам различных программ и там по возможности комментироваться.

6.17.1. Сортировка строк.

Пусть в текстовом файле находится массив слов, разделенных друг от друга пробелами или переводами строки - это может быть например неупорядоченный алфавитно список фамилий учащихся вашей группы и необходимо вывести его на экран или в файл в упорядоченном виде. Имя файла будет задано пользователем в командной строке.

Программа может выглядеть так (а может и иначе - вам надо привыкать к тому, что программирование - творческий процесс и создать "рецептурный справочник" по написанию программ невозможно):

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

char word[40]; //Для одного самого длинного слова из файла
void main(int c, char* w[])
{//В командной строке должно быть имя файла
if(c!=2)
{cprintf("Неполадки с параметрами в командной строке");
return;};
FILE* f=fopen(w[1],"r");
if(f==NULL)
{cprintf("\r\nНеудача с открытием файла %s",w[1]);return;};

//Сосчитаем количество слов в файле
for(unsigned wc=0;!feof(f);fscanf(f,"%s",word),wc++);
rewind(f); //возвратим указатель чтения - записи в начало

//Теперь можем попросить память под указатели на слова
char** wptr=(char**) malloc(wc*sizeof(char**));
if(wptr==NULL)
{cprintf("\r\nНеудача с выделением памяти для адресов слов");
exit(255);};
```

```
/*Перепишем файл в память с попутным ее выделением для
каждого слова */
for(unsigned i=0;i<wc;i++)
{fscanf(f,"%s",word);
wptr[i]=(char*)malloc(strlen(word)+1);
if(wptr[i]==NULL)
{cprintf("\r\nНеудача с выделением памяти для слов");
exit(255);};
strcpy(wptr[i],word);
}
```

/*Только теперь мы добрались до задачи сортировки - выполним ее простейшим "пузырьковым" методом. Обратим внимание на следующее: под каждую строку мы выделяли память по ее длине, а, следовательно, менять местами содержимое самих строк мы не можем - более длинные не уместятся в более коротких. Да это и нецелесообразно с точки зрения скорости - поэтому мы просто поменяем значения указателей */

```
unsigned j;
char* t;
for(i=0;i<wc-1;i++)
for(j=i+1;j<wc;j++)
if(strcmp(wptr[i],wptr[j])>0)
{t=wptr[i];wptr[i]=wptr[j];wptr[j]=t;}

//Осталось вывести результат
clrscr();
cprintf("\r\nОтсортированные по алфавиту слова");
for(i=0;i<wc;i++) cprintf("\r\n%s",wptr[i]);
//И освободить арендованную память
for(i=0;i<wc;i++) free(wptr[i]);
free(wptr);
}
```

6.17.2. Двоичный поиск в упорядоченных массивах.

Основные сведения в 4.9.13.

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

char word[40]; //Для одного самого длинного слова из файла
```

```
/*Дополнительная функция сравнения строк, возвращающая целочисленную оценку "приближения" сравниваемых строк */

cmpstr(char*s1,char*s2)
{int r=*s1-*s2,i=1;
while(!r && (*s1+*s2)){s1++;s2++;i++;r=*s1-*s2;}
if(r==0) return 0;else return(255-i);
}

void main(int c, char* w[])
{//В командной строке должно быть имя файла
if(c<2)
{cprintf("Неполадки с параметрами в командной строке");return;};
FILE* f=fopen(w[1],"r");
if(f==NULL)
{cprintf("\r\nНеудача с открытием файла %s",w[1]);return;};

//Сосчитаем количество слов в файле
for(unsigned wc=0;!feof(f);fscanf(f,"%s",word),wc++);
rewind(f); //возвратим указатель чтения - записи в начало

//Теперь можем попросить память под указатели на слова
char** wptr=(char**) malloc(wc*sizeof(char**));
if(wptr==NULL)
{cprintf("\r\nНеудача с выделением памяти для адресов слов");
exit(255);};

/*Перепишем файл в память с попутным ее выделением для каждого слова */
for(unsigned i=0;i<wc;i++)
{fscanf(f,"%s",word);
wptr[i]=(char*)malloc(strlen(word)+1);
if(wptr[i]==NULL)
{cprintf("\r\nНеудача с выделением памяти для слов");
exit(255);};
strcpy(wptr[i],word);
}
//Сортируем
unsigned j;
char* t;
for(i=0;i<wc-1;i++)
for(j=i+1;j<wc;j++)
if(strcmp(wptr[i],wptr[j])>0)
{t=wptr[i];wptr[i]=wptr[j];wptr[j]=t;}

//Выводим результат сортировки
clrscr();
cprintf("\r\nОтсортированные по алфавиту слова");
for(i=0;i<wc;i++) cprintf("\r\n%s",wptr[i]);
```

```
//Проверяем, нужен ли поиск, если да - выполняем
unsigned index,ibeg=0,iend=wc-1,flag_break=1;
if(c>2){
do
{
index=(ibeg+iend+1)>>1;
if(strcmp(wptr[index],w[2])==0)
{cprintf("\r\nИндекс слова %s равен %u",w[2],index);
flag_break=0;break;}
if(strcmp(wptr[index],w[2])>0) iend=index-1;
if(strcmp(wptr[index],w[2])<0) ibeg=index+1;
}while((iend-ibeg+1)>0); //Пока длина подмассива не 0

if(flag_break) { printf("Слово не найдено \n");
//Проверим окрестности
 if(cmpstr(wptr[index-1],w[2])<cmpstr(wptr[index],w[2]))
 printf("Ближайшее %s с номером %d",wptr[index-1],index-1);
 else
 if(cmpstr(wptr[index+1],w[2])<cmpstr(wptr[index],w[2]))
 printf("Ближайшее %s с номером %d",wptr[index+1],index+1);
 else
 printf("Ближайшее %s с номером %d",wptr[index],index);
 }//if flag_break
} //if c>2
//Освобождаем арендованную память
for(i=0;i<wc;i++) free(wptr[i]);
free(wptr);
}
```

6.17.3. Синтаксический анализ выражений (формул), заданных символьной строкой.

Эта задача является вспомогательной в программах интерпретации формул, заданных пользователем в виде символьных строк, в частности:

- при построении графиков функций, задаваемых пользователем вводом с клавиатуры;
- при выполнении расчетов по часто и непредсказуемо изменяющимся формулам - например, при начислении налогов, в программах - калькуляторах и пр.

В общем случае в формуле могут встретиться 2 класса лексем - операции и операнды и дополнительные промежуточные классы, например, "конец формулы", "строка", "выражение в скобках". К операциям мы отнесем не только очевидный набор арифметических символов {+,-,^,*,/}, но и все

функции, которые распознает и выполняет наш интерпретатор (например sin(выражение), cos(выражение) и пр.).

Классы можем закодировать например так:

```
const OPERAND=1,OPERATION=2,SKOBKA=3, EOL=4;
```

Эти значения будем использовать для инициализации глобальной переменной token_type.

Допустимые для использования в формулах операции (подклассы класса OPERATION) тоже закодируем, чтобы ускорить операции обработки (это называется переводом во внутренний формат):

Коды операций

```
const
MUL=1,DIV=2,POW=3,PLUS=4,MINUS=5,   //Арифметика
ABS=6, ACOS=7,ASIN=8, ATAN=9, COS=10, COSH=11, EXP=12,
LOG=13,  LOG10=14,  SIN=15,  SINH=16,  SQRT=17,  TAN=18,
TANH=19;
```

Все обозначения (имена) операций и их коды сведем в таблицу (массив структур tf[]) по шаблону

```
struct {char on[10];int ov;}
op[]=
{
{"ABS",ABS},{"ACOS",ACOS},{"ASIN",ASIN},{"ATAN",ATAN},
{"COS",COS},{"COSH",COSH},{"EXP",EXP},{"LN",LOG},
{"LOG",LOG10},{"SIN",SIN},{"SINH",SINH},{"SQRT",SQRT},
{"TAN",TAN},{"TANH",TANH},{"+",PLUS},{"-",MINUS},
{"*",MUL},{"/",DIV},{"^",POW},{"",0}};
```

причем в поля op[i].on занесли имена операций, а в поля op[i].ov значения их числовых кодов.

Мы использовали поддерживаемый в C набор математических функций, вы можете его сократить или расширить по потребности.

К операндам (подклассы класса OPERAND) отнесем переменные, именованные константы, непосредственно заданные числа и закодируем:

```
const STRING=1, //просто неклассифицированная еще строка
      NUMBER=2, //числовая строка
      VARIABLE=3,//переменная
      CONSTANT=4,//константа
      EXPRESS =5;//выражение в скобках
```

Значения кодов подклассов будем заносить в глобальную переменную tok.

Будем предполагать, что допустимые для использования в формулах именованные константы сведены в константный массив структур вида:

```
struct cnst {char cn[6],double cv};   //шаблон структуры
     //константный массив структур
const struct cnst c[2]={{"PI",M_PI},{"E",M_E}};
```

При этом в поля `c[i].cn` записаны имена констант, а в поля `c[i].cv` – их числовые коды.

Предварительная обработка формульной строки может осуществляться еще при клавиатурном вводе фильтрацией недопустимых символов, а если нет уверенности, что это выполнено достаточно эффективно, то необходимо в простейшем случае удалить из нее все недопустимые символы –пробелы, символы табуляции, возврата каретки, перевода строки, всевозможные разделители типа запятой или двоеточия, буквы нелатинского алфавита, оставив только предусмотренные к распознаванию знаки.

Кроме того необходимо перед анализом привести все буквенные символы к верхнему регистру.

Таким образом, первое, что нам понадобится – это инструмент для выделения и классификации лексем (неделимых единиц формульного языка) из текста формулы. Остальные разъяснения алгоритма синтаксического анализа формульной строки мы дадим в комментариях к отдельным строкам приводимого ниже текста программы.

```
#include <dos.h>
#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
```

/*Для хранения кода класса лексемы выделим глобальную переменную `token_type`, а для кода ее конкретного значения – `tok`

Приведенный ниже текст можно записать в заголовочный файл, например, `syntax.h` для включения в использующие программы */

```
/*--------------Файл syntax.h---------------------*/
     int token_type,tok;
     char token[10]; //для хранения выделенной лексемы
     //Коды классов лексем
     const OPERAND=1,OPERATION=2,SKOBKA=3,EOL=4;
     //Подклассы операндов
     const STRING=1, //просто неклассифицированная еще строка
```

```
    NUMBER=2, //числовая строка
    VARIABLE=3,//переменная
    CONSTANT=4,//константа
    EXPRESS =5;//выражение в скобках
    //шаблон структуры констант
    struct cnst {char cn[6];double cv;}
     //константный массив структур
    tc[2]={{"PI",M_PI},{"E",M_E}};
    //Коды операций
const
MUL=1,DIV=2,POW=3,PLUS=4,MINUS=5,   //Арифметика
ABS=6, ACOS=7,ASIN=8, ATAN=9, COS=10, COSH=11, EXP=12,
LOG=13,LOG10=14,  SIN=15,  SINH=16,  SQRT=17,  TAN=18,
TANH=19;

//Все обозначения (имена) операций и их коды сведем в таблицу
//(массив структур tf[]) по шаблону
struct {char on[10];int ov;}
op[]=
{
{"ABS",ABS},{"ACOS",ACOS},{"ASIN",ASIN},{"ATAN",ATAN},
{"COS",COS},{"COSH",COSH},{"EXP",EXP},{"LN",LOG},
{"LOG",LOG10},{"SIN",SIN},{"SINH",SINH},{"SQRT",SQRT},
{"TAN",TAN},{"TANH",TANH},{"+",PLUS},{"-",MINUS},
{"*",MUL},{"/",DIV},{"^",POW},{"",0}};

char *OP="+-*/^"; //Перечень арифметических операций

    //НЕКОТОРЫЕ СЛУЖЕБНЫЕ ПОДПРОГРАММЫ
    //Функция приведения латинских букв к верхнему регистру
void touppercase(char *f)
{
char *temp;
for(temp=f;*temp!='\0';temp++)
if(islower(*temp)) *temp=toupper(*temp);
}

//Функция очистки формульной строки от излишков и неточностей
void clearformula(char *f)
{
char *temp;
temp=f;
for(;*temp!='\0';)
if(isalnum(*temp) || strchr(OP,*temp)
|| *temp=='.' || *temp=='(' || *temp==')') temp++;
//Вначале удалим все недопустимые в формуле символы
else memmove(temp,temp+1,strlen(temp+1)+1);
```

```
//Затем несколько раз на наличие сочетаний типа "+*" или "-*"
temp=f+1;
for(;*temp!='\0';temp++)
if((*temp=='*' || *temp=='/' || *temp=='^') &&
(*(temp-1)=='+' || *(temp-1)=='-'))
{
memmove(temp,temp+1,strlen(temp+1)+1);
temp=f;
}
}
```

/*Функция поиска по таблицам для определения типа строковых лексем параметры функции - искомая строка и адрес, по которому положить значение обнаруженной именованной константы */

```
void look_up(char * s,double* cv)
{
int i;
//Просмотрим таблицу операций
for(i=0;strlen(op[i].on);i++)
//Если нашли совпадение с допустимым именем операции
//возвращаем конкретный тип операции

if(!strcmp(op[i].on,s))
{token_type=OPERATION;tok= op[i].ov;return;}

//Поиск в таблице констант
for(i=0;strlen(tc[i].cn);i++)
/*Если нашли совпадение с допустимым именем константы возвращаем индекс константы в массиве структур и ее значение записываем по заданному адресу */

if(!strcmp(tc[i].cn,s))
{token_type=OPERAND;tok=CONSTANT;*cv=tc[i].cv;return;}
//Если ничего не нашли
token_type=0;tok=-1;
}
//Функция для вывода сообщения об ошибке и возврата к вводу
//пользователя

void serror(int error)
{
static char *e[]= {
"Синтаксическая ошибка",
"Непарные круглые скобки",
"Где-то не выражение",
"Где-то не переменная",
"Есть нераспознанная операция",
```

```
"Не распознанное имя константы",
"Получается деление на нуль"
};
cprintf("%s",e[error]);
getch();
exit(233);
}
```

/*Функция выделения и классификации очередной лексемы возвращает класс лексемы. Ее параметры - указатель на указатель текущего текста формулы и адрес для значения операнда-константы - он транзитом передается в look_up(). Первый параметр мы вынуждены получать "по ссылке", чтобы значение адреса текущей лексемы изменялось в вызывающей программе, а не в нашей подпрограмме */

```
get_token(char **cf,double*cv)
{
char *temp; //Временный указатель на лексему
token_type=0;tok=0;
char*opr;

//Если конец формулы
if(**cf=='\0') {*token=0; return(token_type=EOL);}

//Если это открытая круглая скобка
if(**cf=='(')
{temp=token;
*temp=**cf;        //перепишем лексему в token
(*cf)+=1;           // переход на следующую позицию
temp++; *temp=0; //token закроем нулем
tok=EXPRESS;
return(token_type=OPERAND);}

/*Если это закрытая круглая скобка
if(**cf==')')
{//(*cf)+=1;
temp=token;
*temp=**cf;    //перепишем его в token
(*cf)+=1;      // переход на следующую позицию
temp++; *temp=0; //token закроем нулем
tok=EXPRESS;
return(token_type=SKOBKA);}
*/

//Если это символ арифметической операции
if((opr=strchr(OP,**cf))!=NULL)
 {
```

```
 temp=token;
*temp=**cf;    //перепишем его в token
(*cf)+=1;      // переход на следующую позицию
temp++; *temp=0; //token закроем нулем
switch(*opr)
{
case '+': tok=PLUS;break;
case '-': tok=MINUS;break;
case '*': tok=MUL;break;
case '/': tok=DIV;break;
case '^': tok=POW;break;
}
     //сообщим что класс лексемы - операция
return (token_type=OPERATION);
}

//Если встретили цифру
if(isdigit(**cf)) {
//то запишем всю числовую подстроку в token
temp=token;
while(isdigit(**cf) || **cf=='.') {*(temp++)=**cf;(*cf)+=1;}
*temp = '\0';
tok=NUMBER;
return(token_type = OPERAND);
}

//Если встретили букву
if(isalpha(**cf)) {
     /*то это переменная или операция - функция или именованная константа; пока все буквы - цифры перепишем и временно зарегистрируем просто строкой */

temp=token;
while(isalnum(**cf)) {*temp++=**cf;(*cf)+=1;}
token_type=STRING;}
*temp = '\0';

//Не отходя далеко проанализируем полученную строку

//Если это 1-буквенное имя независимой переменной
if(token_type==STRING && !strcmp(token,"T"))
{tok=VARIABLE;return(token_type=OPERAND);}

//В противном случае поищем среди унарных операций и констант
look_up(token,cv);
if(tok<0) serror(4);
}
/*-------------Граница файла syntax.h-------------*/
```

/* Эта подпрограмма может использоваться как вспомогательная, например, в интерпретаторе для вычисления значений формулы при различных значениях независимой переменной. Запишите эти подпрограммки в заголовочный файл, например, gettok.h или syntax.h и включайте его в использующую программу. В следующих разделах это будет продемонстрировано, а пока мы просто протестируем приведенные функции в программе, выводящей на экран отдельные лексемы заданной в командной строке формулы, их классы и подклассы. */

```
char formula[80]; //Для сохранения текста формулы
double cv;
void main(void)
{
if(_argc!=2)
{cprintf("Неполадки с параметрами в командной строке");return;};
strcpy(formula,_argv[1]); //Сохраняем текст формулы
touppercase(formula); //Приводим к верхнему регистру
clearformula(formula);//Убираем недопустимые символы
char *t=formula;
clrscr();
for(get_token(&t,&cv);token_type!=EOL;get_token(&t,&cv))
cprintf("\r\nЛексема %s код класса %d код подкласса %d",
token,token_type,tok);
}
```

6.17.4. Вычисление значений выражений, заданных пользователем в символьной форме на стадии выполнения обслуживающей программы.

Задача интерпретирующей подпрограммы – вернуть вычисленное значение при заданном значении аргументов. Для упрощения и без того достаточно громоздкого алгоритма мы будем предполагать, что формула представляет собой функцию f(t) одного аргумента с фиксированным именем 't' и в нее могут входить в качестве операндов числа, именованные константы (например 'pi', 'e' и пр.), аргумент 't' и набор различных математических операций и функций.

В качестве вспомогательного средства функция – интерпретатор будет использовать подпрограммы синтаксического анализатора строковых выражений, который будет вызываться для получения каждой следующей лексемы – неделимой единицы выбранного "формульного" языка, который у нас по возможности не будет отличаться от принятого в математике.

Неизбежное отличие будет конечно состоять в том, что например произведение dcos(t) придется записывать в виде d*cos(t) с явным, а не подразумеваемым обозначением операции умножения.

Подпрограммы реализации синтаксического анализатора поместим в файл gettok.h и включим его в нашу программу директивой препроцессора.

Собственно интерпретацию и вычисление значения формулы при заданном значении аргумента t будем осуществлять по следующему алгоритму:

Вначале просмотрим текст формулы, предварительно "очищенный" специальной подпрограммой от пробелов и явных ошибок в расстановке операций - например, от знаков умножений, делений, возведения в степень сразу после знаков других операций; в процессе просмотра заполним массив структур, полями которых являются коды операций или операндов и (для операндовой структуры) значения операндов.

При получении операнда класса EXPRESS (выражение в скобках) мы просто перепишем все внутрискобочное выражение в приватный массив подпрограммы и рекурсивно вызовем ее для вычисления внутрискобочного выражения - еще один пример использования рекурсии для упрощения метода решения задачи.

После построения структуры формулы в виде последовательности операций и операндов просто приступим к выполнению операций в порядке их приоритетов - сначала все унарные операции, включая математические функции от скобочных выражений, затем возведения в степень, потом умножения и деления и в последнюю очередь сложения и вычитания. После выполнения каждой операции количество элементов в массиве формульных структур будем сокращать после записи результата операции в поле соответствующего операнда и перемещения влево области занимаемой структурами массива памяти.

После выполнения последней операции результат окажется в 0-м элементе массива структур - оттуда его и возвратит наша функция интерпретации выражений.

```
#include <dos.h>
#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include"syntax.h" //Выделение лексем
```

```
char formula [80]; //Для сохранения текста формулы
double t;       //Для значения аргумента

/* Нам понадобятся некоторые вспомогательные функции, в
частности функция выполнения заданных двухместных матема-
тических операций – ее параметры при вызове – это код требу-
емой двухместной операции и значения левого и правого опе-
рандов.
     Все приводимые ниже функции целесообразно поместить в
заголовочный файл с названием например interpr.h для по-
следующего использования */

/*---------------Файл interpr.h-------------------*/

double oper(int co,double lo, double ro)
{
switch(co)
{
case PLUS :return(lo+ro);
case MINUS:return(lo-ro);
case MUL :return(lo*ro);
case DIV :{if(ro==0) serror(6);else return(lo/ro);}
case POW :return(pow(lo,ro));
default: serror(0);
}
}

     /*Эта функция осуществляет унарные операции, к классу
которых мы отнесли помимо унарных "плюс" и "минус" все ма-
тематические функции, полагая находящееся в скобках после
имени функции выражение единственным их операндом. Для
сокращения дефицитного места в книге мы привели реализации
всего нескольких математических функций, а вы сможете рас-
ширить эти возможности по аналогии */

double unary(int co,double ro)
{
switch(co)
{
case PLUS :return(ro);
case MINUS:return(-ro);
case SIN :return sin(ro);
case COS :return cos(ro);
case TAN :return tan(ro);
case SQRT :return sqrt(ro);
default: serror(0);
}
```

```
}
```

/*Это уже основная подпрограмма вычисления значений по тексту формулы, она возвращает вычисленное значение, а ее параметры - указатель на текст формулы и значение независимой переменной */

```
double result; //для промежуточных результатов вычислений
char privat_form[80]; //для текущего фрагмента формулы
double get_exp(unsigned char* f, double t)
{
int sc,              //Счетчик скобок
 i,j,                //рабочие переменные для циклов
 co;            //для кода операции
double cv;           //для значения именованной константы
```

/* Формула состоит из операндов и операций. Типы операндов - число, именованная константа, переменная, выражение в скобках. */

```
struct {            //Для структуры формулы
unsigned dtok,      //Код типа операнда
co;                 //Код операции
double z;       //значение операнда
}frm[32];

for(i=0;i<32;i++)
memset(&frm[i],0,sizeof(frm[i]));//Обнулим

unsigned cnt=0;    //Количество элементов в формуле

unsigned char* tmp=f; //Исходный адрес формулы

     //Просмотрим текст формулы и заполним ее структуру
for(i=0,get_token(&tmp,&cv);token_type!=EOL;i++,
get_token(&tmp,&cv))
{
//Если получили операнд - число, константу или переменную
if(token_type==OPERAND)
{frm[i].dtok=tok;
if(tok==NUMBER) frm[i].z=atof(token);
if(tok==CONSTANT) frm[i].z=cv;
if(tok==VARIABLE) frm[i].z=t;
if(tok==EXPRESS)  //Если операнд - выражение в скобках
{
int tt=token_type;//Сохраним тип лексемы
```

```
sc=1;
```
/*Перепишем все после нее до парной ей закрытой в массив privat_form, создавая таким образом другую, частную формулу, как фрагмент общей - для нее мы можем использовать такой же механизм исследования */

```
            for(j=0;sc && j<78;j++,tmp++)
            {
            if(*tmp=='(')sc++; if(*tmp==')')sc--;
            privat_form[j]=*tmp;
            }
            privat_form[j-1]=0;//Закроем нулем
            if(sc) serror(1); //Если не нашли парную скобку
            result=get_exp(privat_form,t);
            token_type=tt;//Восстановим тип лексемы
            frm[i].dtok=NUMBER;
            frm[i].z=result;
            }//if EXPRESS

}//if OPERAND

//Если получили операцию
if(token_type==OPERATION) frm[i].co=tok;
} //for

cnt=i;    //Количество заполненных элементов
```

/*Теперь можем обрабатывать. Вначале необходимо выполнить все унарные операции - у них самый высокий приоритет. Признаком унарной операции является либо если она первая в структуре формулы, либо предыдущий в формуле элемент - операция, а последующий - операнд */

```
int flag=1;
for(;flag;){ flag=0;
for(i=0;i<cnt;i++)
if((!i && frm[i].co && frm[i+1].dtok) ||
(i && frm[i-1].co && frm[i].co && frm[i+1].dtok))
{frm[i+1].z=unary(frm[i].co,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+1],(cnt-i-1)*sizeof(frm[i]));cnt--;flag++;
}
}
/*Теперь необходимо выполнить все возведения в степень */
for(i=1;i<cnt;i++)
if(frm[i].co==POW && frm[i-1].dtok && frm[i+1].dtok)
{frm[i-1].z=oper(POW,frm[i-1].z,frm[i+1].z);
//Сомкнем ряды на выполненной операции
```

```
memmove(&frm[i],&frm[i+2],(cnt-i-2)*sizeof(frm[i]));i--;cnt-=2;
}
/*Теперь выполним все деления и умножения */
for(i=1;i<cnt;i++)
if((frm[i].co==MUL || frm[i].co==DIV) && frm[i-1].dtok &&
frm[i+1].dtok)
{frm[i-1].z=oper(frm[i].co,frm[i-1].z,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+2],(cnt-i-2)*sizeof(frm[i]));i--;cnt-=2;
}
/*Теперь выполним все сложения и вычитания */
for(i=1;i<cnt;i++)
if((frm[i].co==PLUS || frm[i].co==MINUS) &&
frm[i-1].dtok && frm[i+1].dtok)
{frm[i-1].z=oper(frm[i].co,frm[i-1].z,frm[i+1].z);
//Сомкнем ряды на выполненной операции
memmove(&frm[i],&frm[i+2],(cnt-i-2)*sizeof(frm[i]));i--;cnt-=2;
}
//Теперь в 0-м элементе лежит результат
return frm[0].z;
}
/*----------Граница файла interpr.h--------------*/

/* Пусть текст формулы и значение аргумента, при котором
она должна быть вычислена, задается в командной строке */

void main(void)
{

if(_argc!=3)
{cprintf("Неполадки с параметрами в командной строке");return;};
strcpy(formula,_argv[1]); //Сохраняем текст формулы
t=atof(_argv[2]);
touppercase(formula);
clearformula(formula);
double r=get_exp(formula,t);
clrscr();
cprintf("Вычисленное значение при t=%lf равно %lf",t,r);
}
```

6.18. Работа в графическом режиме с библиотекой Borland Graphics Interface (BGI) при построении графиков функций.

Предположим, что перед нами поставлена задача составить программу - графический интерпретатор функций, прини-

мающую от пользователя уравнения плоских кривых в параметрической форме (x=fx(t); y=fy(t)) и диапазон изменения параметра t и строящая по этим уравнениям графики соответствующих функций.

Для решения этой задачи нам, очевидно, пригодятся рассмотренные в предыдущих разделах программы синтаксического анализа заданных в строковом виде формул, интерпретатор строковых выражений.

Составление любой программы лучше всего начинать с составления ее словесного плана-алгоритма, вначале крупноблочного, а затем постепенно детализируемого. Это даст возможность структурировать программу разбиением на отдельные подпрограммы (разложить сложную задачу на ряд простых), определить функциональное назначение всех подпрограмм.

Итак, начнем:

1.Сохраним содержимое экрана при входе в нашу программу, чтобы восстановить перед выходом из нее и "не наследить".

2.Оформим интерьер для ввода-вывода. Выведем приглашение пользователю для ввода уравнений кривых. Дадим ему возможность свободно редактировать текст вводимых формул, перемещаясь между предусмотренными для этого строками ввода. Результаты ввода и редактирования должны отображаться и на экране и в буфере памяти.

3.После завершения ввода вычислим значения переменных с определенным шагом и результаты вычисления сохраним в таблице для последующего построения графика. Для этого вызовем подпрограмму - интерпретатор текста формул (нам ее потом предстоит написать) для расчета массивов значений x и y при различных значениях параметра t.

4.Инициализируем графический режим и определим основные параметры - максимальные значения координат и пр.

5.Оформим интерьер для вывода графика функции, просчитаем масштабы по осям координат.

6.Нарисуем график и, когда пользователь им налюбуется и нажмет, например, Esc - предложим ему продолжить работу с другой кривой или с той же самой, но с другими коэффициентами. По желанию пользователя завершим работу программы.

Пункты с 2-го по 6-й будем повторять циклически, пока пользователь не введет признак завершения программы - нам этот признак тоже предстоит придумать.

8.Выйдем из графического режима и восстановим содержимое текстового экрана, из которого мы стартовали.

Вот крупным планом и все – теперь на очереди детализация плана.

Этот же план в виде главной функции main на языке Си выглядит так:

```
#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include <setjmp.h> //Для функций дальних переходов
#include <math.h>
#include <ctype.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <graphics.h>
#include"keycodes.def" //Двухбайтовые коды клавиш
#include"formread.h" //Здесь редактор для ввода формул
#include"syntax.h"     //Выделение лексем
#include"interpr.h"   //Интерпретация формул
#include"tablev.h" //Расчет таблиц для построения графиков
#include"makegraph.h" //Построение графиков функций по таблицам x,y

jmp_buf e_buf;//Буфер для дальнего возврата по ошибкам
char sbuf[4096];   //для сохранения экрана

void main(void)
{
int key1,i;
gettext(1,1,80,25,sbuf);   //запоминаем экран в sbuf
textattr(0x07);clrscr();

   /*Заполним буфер для дальнего возврата при ошибке, обнаруженной синтаксическим анализатором – именно со следующего фрагмента редактирования формул предстоит начинать исправление ошибок */
setjmp(e_buf);

for(;;)       //В бесконечном цикле повторяем
{
//ввод текста формулы и пределов изменения аргумента
if((key1=formread())==Esc) break;

//Приведение всех строк к одному регистру и очистка
for(i=0;i<4;i++)
{touppercase(formula[i]);clearformula(formula[i]);}
```

```
/*функция расчета таблиц, использующая синтаксический анализатор текста формулы */
tablevar();

//Переход в графику с сохранением текста
ginit();

//вывод графика функции
gshow();

fflush(stdin); //Очищаем буфер ввода
restorecrtmode();//Снова в текстовый режим
}
//Перед выходом из программы восстанавливаем экран
textattr(0x07);clrscr();
puttext(1,1,80,25,sbuf);
}
```

Приведем некоторые разъяснения.

1. Сохранение и восстановление текстового экрана.

Любая "культурная" программа должна после своего завершения оставить используемые ресурсы в том же состоянии, в котором они находились до ее начала; в нашем случае это прежде всего экран монитора, на котором мы собираемся что-то рисовать. Чтобы потом восстановить его содержимое, мы должны запомнить его при входе в программу.

Будем для упрощения предполагать, что мы стартуем из текстового режима номер 3 с 25-ю строками и 80-ю символами в строке. (Строго говоря, это не обязательно так - в наиболее общем случае программа должна определить, из какого режима она стартует и вызывать соответствующие подпрограммы сохранения и восстановления. Но пока для упрощения мы будем думать, что стартуем из наиболее распространенного режима). Так как каждый символ в видеопамяти представлен 2-мя байтами (байт кода символа и байт атрибута), то для хранения образа символьного экрана нам понадобится 2*25*80=4000 байт или 2000 2-байтовых слов.

Отведем для хранения массив

```
char video[4000];
```

и сохраним экран с помощью библиотечной функции

```
gettext(1,1,80,25,video);
```

с прототипом в conio.h. Перед выходом из программы мы сможем восстановить содержимое экрана из массива video с помощью функции

```
puttext(1,1,80,25,video);
```

Можно это же сделать прямым обращением к видеопамяти - для этого надо определить указатель на видеобуфер, адрес которого для текстового режима номер 3 равен 0xB8000000:

```
char * vptr=(char*)0xB8000000;
```

и затем просто прочитать из видеобуфера:

```
for(int i=0;i<4000;i++) video[i]=vptr[i];
```

При восстановлении в конце программы в этом случае аналогичный цикл переписи назад:

```
for(i=0;i<4000;i++) vptr[i]=video[i];
```

2.Диалог с пользователем и редактирование текстов формул.

Теперь можем заняться интерьером. Для этого нужно представить себе, как будет выглядеть удобный для пользователя экран во время работы с программой. В простейшем случае это 2 прямоугольных области, в одной из которых будет диалог с пользователем по поводу формул и диапазона изменения параметра, а во второй будет рисоваться график кривой. После отрисовки кривой во второй области активной снова должна становиться первая и желательно без стирания предыдущих формул - ведь пользователю может понадобиться изменить в формуле только какие-нибудь численные коэффициенты. Здесь не было бы особых проблем, если бы курсор в графическом режиме не был невидимым - пользователь не видит, куда будет введен очередной символ, если мы об этом не позаботимся в нашей программе. Можно было бы создать свой программно управляемый графический курсор, но это слишком затемнило бы основную нашу задачу второстепенными деталями и осложнило бы восприятие учебного материала. Поэтому мы принимаем следующее решение: ввод пользователя осуществляем в текстовом режиме, по сигналу пользователя о завершении ввода сохраняем содержимое окна ввода в отведенном для этого буфере с помощью функции gettext(). После перехода в графический режим и перевода функций из conio.h в режим работы через прерывания bios (установкой переменной directvideo=0 вместо прямого доступа к видеобуферу при directvideo=1) восстанавливаем окно ввода с помощью puttext() из буфера сохранения. При работе с видеоадаптерами устаревших моделей класса CGA могут возникнуть проблемы с изменением цветов; избежать этого можно предварительной обработкой байтов атрибутов в буфере сохранения, но мы этим заниматься не будем.

Мы пока займемся текстовой частью, а графикой позже. Оформим массив строк inpstr:

Файл formread.h

```
const
xbeg=2,ybeg=1,xend=78,yend=8, //Размеры и позиция окна диалога
stroffs=12,lstr=60;    //Координаты и длина строк ввода

char *inpstr[]=              //Массив строк диалога
{
"╔═Введите параметрическое кривой и пределы параметра t═╗",
"║ X(t)=                                                ║",
"║ Y(t)=                                                ║",
"║ tmin=                                                ║",
"║ tmax=                                                ║",
"║ Ошибка:                                              ║",
"╚═══════F6 -рисовать═════Esc-выход═══════════════════╝ ",
 ""
     };
     /*Примечание: размер рисунка рамки уменьшен здесь по сравнению с фактическим для размещения на книжной странице */

char x_scr,y_scr,temp; //Позиция курсора на экране и в строках
#define x_buf x_scr-1//Для отображения позиции курсора в буфере
#define y_buf y_scr-1 //связываем их через макроподстановку

//Буфер для сохранения окна диалога
char textbuf[2*(xend-xbeg+1)*(yend-ybeg+1)];

//для пределов переменных
float tmin,tmax,xmin,xmax,ymin,ymax;

int index;          //Текущий номер строки ввода

char formula[4][80]; //Для текста формул и пределов параметра

//Функция для ввода формул и пределов изменения параметра
 formread()
 {
 int i,j,key;
 directvideo=1;     //Работаем прямо с видеопамятью
 window(1,1,80,25); //Занимаем весь экран
 //Выведем шаблон для ввода формул
 textattr(0x0A);clrscr();
 for(i=0;strlen(inpstr[i]);i++)
 {gotoxy(xbeg,ybeg+i); printf("%s",inpstr[i]);}
 //Окно внутри рамки
 window(xbeg+stroffs,ybeg+1,xbeg+lstr,yend-1);
```

```
//Если в строках буфера ввода что-то есть - выведем это
for(i=0;i<4;i++)
if(strlen(formula[i]))
{gotoxy(1,i+1);cputs(formula[i]);}

x_scr=1;y_scr=1;
gotoxy(x_scr,y_scr); //Курсор в начальную позицию

//Теперь предстоит создать 4-хстрочный текстовый редактор
for(;;)
{
    fflush(stdin); //Очистим клавиатурный буфер
    key=bioskey(0); //Получаем 2-хбайтовый код клавиши
    switch(key)
    {
    case F6:  //Если ввод закончен
                  //Сохраняем окно с введенным текстом
    gettext(xbeg,ybeg,xend,yend,textbuf);
    return 0;
    //Если пользователь закончил работу с программой
     case Esc:return Esc; //на выход!

     /*Управление курсором - действия по клавишам - стрелкам сводятся к изменению позиции курсора и через макроподстановку - позиции в буфере x_buf,y-buf */

     case     Left:if(x_scr>1)
     {x_scr--;gotoxy(x_scr,y_scr);}break;

     case Right:if(x_scr<(strlen(formula[y_buf])+1))
              {x_scr++;gotoxy(x_scr,y_scr);}break;

     case Up: if(y_scr>1)
             {y_scr--;x_scr=1;gotoxy(x_scr,y_scr);}break;

     case Down: if(y_scr<5) {y_scr++;x_scr=1;
     gotoxy(x_scr,y_scr);}break;

     case Home: x_scr=1;gotoxy(x_scr,y_scr);break;

     case End: x_scr=strlen(formula[y_buf]);
     gotoxy(x_scr,y_scr);break;

     /*Редактирование текста
При стирании символа весь "хвост" после него смещаем на 1 байт влево в памяти и на экране (стиранием и выводом) */
```

```
case Del:
memmove(formula[y_buf]+x_buf,
formula[y_buf]+x_buf+1,strlen(formula[y_buf]+x_buf+1)+1);
temp=x_scr;clreol();cputs(formula[y_buf]+x_buf);
x_scr=temp;gotoxy(x_scr,y_scr); break;

case BackSpace:
if(x_scr>1){
memmove(formula[y_buf]+x_buf-1,
formula[y_buf]+x_buf,strlen(formula[y_buf]+x_buf)+1);
gotoxy(--x_scr,y_scr);
temp=x_scr;clreol();
cputs(formula[y_buf]+x_buf);
x_scr=temp;gotoxy(x_scr,y_scr);
}
break;

default:
//Если символ допустим в тексте формулы
if((char)key){
if((y_scr<5) && (isalnum((char)key) ||
strchr("*+-/^.()",(char)key)!=NULL))
{
memmove(formula[y_buf]+x_buf+1,formula[y_buf]+
x_buf,strlen(formula[y_buf]+x_buf));
formula[y_buf][x_buf]=(char)key;
temp=x_scr;clreol();
cputs(formula[y_buf]+x_buf);
x_scr=++temp;gotoxy(x_scr,y_scr);
} } } }}
```

3. Наша следующая задача - заполнение таблицы значений t, x и y. При этом помним, что x вычисляется по formula[0], y - по formula[1].

Файл tablev.h

```
const MAX_POINT=50; //Количество вычисляемых точек для графика
double t[MAX_POINT];   //массив значений параметра
//таблица для значений x,y в натуральных единицах
double xy[MAX_POINT][2];
int xyint[MAX_POINT][2]; //таблица для значений x,y в пикселах
```

/*Функция расчета элементов таблиц - использует интерпретатор для вычисления значений x и y при различных значениях параметра t.*/

```
void tablevar(void)
{
 int i;
```

/*Вначале с помощью интерпретатора обрабатываются строки с значениями пределов параметра t - так предоставляется возможность использовать для задания пределов именованные константы типа PI, E и любых других, специфических для какой либо прикладной области - это могут быть физические, химические константы и пр. */

```
char *curf=formula[2];
tmin=get_exp(curf,0);
curf=formula[3];
tmax=get_exp(curf,0);
double incr=(tmax-tmin)/MAX_POINT;
for(index=0;index<MAX_POINT;index++) t[index]=tmin+index*incr;
for(i=0;i<2;i++)
{curf=formula[i];
for(index=0;index<MAX_POINT;index++)
  xy[index][i]=get_exp(curf,t[index]);
}
}
```

4. Инициализация графического режима и оформление интерьера.

Наконец мы добрались до графики.

Собственно инициализация графики в простейшем случае сводится к вызову библиотечной функции initgraph с передачей ей адресов переменных для графического драйвера, графического режима и пути доступа к BGI - драйверу. Но такая программа может быть запущена только на компьютере с имеющимся на диске файлом BGI-драйвера (с расширением .bgi) и, как следствие, - переноситься на всякий случай вместе с файлом этого драйвера на дискете.

Аналогичная проблема возникает с файлами графических масштабируемых сегментированных шрифтов (с расширением .chr) - система BGI предполагает по умолчанию, что путь доступа к этим файлам такой же, как и к драйверу графики. Поэтому целесообразно включить и файл драйвера и файл используемого шрифта непосредственно в код программы. Это делается до инициализации графики следующим образом:

с помощью специальной программы-утилиты в составе пакета Си bgiobj.exe все необходимые программе .bgi и .chr-файлы преобразуются в объектные .obj-файлы, регистрируются в программе с помощью библиотечных функций

```
int registerbgidriver(void (* driver)(void))
int registerfarbgidriver(void far *driver)
int registerbgifont(void (*font)(void))
int registerfarbgifont(void far *font)
```

Эти функции информируют систему BGI-графики о том, что необходимые драйверы присутствуют в коде программы и не требуют загрузки с диска. Аргумент этих Функций - имя регистрируемого драйвера или шрифта, образуемого присоединением к имени драйвера суффикса _driver, _driver_far или _font, _font_far, например (это имя вам сообщит утилита bgiobj.exe):

```
registerbgidriver(EGAVGA_driver);
registerbgifont(triplex_font_far);
```

После этого строка пути доступа к драйверам и графическим шрифтам при вызове может быть пустой.

Еще одно - для компоновки полученных объектных файлов совместно с вашей программой создайте файл проекта и включите в него эти объектные файлы.

Таким образом, функция инициализации BGI-графики может выглядеть так:

Файл makegraph.h

```
unsigned X,Y;           //размеры экрана в пикселах
int graphflag=1;

//инициализация графики
 void ginit(void)
 {
int gd = DETECT,gm, ec;

//Для исключения повторной регистрации драйвера и шрифтов
if(graphflag)
{
ec = registerbgidriver(EGAVGA_driver);
if (ec < 0)
{printf("Ошибка графики: %s\n", grapherrormsg(ec));
getch();exit(1);}
ec = registerbgifont(triplex_font);
if (ec < 0)
{printf("Ошибка графики: %s\n", grapherrormsg(ec));
getch();exit(1);}

ec = registerbgifont(sansserif_font);
if (ec < 0)
{printf("Ошибка графики: %s\n", grapherrormsg(ec));
getch();exit(1);}
//Функция инициализации графического режима
initgraph(&gd, &gm, "");
```

```
ec = graphresult();
if (ec != grOk){printf("Ошибка: %s\n", grapherrormsg(ec));
getch(); exit(1);}
graphflag=0;
}
else setgraphmode(getgraphmode());
```

/*Вывод сохраненного через gettext фрагмента из текстового режима */

```
directvideo=0;     //переходим на вывод через BIOS
puttext(xbeg,ybeg,xend,yend,textbuf);
 }
```

/*Функция вывода графика кривой, заданной массивом xy[RowCount][ColColnt], в графическое окно */

```
 void gshow()
 {
/*Для масштабов по x и y - количества пикселов на натуральные единицы */
 double picx,picy;

 double dx,dy; //Диапазоны переменных
 int i;
 //определим максимальные значения x и y просмотром массива
 xmax=xy[0][0];xmin=xy[0][0];
 ymax=xy[0][1];ymin=xy[0][1];
 for(i=1;i<MAX_POINT;i++)
 {
 if(xy[i][0]>xmax) xmax=xy[i][0];
 if(xy[i][0]<xmin) xmin=xy[i][0];
 if(xy[i][1]>ymax) ymax=xy[i][1];
 if(xy[i][1]<ymin) ymin=xy[i][1];
 }
 dx=fabs(xmax-xmin);dy=fabs(ymax-ymin);
 //Нормируем данные относительно диапазонов
 for(i=0;i<MAX_POINT;i++) {xy[i][0]/=dx;xy[i][1]/=dy;}

 //Определимся с размерами окна и масштабами
 X=getmaxx();Y=getmaxy();
 int scrhight=(3*Y)>>2; //Высота окна
 int scrwidth=scrhight; //Ширина окна - оно будет квадратным
 picy=picx=scrwidth;
 if(dx<dy) picx*=(dx/dy);
 if(dx>dy) picy*=(dy/dx);

 //определяем графическое окно с отсечкой вывода на границах
 setviewport(X-scrwidth,Y>>2,X,Y,1);
 //Установим XOR -стираемый режим вывода отрезков
 setwritemode(XOR_PUT);
```

```
setcolor(6);            //цвет рисования
setfillstyle(1,7);          //заполнение
setlinestyle(0,8,1);        //стиль линий
rectangle(2,2,scrwidth-2,scrhight-2);   //рамка в окне
bar(3,3,scrwidth-3,scrhight-3); //заполненный прямоугольник

    // координаты центра системы координат
    int xc=scrwidth>>1;
    int yc=scrhight>>1;

    //Координатные оси
moveto(5,yc);lineto(scrwidth-5,yc);
linerel(-6,-2);linerel(0,4);linerel(6,-2);//Стрелки на осях

for(i=0;i<10;i++)    //Метки на осях
{moverel(-(int)(0.1*scrwidth),0);
if(i&1) linerel(0,-4);else linerel(0,4);
}

moveto(xc,scrhight-5);lineto(xc,5);
linerel(-2,6);linerel(4,0);linerel(-2,-6);

for(i=0;i<10;i++)    //Метки на осях
{moverel(0,(int)(0.1*scrwidth));
if(i&1) linerel(4,0);else linerel(-4,0);
}

outtextxy(scrwidth-10,yc-10,"X");
outtextxy(xc+10,10,"Y");

    //Вывод кривой
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
//Цикл просмотра и корректировки кривой
int k;
do
{
k=bioskey(0);
switch(k)
{
case Up:
    //Стирание кривой повторным выводом
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}

//Изменение масштаба и по клавишам - стрелкам вывод
```

```
picy*=1.1;
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
break;
case Down:
//Стирание кривой повторным выводом
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
//Изменение масштаба и вывод
picy/=1.1;
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
break;
case Right:
//Стирание кривой повторным выводом
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
//Изменение масштаба и вывод
picx*=1.1;
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
break;
case Left:
//Стирание кривой повторным выводом
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
//Изменение масштаба и вывод
picx/=1.1;
moveto((int)(xy[0][0]*picx)+xc,yc-(int)(xy[0][1]*picy));
for(i=1;i<MAX_POINT;i++)
{lineto((int)(xy[i][0]*picx)+xc,yc-(int)(xy[i][1]*picy));}
break;
default:break;
}
}while(k!=Esc);
//Дополнительное окно
setviewport(0,Y>>2,X-scrwidth,Y,1);
setcolor(6);            //цвет рисования
setfillstyle(1,7);          //заполнение
setlinestyle(0,8,3);        //стиль линий
bar(0,0,scrwidth,scrhight);
rectangle(1,1,scrwidth-1,scrhight-1);   //рамка в окне
settextstyle(DEFAULT_FONT,HORIZ_DIR,1);
```

```
char far* ps[]={"В этом окне можно дать",
                "инструкцию по использованию",
                "настоящей программы"};
for(i=0;i<3;i++)
outtextxy(10,2*(i+1)*textheight(ps[0]),ps[i]);
getch();
}
```

6.19. Работа с массивами структур файлового хранения.

Массивы или списки структур в файлах представляют собой обычно так называемые базы данных - многочисленные совокупности информационных блоков, содержащих возможно разнотипные, но связанные по смыслу и способам совместной обработки данные - это могут быть сведения о служащих некоторой фирмы, библиографический справочник, сведения об автомобилях и их владельцах в базе данных Госавтоинспекции региона, регистрационные данные о посетителях стоматологической поликлиники, операционные сведения за текущий день, с начала недели, месяца, года в коммерческом банке и многое другое.

Для создания баз данных разработан целый ряд специализированных или, как их называют, проблемно-ориентированных языков, например Clipper, а также большое количество интегрированных систем управления базами данных, позволяющих создавать, пополнять, корректировать базы, осуществлять поиск информации по заданным критериям и получать много других услуг без программирования, а также программировать нестандартизованные алгоритмы обработки на входных языках этих систем - к ним относятся FoxPro, Paradox, Oracle, Informix, Sybase и др. Специальные средства для разработки баз данных встраиваются и в языки программирования общего назначения - Delphi, C, C++. Для языка С давно создана специальная библиотека CodeBase для работы с файлами dbf - формата, включающая широкий набор функций, не уступающий по возможностям специализированным инструментальным средствам.

В настоящем разделе мы рассмотрим методы работы с записями (структурами) файлового хранения на простом примере создания пополняемого толкового словаря. Реализующая его программа должна будет обеспечить предоставление пользователю следующих услуг: добавление записей в словарь, логическое и физическое удаление записей, поиск по ключевому

слову, последовательный просмотр содержимого словаря и списка помеченных к удалению записей.

Каждая запись в словаре – это структура с двумя полями типа символьных массивов (строк), одно из которых содержит ключевое слово, а другое, большей длины, толкование этого слова.

Метод создания словаря будет ясен из приводимого ниже текста программы и сопровождающих его комментариев.

```
#include <alloc.h>
#include <dos.h>
#include <mem.h>
#include <stdio.h>
#include <string.h>
#include <io.h>
#include <conio.h>
#include <stdlib.h>

//максимальное количество слов в словаре
#define MAXWORDS  10000
#define KEYLENGTH   20//длина слова
#define VALUELENGTH 255//длина толкования
#define NUMPUNKTS   8//количество пунктов в меню услуг

     /*В структурной переменной по этому шаблону хранятся основные справочные сведения в памяти */
struct shortrec
{
  char  IsDel;//если ==0, запись пустая или помечена к удалению
  char key[KEYLENGTH];//слово
};

/*Этот шаблон предназначен для создания структурных переменных файлового хранения, он содержит в качестве поля структурную переменную по вышеприведённому шаблону и поле толкования */

struct rec
{
  shortrec memrec;//признак занятости и слово-ключ
  char value[VALUELENGTH];//значение по данному ключу
};
//Прототипы подлежащих программированию (определению) функций
void add(), //добавления записей
del(),      //логического удаления записей
find(),     //поиска по ключевому слову
view(),     //просмотра содержимого словаря
pack(),     //физического удаления (упаковки файла)
```

```
undel(),     //отмены логического удаления (пометки об удале-
нии)
dellist(),   //просмотра списка помеченных к удалению записей
quit(),      //завершения работы со словарем
init();      //начальной инициализации
long findinmemory(char *),//поиска записи
rcompare(char *, char *), //сравнения записей
getnumber(char *);        //возвращает номер найденной
char rtolower(char c);    //приведения к нижнему регистру

/*этот массив структур содержит список возможных действий и реак-
цию на них */
struct
{
  char msg[30];//сообщение - пункт меню
  //указатель на функцию, соответствующую данному пункту
  void (*func)();
}message[NUMPUNKTS]=
//количество элементов в массиве равно количеству пунктов меню
{
  {"0. Добавить запись", add},
  {"1. Удалить запись", del},
  {"2. Найти запись", find},
  {"3. Просмотр словаря", view},
  {"4. Упаковать словарь", pack},
  {"5. Отменить удаление", undel},
  {"6. Список удалённых", dellist},
  {"7. Выход", quit}
};
/*временная запись, используемая для различных промежуточных
действий */
rec temp;
shortrec *dictionary;//указатель на справочник в памяти
//параметры главной функции возьмём из внешних переменных
void main()
{
  if(_argc<2)//в качества параметра передадим имя словаря
  {
    printf("Используйте: %s vocab.dat\n",_argv[0]);
    return;//выход из программы
  }
  init();//начальная инициализация
  //в бесконечном цикле обработки сообщений от клавиатуры
  for(;;)
  {
    puts("");//выводим пустую строку перед списком
    for(int i=0;i<NUMPUNKTS; i++)//распечатываем все пункты меню
      printf("%s\n",message[i].msg);
    printf("Ваш выбор : ");//приглашение к диалогу
    //ввод с отсечением недопустимых номеров пунктов меню
```

```
    for(char what=-1;what-'0'<0||
    what-'0'>=NUMPUNKTS;what=getch());
    what-='0';//преобразуем цифровой символ в номер
    printf("%s\n",message[what].msg+3);//печатаем, что выбрали
    message[what].func();//и вызываем соответствующую функцию
  }
}

//Теперь определения объявленных функций
//процедура начальной инициализации
void init()
{
  //отводим память под справочник
  dictionary=(shortrec*)farcalloc(MAXWORDS,sizeof(shortrec));
  if(!dictionary)//если не удалось отвести
  {
    printf("Не хватает памяти под справочник\n");
    exit(1);//выходим в ОС
  }
  //очищаем память, выделенную под справочник
  _fmemset(dictionary,0,MAXWORDS*sizeof(shortrec));
  //открываем двоичный файл для чтения и записи
  FILE *fp=fopen(*(1+_argv),"r+b");
  if(!fp)//Проблемы?
  {
    fp=fopen(_argv[1],"wb");//пытаемся создать файл
    if(fp==0)//нет места, защита от записи etc
    {
      printf("Не могу открыть файл %s\n",_argv[1]);
     //освобождаем память из под справочника
      farfree(dictionary);
      exit(0);            //и выходим
    }
    fclose(fp);
    //переоткрываем вновь созданный файл
    fp=fopen(*(1+_argv),"r+b");
  }
  //определяем число пар слов-значение в словаре
  long RecordCount=filelength(fileno(fp))/sizeof(rec);
  for(long i=0;i<RecordCount;i++)
  { //и считываем слова в справочную часть
    fread(&temp,sizeof(temp),1,fp);
    _fmemcpy(dictionary+i,&(temp.memrec),sizeof(shortrec));
  }
  fclose(fp);//закрываем файл
}

//Функция добавления записи
void add()
{
  FILE *fp=fopen(*(1+_argv),"r+b");//открываем словарь на диске
  memset(&temp,0,sizeof(rec));//очищаем рабочую запись
```

```
  printf("Введите слово : ");//запрашиваем ключевое слово
  scanf("%s",temp.memrec.key);
  if(!findinmemory(temp.memrec.key))//если его нет в словаре
  {
    //вводим с клавиатуры значение с ограничением на длину
    printf("Введите значение этого \
    слова (Enter - конец ввода) :\n");
    *temp.value=VALUELENGTH-3;
    memmove(temp.value,temp.value+2,1+
    strlen(cgets(temp.value)));
    temp.memrec.IsDel='A';//признак неудалённости записи
    for(long i=0;i<MAXWORDS;i++)//дрейф по справочнику в памяти
    //в поисках пустой (удалённой) ячейки
    if(!dictionary[i].IsDel)
      { //заносим слово в память
        memcpy(dictionary+i,&temp.memrec,sizeof(shortrec));
        fseek(fp,i*sizeof(rec),SEEK_SET);
        //а вместе со значением - на диск
        fwrite(&temp,sizeof(rec),1,fp);
        break;
      }
    if(i==MAXWORDS)
    printf("Нет места в словаре для добавления нового слова \n");
  }
  else//если нашли слово в памяти
    printf("Такое слово уже есть в словаре. \
    Попробуйте изменить его значение \n");
  getch();
  fclose(fp);
}

//функция поиска слова в справочнике
long findinmemory(char *key)
{
  //слово считается найденным, если оно не помечено к удалению и
  //совпадает с искомым с точностью до регистра
  for(long i=0;i<MAXWORDS;i++)
    if(dictionary[i].IsDel&&!rcompare(key,dictionary[i].key))
      return 1;
  return 0;
}

//сравнение двух строк без учёта регистра по аналогии с stricmp,
// но с проверкой на русские буквы
long rcompare(char *f, char *s)
{
  if(strlen(f)!=strlen(s))
    return 1;
  for(long i=0;f[i];i++)
    if(rtolower(f[i])!=rtolower(s[i]))
      return 1;
```

```
  return 0;
}
```

//переводит символ в нижний регистр, возвращая результат

```
char rtolower(char c)
{ //отключаем  предупреждения компилятора
  #pragma warn -rng
  if(c>=(unsigned char)'A'&&c<=(unsigned char)'Z')
    c+=0x20;
  if(c>=(unsigned char)'A'&&c<=(unsigned char)'П')
    c+=0x20;
  if(c>=(unsigned char)'P'&&c<=(unsigned char)'Я')
    c+=0x50;
  if(c=='Ё')
    c='ё';
  return c;
  #pragma warn +rng
}

//процедура удаления слова
void del()
{
  FILE *fp=fopen(*(1+_argv),"r+b");//открываем словарь
  memset(&temp,0,sizeof(rec));//очищаем временную запись
  printf("Введите слово : ");
  scanf("%s",temp.memrec.key);//вводим удаляемое слово
  if(findinmemory(temp.memrec.key))//если таковое имеется
  {
    //получаем номер записи с данным словом
    long i=getnumber(temp.memrec.key);
    dictionary[i].IsDel=0;//устанавливаем признак удаления
    //позиционируемся на эту запись в файле
    fseek(fp,i*sizeof(rec),SEEK_SET);
    //записываем только признак
    fwrite(&dictionary[i].IsDel,1,1,fp);
  }
  else
    printf("Такого слова в словаре нет \n");
  getch();
  fclose(fp);
}
```

//функция, ищущая слово в памяти и возвращающая номер записи

```
long getnumber(char *key)
{ //отключаем глупые компилятора, не разобравшего, что эта
  //функция что-то возвращает
  #pragma warn +rvl
  for(long i=0;i<MAXWORDS;i++)//дрейфуем по словам до тех пор,
    if(!rcompare(key,dictionary[i].key))//пока не найдём; тогда
      return i;              //возвращаем номер
  if(i==MAXWORDS)//а может и не найдём...
```

```
    return -1;
  #pragma warn -rvl
}

//если запись только помечена, признак удаления  можно снять
void undel()
{
  FILE *fp=fopen(*(1+_argv),"r+b");//открываем словарь
  memset(&temp,0,sizeof(rec));//очищаем рабочую запись
  printf("Введите слово : ");
  //вводим предположительно удалённое слово
  scanf("%s",temp.memrec.key);
  for(long i=0;i<MAXWORDS;i++)//ищем только среди удалённых
    if(!rcompare(dictionary[i].key,temp.memrec.key)&&
    !dictionary[i].IsDel)
    {
      dictionary[i].IsDel='R';//убираем признак удалённости
      //позиционируемся на удалённую запись
      fseek(fp,i*sizeof(rec),SEEK_SET);
      fwrite(&dictionary[i].IsDel,1,1,fp);//и восстанавливаем
      printf("Восстановлено! \n");
      i=-1;//признак восстановления
      break;
    }
  if(i+1)//если не удалось восстановить
    printf("Такого слова в словаре не было \n");
  getch();
  fclose(fp);
}

//поиск всех слов, содержащих заданное как подстроку
void find()
{
  char key[KEYLENGTH];
  //открываем словарь только для чтения
  FILE *fp=fopen(*(1+_argv),"rb");
  memset(&temp,0,sizeof(rec));//обнуляем рабочую запись
  printf("Введите слово : ");
  scanf("%s",key);//вводим подстроку для поиска
  for(long i=0,count=0;key[i];i++)
    key[i]=rtolower(key[i]);//преобразуем её к нижнему регистру
  //определяем число записей
  long RecordCount=filelength(fileno(fp))/sizeof(rec);
  for(i=0;i<RecordCount;i++)
  {
    fread(&temp,sizeof(rec),1,fp);
    for(long j=0;temp.memrec.key[j];j++)
      temp.memrec.key[j]=rtolower(temp.memrec.key[j]);
    if(strstr(temp.memrec.key,key)&&temp.memrec.IsDel)
      printf("Слово - %s, значение - %s\n",
      temp.memrec.key,temp.value),count++;
  }
```

```
  if(!count)
    printf("Такого слова в словаре нет \n");
  getch();
  fclose(fp);
}

//вывод на экран всего словаря
void view()
{
  FILE *fp=fopen(*(1+_argv),"rb");
  printf("Содержимое словаря \n");
  memset(&temp,0,sizeof(rec));
  for(long i=0,count=0;i<MAXWORDS&&!feof(fp);i++)
    if(dictionary[i].IsDel)//если слово не удалено
    {
      fseek(fp,i*sizeof(rec),SEEK_SET);
      fread(&temp,sizeof(temp),1,fp);//считываем его значение
      printf("%li Слово - %s, значение - %s\n",
      ++count,temp.memrec.key,temp.value);
      getch();//печатаем и даём полюбоваться до нажатия клавиши
    }
  printf("Всего записей : %li\n",count);//вывод статистики
  getch();
  fclose(fp);
}

//вывод списка помеченных к удалению словарных единиц
void dellist()
{
  //пара слово-значение считается удалённой, если слово есть, а
  //признак удаления установлен
  printf("Список удалённых слов: \n");
  for(long i=0,count=0;i<MAXWORDS;i++)
    if(!dictionary[i].IsDel&&strlen(dictionary[i].key))
      printf("%li слово - %s\n",++count,dictionary[i].key);
  printf("Всего удалённых записей : %li\n",count);//статистика
  getch();
}

//Записи, помеченные к удалению, продолжают занимать место,
//а потому их имело бы смысл физически удалить
void pack()
{
  printf("Упаковываю...");
  FILE *fp=fopen(_argv[1],"rb"),
       *ff=fopen("vocab.tmp","wb");
  long RecordCount=filelength(fileno(fp))/sizeof(rec);
  //переписываем только существующие записи
  for(long i=0;i<RecordCount;i++)
  {
    fread(&temp,sizeof(temp),1,fp);
```

```
    if(temp.memrec.IsDel)
      fwrite(&temp,sizeof(temp),1,ff);
  }
  fclose(fp);
  fclose(ff);
  unlink(_argv[1]);
  rename("vocab.tmp",_argv[1]);
  printf("\b\b\b\b\b\bано!    \n");
  getch();
}

//функция выхода - прекращения работы со словарем
void quit()
{
  farfree(dictionary);//освобождаем выделенную память
  exit(0);
}
```

6.20. Методы разработки резидентных программ.

РЕЗИДЕНТНОЙ называют программу, которая после загрузки и выполнения части своего кода записывает свой адрес в таблицу векторов прерываний и остается в памяти после так называемого резидентного завершения, готовая к выполнению по возникновению прерывания, соответствующего номеру вектора с ее адресом в памяти. В литературе такие программы называют TSR (Terminate but Stay Resident).

Трудности в создании резидентных программ в среде MS-DOS обусловлены в основном тем, что сама ОС однозадачна и не поддерживает многозадачных режимов работы с асинхронным поступлением запросов на выполнение лежащих в памяти программ.

Для резидентного завершения MS-DOS предоставляет функцию 31h 21h-го прерывания, которой надо передать размер резервируемой под резидента памяти в регистре DX и код возврата в регистре AL. В языках Си и Паскаль есть соответствующие библиотечные функции под названием keep() для резидентного завершения программ с теми же аргументами в Си и с одним аргументом кода возврата в Паскале.

Очевидно, что TSR-программу можно разделить на 2 функциональные части: инициализирующую (выполняющуюся один раз при инсталляции резидента) и саму TSR , которая активизируется периодически по сигналам аппаратного прерывания или по команде программного генерирования прерывания и состоит, как правило, из одного или более обработчиков преры-

ваний ISR (Interrupt Service Routine) и обычных подпрограмм, вызываемых из ISR.

ISR-подпрограммы должны в Си или Паскале объявляться с модификатором interrupt - это обеспечивает добавку компилятором в эту подпрограмму секций сохранения всех регистров процессора при входе в обработчик прерывания и восстановление их при выходе по специальной команде IRET для обработчиков прерываний. Кроме того, компилятор устанавливает регистр DS так, что ISR получает доступ к статическим и внешним данным программы. Конечно, перед завершением ISR должна выполнить восстановление того, что изменяла в своей работе, чтобы прерванная ею программа, вновь продолжая свое выполнение не встретилась с неожиданностями.

Инициализирующая часть программы должна обеспечить занесение адресов всех ISR в соответствующие векторы прерываний (это называют перехватом прерывания), резидентное завершение программы, установить флаги предотвращения повторной загрузки, перехватить прерывания по Ctrl Break и по критической ошибке MS-DOS.

Применяются 2 основных способа подключения ISR к векторам аппаратных прерываний - "каскадом", когда новый обработчик подменяет своим адресом адрес старого, но вызывает старый на выполнение до или после выполнения собственной работы; второй способ связан с полным замещением старого обработчика без последующих вызовов.

ПЕРЕХВАТ ПРЕРЫВАНИЯ лучше всего выполнить с помощью библиотечных функций. В Си это функции с прототипами в dos.h:

```
void interrupt (*getvect(int номер_прерывания))();
```

для получения адреса точки входа текущего обработчика прерывания. Возвращает far - указатель на функцию типа

```
void interrupt name(void).
```

```
void setvect(int номер_прерывания, void interrupt(*isr)());
```

для записи в таблицу векторов прерываний far-указателя на точку входа в новый обработчик, который должен быть описан как функция с переменным числом параметров:

```
void interrupt isr(...).
```

В Паскале аналогами этих функций являются процедуры из модуля Dos:

процедура GetIntVec(Номер прерывания: byte; VAR адрес: Pointer);
для получения адреса текущего обработчика;
процедура SetIntVec(Номер_прерывания : byte; адрес:Pointer);

Использование библиотечных подпрограмм избавляет нас от необходимости запрещать прерывания на время замены векторов и затем разрешать снова.

Самое сложное в процессе инсталляции резидента - определить размер необходимой для него памяти - проще задать размер в 2 раза больше размера .EXE - файла, а затем после завершения отладки постепенно уменьшать, пока резидент не начнет зависать, после чего снова можно увеличивать, пока работоспособность не восстановится.

ПРЕДОТВРАЩЕНИЕ ПОВТОРНОЙ ЗАГРУЗКИ РЕЗИДЕНТА. Инсталляционная часть программы должна иметь возможность проверить наличие в памяти предыдущей копии резидента - быть может он уже инсталлирован и тогда повторная инсталляция с резидентным завершением не нужна. Эту непростую задачу вы решите на профессиональном уровне, когда его достигнете, либо сканированием памяти с целью обнаружить в ней резидента, либо другими методами. На первом этапе можно посоветовать запись придуманного вами числового кода по одному из пользовательских векторов прерываний, например, по одному из векторов 60h..66h. При этом возникает опасность использовать для записи своего кода занятый каким-либо обработчиком номер; избежать этого можно простым приемом - среди указанных векторов отыскивают 2 с одинаковым значением и используют любой из них.

При деинсталляции TSR, если таковая предусматривается, нужно восстановить старое значение по использованному вектору.

ПРЕДОТВРАЩЕНИЕ ПОВТОРНОГО ВХОЖДЕНИЯ В TSR.

Речь идет о предотвращении возможности прерывания работы обработчика прерывания во время обработки предыдущего прерывания. Эта задача решается просто установкой флага занятости обработчика при входе и снятии его при выходе из обработки. Этот флаг должен проверяться перед возможным входом в подпрограмму обработки прерывания.

ПРЕДОТВРАЩЕНИЕ ПОВТОРНОГО ВХОЖДЕНИЯ В MS-DOS.

MS-DOS и почти все ее подпрограммы - нереентерабельные, то есть не допускают повторного начала своего выполнения до завершения начатого ранее. Следовательно, нереентерабельны и все библиотечные подпрограммы Си или Паскаля, использующие в своей работе DOS. Так как момент поступления аппаратных прерываний непредсказуем (например, момент

нажатия клавиш), то возникает опасность вклиниться в выполняемую сейчас работу ОС со своими обращениями к ее услугам, что немедленно вызовет неустранимую ошибку.

MS-DOS знает свой недостаток и тем TSR, которым нужны ее услуги, сообщает о своей незанятости файловыми операциями периодическим вызовом прерывания 28h. Это позволяет подключить свой обработчик к прерыванию 28h (сделать его рабочим), а обработчику реального прерывания поручить только установку флага поступления прерывания (сделать его активирующим с возможностью откладывания обработки до освобождения ОС), который будет анализироваться при поступлении прерывания с номером 28h и в случае его установки будет выполняться необходимые действия и сброс флага после их завершения. Единственным ограничением является невозможность использования функций DOS с номерами 0..0Ch (клавиатурного и экранного ввода-вывода), что вызовет повторное вхождение в ISR 28h-го прерывания и повисание системы. Но эту проблему можно решить установкой флага занятости на самом входе в обработчик 28h-го прерывания.

ПЕРЕХВАТ ПРЕРЫВАНИЯ КРИТИЧЕСКОЙ ОШИБКИ. Правильно составленная TSR должна подменять перед активизацией вектор 24h обработчика критической ошибки адресом своего обработчика, возможно пустого, ничего не делающего. Конечно, адрес старого обработчика при этом должен быть сохранен и восстановлен перед выходом из TSR.

ПЕРЕХВАТ ПРЕРЫВАНИЯ 23h ОТ Ctrl Break необходим для предотвращения случайного нажатия этой комбинации клавиш и выхода из программы с невосстановленными векторами прерываний.

ПЕРЕКЛЮЧЕНИЕ СТЕКА. Чтобы не столкнуться с проблемой переполнения стека при "паразитировании" TSR на стеке фоновой программы, рекомендуется непосредственно перед активацией функций обработки переключить TSR на свой стек. Делается это так:

⇒предусматривается в глобальной области массив для стека, например, в 512 беззнаковых слов;

⇒при подготовке к выполнению функций обработки прерывания в отведенных для этого переменных сохраняются значения регистров SS и SP;

⇒маскируются прерывания и в регистры SS и SP записывают соответственно сегмент и смещение адреса отведенного под стек TSR массива;

⇒разрешаются прерывания;

⇒выполняется обработка;

⇒перед выходом восстанавливаются сохраненные значения
⇒регистров стека при выключенных прерываниях.

Для запрещения и разрешения прерываний в Си используют функции disable() и enable(), а в Паскале можно использовать встроенный ассемблер с командами cli, sti.

УДАЛЕНИЕ РЕЗИДЕНТА ИЗ ПАМЯТИ. Запросом на удаление может быть либо заданное слово командной строки при повторном запуске, либо нажатие обусловленной клавиши - во всех случаях осуществляются одни и те же действия по отгрузке резидента из памяти, а именно:

1. Восстановление всех векторов прерываний;
2. Закрытие всех файлов программы;
3. Освобождение занимаемой TSR памяти;
4. Возврат в прерванную программу.

При освобождении памяти следует помнить, что программа владеет двумя блоками памяти - собственно программы с сегментным адресом PSP и блоком строк среды, сегментный адрес которого хранится по смещению 2Ch в PSP.

Мы затронули далеко не все проблемы обеспечения безопасной работы TSR - остались не рассмотренными методы переключения PSP (пока мы предполагаем "паразитирование" резидента на PSP фоновой программы), использование флага повторной входимости и многое другое. Уже изложенные материалы показывают, что разработка резидентных программ для неприспособленной к этому операционной системы дело довольно хлопотное.

Мы проиллюстрируем этот факультативный раздел примером, в котором реализованы только некоторые из приведенных методов работы с резидентными программами.

Задача: составить резидентную программу, выделяющую по сигналу прерывания слово в позиции курсора на фоне работающей в это время другой программы и выводящую на экран значение, толкование или перевод этого слова на русский язык - конкретное воплощение термина "значение" нам не важно. По существу речь идет о создании резидентной программы для работы с ассоциативным словарем файлового хранения.

```
#include <io.h>
#include <dos.h>
#include <bios.h>
#include <ctype.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
```

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

/*Это пример резидентной программы, "паразитирующей на стеке" исполняющегося в данный момент процесса, что при малом объёме стека последнего, характерном для COM-программ, приводит к переполнению стека. В большинстве EXE-файлов этой проблемы не возникает - Вы можете, например, использовать этот пример при работе в IDE Borland C++.*/

```
#define KEYLENGTH      20   /*длина слова */
#define VALUELENGTH   255   /*длина толкования */
#define AltX        11520 /*код клавиши удаления резидента */
```

```
/*в этой структуре хранятся основные справочные сведения */
struct totalrec
{
  char IsDel;/*если ==0, запись пустая или помечена к удалению */
  char key[KEYLENGTH];/*слово-ключ */
  char value[VALUELENGTH];/*значение по данному ключу */
};
 /* busy установлена в 1,когда программа активна, иначе - в 0 */
char busy = 0;
char database[40];/*имя файла со словарём */
  /*для области видеопамяти, затираемой при выводе толкования */
char ScreenSaver[700];
/*количество записей в словаре, не обязательно равно числу слов */
long RecordCount;
totalrec temp;                    /*рабочая запись*/
char far *idosptr;   /*флаг повторной входимости ДОС*/
int irq;  /*флажок необходимости обработки события - нажатия F1*/

/*каскад вызываемых обработчиков перехваченных прерываний */
void interrupt (*oldReEnterable)(...),(*oldKbdVec)(...),(*installFlag)(...);

/*В этой процедуре мы проверяем словарь на корректность */
void CheckVocab()
{
  if(_argc<2)
  {
    printf("Используйте имя словаря в командной строке \n"
           "For example, %s vocab.dat\n",*_argv);
    exit(0);
  }
  /*Проверка на существование */
  FILE *fp=fopen(_argv[1],"rb");
     /*копируем во внешнюю переменную */
  strcpy(database,_argv[1]);
  if(!fp)
  { printf("Не могу открыть файл %s\n",database); exit(1);}
```

```
  /*определяем число пар слово-значение в словаре */
  RecordCount=filelength(fileno(fp))/sizeof(totalrec);
  if(!RecordCount) /*скорее всего, это не файл словаря */
  {
    printf("Файл %s пуст \n",database);
    fclose(fp);
    exit(1);
  }
   /*подсчитаем количество непустых словарных ячеек */
  for(long i=0,noempty=0;i<RecordCount;i++)
  {
    fread(&temp,sizeof(temp),1,fp);
    if(temp.IsDel)
      noempty++;
  }
  if(!noempty)  /*если все записи помечены к удалению */
  {
    printf("Файл %s пуст \n",database);
    fclose(fp);
    exit(2);
  }
  fclose(fp);/*закрываем файл */
}
```

/*Функция получения номера активной видеостраницы с помощью BIOS-ной функции определения текущего видеорежима, попутно определяющей и номер страницы */

```
inline int GetCurVPage()
{
  REGS  r;/*средство для доступа к регистрам микропроцессора
*/
  r.h.ah=0xf; int86(0x10,&r,&r);
  return r.h.bh;
}
```

/*Процедура установки курсора в заданную позицию на экране */

```
inline void SetCursorAtXY(char x,char y)
{
  REGS r;
  r.h.ah=0x2;
  r.h.bh=GetCurVPage(); /*получаем номер текущей видеостраницы
*/
  r.h.dh=y;r.h.dl=x;
  int86(0x10,&r,&r);
}
```

/*Процедура получения текущих координат курсора; x и y - псевдонимы переменных, существующих в программе или, как их ещё называют, ССЫЛКИ на них */

```
inline void GetCursorAtXY(char &x,char &y)
{
REGS r;
r.h.ah=0x3; r.h.bh=GetCurVPage();int86(0x10,&r,&r);
  y=r.h.dh; x=r.h.dl;
}

     /*Эта функция получает код символа в заданной по-
зиции экрана, корректно работая даже в графическом ре-
жиме (что BIOS вывел, то он и прочтёт...)*/

inline char GetCharAtXY(char x,char y)
{
  REGS r;
  char xs,ys;  /*текущая позиция курсора на экране */
  GetCursorAtXY(xs,ys);/*запоминаем текущую позицию курсора */
  SetCursorAtXY(x,y); /*устанавливаем курсор в заданную позицию
*/
  r.h.ah=0x8;  r.h.bh=GetCurVPage(); int86(0x10,&r,&r);
  SetCursorAtXY(xs,ys);/*восстанавливаем позицию курсора */
  return r.h.al;       /*возвращаем искомый символ */
}

   /*Функция получения количества символьных столбцов на экране
с помощью BIOS-ной функции определения текущего видеорежима
*/

inline char GetColCount()
{
  REGS r;
  r.h.ah=0xf; int86(0x10,&r,&r);
  return  r.h.ah;/*обычно  это  значение  80,  но  может
быть и другое */
}

/*Функция, получающая слово в заданной позиции на экране */
char* GetWordAtXY(char x,char y)
{
  int i=0;
/*статический массив для хранения последнего слова */
  static char __word[KEYLENGTH];
  char fch=GetCharAtXY(x,y);/*получаем букву в текущей позиции */
  if(!fch||isspace(fch)) /*в заданной позиции ничего нет */
    return 0;         /*нет слова в позиции курсора */
  /*иначе идём влево до тех пор, пока что-то есть */
  for(;x>=0&&!(fch>=0x09&&fch<=0x0D)&&fch!=0x20&&fch;
                              fch=GetCharAtXY(--x,y));
  x++;     /*первая позиция, в которой есть символ */
  for(__word[0]=fch=GetCharAtXY(x,y);
     x<GetColCount()&&!(fch>=0x09&&fch<=0x0D)
```

```
      &&fch!=0x20&&fch;__word[++i]=fch=GetCharAtXY(++x,y));
      __word[i]=0;           /*закрываем слово нулём*/
  return __word;    /*и возвращаем указатель на стати-
ческий буфер */
}

   /*переводит символ в нижний регистр, возвращая результат */
inline char rtolower(char c)
{
#pragma warn +rng /*отключаем  предупреждения компилятора */
  if((c>='A'&&c<='Z')||(c>='A'&&c<='П'))
    c+=0x20;
  if(c>='P'&&c<='Я')   c+=0x50;
  if(c=='Ё')           c='ё';
  return c;
#pragma warn -rng
}

     /*Удаление резидента из памяти */
inline void deinstall()
{                              /*запретим прерывания */
  disable();
  setvect(0x9,oldKbdVec); /*восстановление векторов */
  setvect(0x28,oldReEnterable);
  setvect(0x60,installFlag);
  /*освобождение памяти* /
  freemem(*(unsigned far*)MK_FP(_psp,0x2c));
  freemem(_psp);
  enable();                    /*разрешим прерывания */
}

   /*поиск первого слова, содержащих заданное как подстроку */
void find(char *key)
{
  for(long i=0,count=0;key[i];i++)
     key[i]=rtolower(key[i]);/*преобразуем к нижнему регистру */
  memset(&temp,0,sizeof(totalrec)); /*обнуляем рабочую запись */
  window(1,1,50,7);              /*организуем окно */
  textattr(0x0);
  gettext(1,1,50,7,ScreenSaver);/*сохраняем часть экрана */
  clrscr();            /*очищаем его чёрным цветом */
  textattr(0xa);
  FILE  *fp=fopen(database,"rb");/*открываем  файл  словаря
*/
  for(i=0;i<RecordCount;i++) /*движемся по словарю */
  {
    fread(&temp,sizeof(totalrec),1,fp);/*читаем текущую запись */
    for(long j=0;temp.key[j];j++)
      temp.key[j]=rtolower(temp.key[j]);
    if(strstr(temp.key,key)&&temp.IsDel)/*сравниваем
искомое слово с ключом текущей неудалённой записи */
```

```
    {
      cprintf("Слово - %s, значение - %s\n",temp.key,temp.value);
      count=1;              /*Нашли словарную пару! */
      break;               /*поиск  не продолжаем */
    }
  }
  fclose(fp);                          /*закрываем файл */
  if(!count)                           /*если не нашли */
    cprintf("Такого слова, как %s, в словаре нет \n",key);
  int what=bioskey(0);          /*считываем клавишу */
  puttext(1,1,50,7,ScreenSaver);      /*восстанавливаем экран */
  if(what==AltX)     /*анализ запроса на удаление */
  deinstall();      /*Выход с удалением резидента */
}

/*Процедура, работающая по нажатию клавиши F1*/
inline void WhatIsIt()
{ char xs,ys;  /*текущая позиция курсора на экране */
  GetCursorAtXY(xs,ys);/*запоминаем текущую позицию курсора */
  char *ptr=GetWordAtXY(xs,ys);/*получаем слово из-под курсора */
  if(ptr)           /*если слово под курсором есть */
    find(ptr);          /*ищем совпадения */
  else     /*иначе отделываемся звуковым сигналом */
    putch('\a');
  SetCursorAtXY(xs,ys);/*восстанавливаем позицию курсора */
}

void interrupt KeyboardInterrupt(...)
{ /*"голова" буфера клавиатуры */
  char far *t =(char far *)MK_FP(0x40,0x1a);
  oldKbdVec(...);  /*вызываем старый обработчик 9-го прерывания */
  if(*t != *(t+2))       /* если не пустой */
  {  t += *t - 30 + 5;/* перейти к позиции символа */
     if(*t==59)          /*F1==59*/
    { bioskey(0);        /* сбросить F1 */
      if(!busy)          /*если не занят */
      {busy = !busy; /*устанавливаем флаг занятости */
     /*Использование флага повторной входимости может
очень сильно замедлить реакцию на прерывание, а в мно-
гозадачной среде даже "заморозить" процесс. Тем не ме-
нее, это делается так:
         _AH=0x34;geninterrupt(0x21);
         idosptr=(char far*)MK_FP(_ES,_BX);
         for(;*idosptr!=0;) //ожидаем освобождения DOS
         { _AH=0x34; geninterrupt(0x21);
           idosptr=(char far*)MK_FP(_ES,_BX);
         } */
          irq=1;    /*отложенный запрос для обработчика 28-го пре-
     рывания */
        busy = !busy; /*сбрасываем флаг занятости */
      }
```

```
    }
  }
}

     /*Это прерывание генерируется только тогда, когда
ДОС находится в реентерабельной (допускающей повторное
вхождение) секции, что позволяет вызывать её функции
[в частности, для работы с файлами]*/

void interrupt IAmFree(...)
{
  if(irq)
  {
    WhatIsIt();/*получение толкования слова в позиции курсора */
    irq=0;           /*до следующего нажатия на F1 */
  }
  (*oldReEnterable)(...);/*каскадный вызов старого обработчика */
}

void main()
{
  if(*(int*)(0x60*4)==170836)     /* признак инсталлированности* /
  { printf("Программа уже инсталлирована \n"); return; }
  irq=directvideo=0;    /*вывод на экран только средствами BIOS*/
  CheckVocab();       /*проверка словаря на корректность */
  /*получаем старые обработчики перехватываемых прерываний */
  oldKbdVec=getvect(0x9);
  oldReEnterable=getvect(0x28);
  installFlag=getvect(0x60);
  unsigned size=2000; /* размер резидентной части в параграфах. Для COM-программ этот размер можно определить точно: _SS+(_SP+15)/16-_psp; , а для EXE это обычно размер файла, делённый на 16 нацело с округлением в большую сторону. Затем это значение уменьшается до тех пор, пока программа не перестанет работать: искомым будет предпоследнее значение */

  disable();                  /*отключаем прерывания */
  setvect(0x9,KeyboardInterrupt);/*устанавливаем новые векторы */
  setvect(0x28,IAmFree);
  *(int far*)(0x60*4)=170836;    /*отмечаем признак инсталляции* /
  enable();                   /*разрешаем прерывания */
  keep(0,size);          /*и завершаемся резидентно */
}
```

7. Описание языка (Турбо, Борланд) Паскаль.

7.1. Общая структура текста программы на Паскале.

Комментарии к паскалевской программе следует размещать в фигурных скобках:

{Это комментарий в синтаксисе языка Паскаль}

Альтернативная форма комментария - круглые скобки со звёздочками внутри:

(*Это тоже комментарий в синтаксисе языка Паскаль*)

Комментарии разных типов могут быть вложенными:

{...(*...*)...}

В таких же скобках размещаются предваряемые знаком $ директивы компилятору:

{$R-} - отключить проверку выхода индексов массивов за допустимые пределы.

Паскаль-программа состоит из главной программы и вспомогательных подпрограмм, выполняющих специализированные задачи.

На Паскале главная процедура не имеет имени, параметров и локальных данных. Ее тело размещается между словами begin и end. (с точкой). Поэтому ничего не делающая Паскаль - программа выглядит совсем просто:

```
procedure proc1 begin end; {Ничего не делающая процедура}
procedure proc2 begin end; {Ничего не делающая процедура}
          { главная программа}
          begin
          proc1;proc2; {вызовы вспомогательных процедур}
          end.
```

7.2. Скалярные (одноэлементные) типы данных в Паскале.

Чтобы разместить любой объект в памяти для обработки, необходимо объявить его тип, чтобы тем самым сообщить компилятору о размере необходимой для этого объекта памяти и допустимых операциях с ним.

Ассортимент скалярных типов в Паскале обширен в основном за счет включения в язык совершенно одинаковых по

содержанию, но "несовместимых " друг с другом типов. Например, типы char и byte - оба хранят целые беззнаковые однобайтовые числа, но не могут участвовать вместе в операциях присваивания, так как char хранит целочисленные коды символов, а byte - просто целые числа в том же диапазоне. Создателям языка казалось, что это способствует более эффективному и быстрому освоению программирования, а по моему опыту - только запутывает обучаемого, формируя искаженные представления о способах представления данных в памяти компьютера.

Итак, скалярные типы:

Числовые:

Содержание	Обозначение	Диапазон
1-байтовое целое без знака	Byte	(0..255);
1-байтовое целое со знаком	ShortInt	(-128..127);
2-байтовое целое без знака	Word	(0..65535);
2-байтовое целое со знаком	Integer	(-32768..32767);
4-байтовое длинное целое	LongInt	(-2147483648.. 2147483647);
4-байтовое вещественное	Single	(1.5*10^-45..3.4*10^38);
6-байтовое вещественное	Real	(2.9*10^-39..1.7*10^38);
8-байтовое вещественное	Double	(5.0*10^-324 ..1.7*10^308);
10-байтовое вещественное	Extended	(3.4*10^-4951 ..1.1*10^4932);

Логические типы	Boolean (True,False)
Символьный тип	Char (любой символ из набора ASCII)
Строковый тип	String, String[n]
Указатель	Pointer
Перечисление	
Диапазон	

Размер типов "указатель" (Pointer) и "символ" (Char) равен 4 и 1 байту соответственно. Размер логического типа (Boolean) является плавающим; при компиляции программы компилятор выполняет автоматическое преобразование типа Boolean в один из следующих типов: ByteBool, WordBool и

LongBool длиной в 1, 2 и 4 байта соответственно. В связи с этим любой целый скалярный тип может быть преобразован к логическому, при этом ноль и нулевой указатель переходят в false, а все остальные значения - в true.

Синтаксис Паскаля требует при размещении в памяти переменной любого типа делать это всегда в разделе программы по имени Var с указанием имени или списка имен и последующим указанием через двоеточие типа (Паскаль не чувствителен к регистру букв - можно писать большими и малыми - они считаются одинаковыми):

```
var
i1,i2,i3:Integer;
b1,b2,b3:Boolean;
s1,s2,s3:String[123;
p1,p2: Pointer;
```

Присвоение начальных значений при описании переменных не допускается. При необходимости это сделать приходится объявлять переменную в разделе констант (это будет типизированная псевдоконстанта, на самом деле - переменная со стартовым значением) с указанием типа и значения:

```
const
i1:Integer=12; b1:Boolean=True; r1:Real=12.546;
```

Непосредственные числовые значения целых задаются в 10-тичной или 16-ричной системах счисления, в последнем случае перед значением ставится символ $:

```
h1:= $a53f;
```

ШАБЛОН ПЕРЕЧИСЛЕНИЙ необходимо вначале описать в разделе Type, а затем в разделе Var описать переменную этого типа:

```
type
fruit= (груша, яблоко , слива);
var
f:fruit;
```

После этого где-то в процедуре можно использовать:

```
f:=груша;
```

Нам с вами не разрешается присваивать значения элементам перечисления, это делает компилятор (нумеруя элементы с 0 с шагом 1), а мы можем получить соответствующие значения с помощью библиотечной функции Ord (имя элемента перечисления), возвращающей длинное целое. Максимальное количество элементов в одном перечислении - 65535. Имена элементов не могут повторяться в пределах одной программы.

Значения из перечислимого типа не могут непосредственно участвовать в операциях ввода-вывода - их нельзя вывести на экран или принтер или явно ввести с клавиатуры.

ТИП ДИАПАЗОН позволяет определить тип переменных, которые могут принимать значения только в ограниченном поддиапазоне некоторого базового типа - целочисленного, символьного или любого введенного нами перечислимого типа. Для введения типа ДИАПАЗОН надо в блоке `TYPE` указать его имя и границы через 2 точки:

```
type
Centuty = 1..20;
CapsLetter = 'A'..'Z';
```

Операции ввода - вывода допустимы только с переменными, для которых диапазон взят из "выводимого" базового типа.

7.3. Простые константы.

Простые константы-значения компилятор Паскаля распознает по синтаксису их записи, а именованные константы объявляются в разделах `const` программы или подпрограмм без указания типа:

```
const Min=0; Max=500; e=2.7;
```

Значения символьных и строковых констант размещаются в одиночных кавычках

```
SpecChar='\';
HelpStr='Нажмите клавишу F1';
NoAdress=nil;{Адресная константа со значением
```

отсутствующего адреса `nil`. Того же самого эффекта вы можете
достичь, инициализировав NoAdress значением `Pointer(0)` или присвоив этой переменной `Ptr(0,0)}`

Альтернативной формой представления значения символов есть их запись через значение числового кода, предваряемого символом #:

```
#97 (символ a)
#0 (нуль-символ)
#32 (пробел)
#$20 (тоже пробел)
Ok=True; {логическая константа}
```

Управляющие символы с кодами от 0 до 31 могут быть представлены их "клавиатурными" обозначениями - значком ^ и буквой алфавита с тем же номером (для диапазона кодов 1..26) или служебным знаком (в диапазоне 27..31):

^A - код 1
^B - код 2
...
^Z - код 26
^[- код 27

в том числе ^G - звонок, ^I - TAB, ^J - код 10 - перевод строки, ^M - код 13 - возврат каретки

```
Alpha=['A'..'Z']; {константа типа диапазон}
```

При определении значений констант можно использовать выражения из чисел, некоторых функций языка и ранее определенных простых констант:

```
const
 Interval=Max-Min+1;
 Key=Chr(27);
 e2=e*e;
 BigHelpStr=HelpStr + 'для подсказки';
```

Непосредственные числовые значения целых задаются в 10-тичной или 16-ричной системах счисления, в последнем случае перед значением ставится символ $:

```
const
 h1= $a53f;
 Flag=Ptr($0000,$00F0);{адресная константа}
```

В выражениях могут использоваться все арифметические операции, поразрядные операции, логические операции, операции отношения и следующие функции:

Abs, Round, Trunc, Chr, Ord, Pred, Succ, Odd, SizeOf, Length, Ptr, Lo, Hi, Swap

7.4. Операции над простыми типами.

7.4.1. Операции над целочисленными типами

Операция	Обозначение
Сложение	+
Вычитание	-
Умножение	*
Деление (получение частного)	div
Деление (получение остатка)	mod

Побитовое И	and
Побитовое ИЛИ	or
Побитовое исключающее ИЛИ	xor
Инверсия битов	not
Сдвиг влево	shl
Сдвиг вправо	shr

Присвоение значения	=

7.4.2. Операции с вещественными числовыми типами

Операция	Обозначение
Сложение	+
Вычитание	-
Умножение	*
Деление (получение частного)	div
Возведение в степень	----------

7.4.3. Операции с любыми числовыми типами

Операция	Обозначение
Получение размера типа или объекта	sizeof
Операции сравнения на	
больше	>
меньше	<
равно	=
не равно	<>
больше или равно	>=
меньше или равно	<=

Переменные целых типов можно инкрементировать и декрементировать с помощью библиотечных процедур INC и DEC. Первым параметром в них всегда является имя переменной, второй определяет, на сколько она будет наращиваться (уменьшаться) и является необязательным (по умолчанию равен 1).

7.4.4. Операции над символами в Паскале.

Символы в Паскале (элементы типа char) можно лишь присваивать и сравнивать друг с другом - они равны, если рав-

ны их коды и большим считается тот, у которого больше кодовое число.

7.4.5. Операции над строками.

Операция	Обозначение
Пересылка строк между областями памяти (присвоение)	:=
Сравнение строк	> < <> =
Конкатенация (сцепление) строк	+

7.4.6. Операции над адресами

Паскаль для адресного типа указателей поддерживает только операции присвоения и сравнения, инкремента и декремента. Операции сложения и вычитания адресов определены только для двух указательных типов : PChar и PByte. Вычтя один указатель из другого, вы получаете расстояние между ними в оперативной памяти, прибавляя (вычитая) к указателю на символ или на байт число, вы создаёте временный указатель, смещённый в памяти на соответствующее значение байт.

Операция получения адреса расположенного в памяти объекта обозначается в Паскале значком @имя_объекта. Возвращаемый операцией результат размещается в переменных типа указатель.

Для изменения указателя на любой тип следует пользоваться процедурами INC и DEC. При этом указатель наращивается не на один байт, а на длину типа. Например,

```
type
     PAnyType=^AnyType;
     {объявление типа "указатель на тип AnyType"}
var
 at,bt:AnyType; (*переменная*)
 pat:PAnyType;(*указатель*)
begin
 pat:=@at; {pat содержит адрес at (указывает на него)}
 INC(pat); {pat содержит адрес at, увеличенный на sizeof(at), т.е. на sizeof(AnyType)}
```

Разадресовать указатель - значит получить значение переменной, на которую он указывает (значение по заданному адресу). Например, запишем в новую переменную bt значение at, используя указатель pat, хранящий её адрес:

```
 bt:=pat^;
```

Для работы с абсолютными адресами в Паскале существует директива absolute :

```
type
ByteArray = Array [1..2000] of Byte;
var
    Memory: Byte absolute $0000:$0417;
    VidMem: ByteArray absolute $B800:$0000;
```

Однобайтовая переменная Memory займет указанное адресом место в системной области и даст возможность получать и изменять значение по этому адресу, а массив VidMem совместится с видеобуфером, что даст возможность читать и изменять лежащие там коды символов.

7.5.Сложные типы данных - массивы.

Массив в алгоритмических языках - это обозначенная одним именем совокупность элементов одного и того же типа. Таким образом, массив - это составной, агрегированный тип, он состоит из определенного количества объектов другого типа. Элементами массивов в алгоритмических языках могут быть как простые, так и сложные, составные типы: например, массивы целых или вещественных чисел, массивы строк, массивы массивов (2-мерные или многомерные массивы любых элементов), массивы записей (мы их рассмотрим несколько позже), массивы указателей и т д. Одномерные и двумерные массивы встречаются довольно часто, 3-мерные массивы - это уже почти экзотика. Массивы числом измерений более 3 объявить несложно, надо только помнить о темпах расходования памяти.

Тип "массив" определяется в Паскале конструкцией

```
Array [диапазон] of ТипЭлементов
```

Диапазон в квадратных скобках определяет значения индексов первого и последнего элемента, так что при объявлении типов

```
type
A1 = Array [1..10] of Real;
A2 = Array [11..20] of Real;
```

мы вводим 2 различных типа, отличающихся способом нумерации своих элементов.

Доступ к i-му элементу осуществляется через запись индекса в квадратных скобках, а попытка обратиться к элементу с индексом, не соответствующим диапазону в описании типа, вызовет ошибку компилятора, если при компиляции задан ключ {$R+}. Чтобы отменить проверку выхода за диапазон, устанавливают {$R-} - обычно после завершения отладки програм-

мы, чтобы удалить из нее излишние фрагменты кода, связанные с такой проверкой.

Пример описания массива в Паскале:

```
var
V1:Array[1..31] of Real;
```

Диапазон числовых индексов должен умещаться максимум в типе Word, а длина массива в байтах не должна превышать 65520 байт. Диапазон может задаваться любым перечислимым типом, как встроенным, так и вводимым нами, причем если диапазон соответствует всему типу, то можно указывать просто имя перечислимого типа:

```
type
MT = (January, February, March, April, May);
CT = Array [MT] of Word;
ST = Array [March..May] of Word;
var
C:CT; S:ST;
A1: Array ['A'..'Z'] of char;
B1:Array [Boolean] of Byte;
```

Можно объявлять массивы массивов:

```
type
VectorType=Array[1..3] of Real;
MatrixType=Array[1..10] of VectorType;
```

2-мерный массив можно описать и по-другому:

```
type
MatrixType = Array[1..10] of Array [1..3] of Real;
```

или MatrixType = Array[1..10,1..3] of Real;

Каждое измерение массива совершенно не зависит от остальных и можно объявлять массивы с различными индексами:

```
var
M:Array[-10..0,'A'..'Z'] of Byte;
```

При обращении к элементам массива индексы можно задавать как через запятую, так и в отдельных квадратных скобках:

M[-3,'B',True] эквивалентно M[-3]['B'][True]

Компонентом массива может быть любой тип. В памяти элементы массива хранятся так, что быстрее всего изменяется самый дальний индекс, в частности 2- мерные массивы - построчно.

К 2-м совместимым по типам массивам применима операция присваивания:

```
A:=B;
```
которая копирует поэлементно массив B в массив A.

Это свойство можно, например использовать для сохранения текстового экрана для последующего восстановления:

```
type
 S=Array[1..2000] of Byte;
var
 S1:S;
 S2:S absolute $b800:$0000;
где-то в программе
begin
 S1:=S2; {сохраняем экран}
 ... что-то выводим
 ... делаем еще что-то
 S2:=S1; {восстанавливаем}
end;
```

Если длина массива заранее неизвестна, бывает удобно при отключенной проверке выхода за границы массива объявить одноэлементный массив и указатель на него, под который затем выделить память:

```
type
 OneElementArray=Array [0..0] of SomeThing;
 POneElementArray=^OneElementArray;
var
 obj:POneElementArray;
{выделение памяти под obj
 ...
}
{$R-}
obj^[i]:=0; (*пример индексации - вначале указатель разадресуется, превращаясь в массив по данному в указателе адресу, а затем этот массив индексируется*)
```

Заметим, что опция компилятора для отключения проверки выхода за пределы массива действует только для переменных, и обращение вида obj^[1] вызовет ошибку времени выполнения. Для того, чтобы избежать такой ситуации, присвойте это фиксированной значение индекса какой-либо целочисленной переменной и используйте эту переменную для обращения к соответствующему элементу массива.

7.6. Сложные типы - записи.

Тип "запись" (другое название - "структура") служит для объединения "под одной крышей" возможно разнотипных эле-

ментов данных, взаимосвязанных по смыслу или по характеру обработки. Запись определяется конструкцией:

```
RECORD
    Поле1 : ТипПоля1;
    Поле2 : ТипПоля2;
    ...
    ПолеN : ТипПоляN;
END;
```

Поле записи может иметь любой Паскалевский тип данных - массив, другая запись, множество. Доступ к полям осуществляется через имя переменной и имя поля, разделяемых точкой.

```
type
PointRecType = RECORD x,y :Integer END;
var
Point:PointRecType;
begin
    ...
    Point.x = 123;
    ...
end;
```

Независимо от количества объявленных переменных типа "запись", поля каждой переменной называются одинаково, в соответствии с шаблоном. Поскольку имена полей "скрыты" внутри типа, они могут совпадать с именами "внешних" переменных и поля в других описаниях записей, например:

```
type PointRecType = RECORD
X,Y : Integer
END;
ColorPointRecType = RECPRD
X,Y:Integer;
Color:Word
END;
var X,Y :Integer; Point :PointRecType;
ColorPoint :ColorPointRecType;
```

В программе X, Point.X, ColorPoint.X - совершенно разные значения.

Записи в Паскале могут иметь так называемую вариантную часть. Это означает, что в пределах одного типа можно задать несколько различных структур. Непосредственный выбор одной из структур будет определяться контекстом или каким - либо сигнальным значением. Вариантные поля в описании шаблона записи указываются после того, как перечислены все поля фиксированного типа и оформляются особым образом:

```
TYPE
VRecType = RECORD          {Тип записи с вариантами}
Number :Byte;
case Measure : Char of     {Признак единиц измерения длины}
'д','Д':(INChes : Word);   {В дюймах}
'с','С':(cantimeters :LongInt); {В сантиметрах}
'?':(Comment1,Comment2 : String[16]) {Тексты - комментарии}
END;
```

В примере записи есть обычное фиксированное поле Number, и поле-селектор Measure, обрамленное словами CASE и OF, после чего идет перечисление вариантов 3-го поля в круглых скобках. Содержимое поля-селектора определяет, какой именно вариант будет выбран при работе с записью. В текущий момент доступны поля только одного из возможных вариантов, в зависимости от значения, присвоенного полю - селектору. Все варианты располагаются в одном и том же месте памяти, а размер этого места определяется самым объемным вариантом - в нашем случае 2*(16+1) байтов для поля String[16].

Поле-селектор можно игнорировать, обращаясь к любому из полей вариантов - одно и то же значение будет трактоваться в соответствии с типом поля. Здесь всегда таится возможность ошибок, которые никак не диагностируются, но если не ошибаться, то можно с удобством использовать вариантные записи для хранения различных вариантов например в нашем случае единиц измерения длины.

Если явный указатель варианта не нужен, его можно заменить именем любого перечислимого типа, имеющего достаточное количество вариантов :

```
TYPE
VRecType = RECORD
Number :Byte;
case Byte of
1:(INChes : Word);     {В дюймах}
2:(cantimeters :LongInt); {В сантиметрах}
3:Comment1,Comment2 : String[16]) {Тексты - комментарии}
END;
```

Значения констант в описании в этом случае - чистая условность и могут быть любыми неповторяющимися в пределах типа Byte. При отказе от поля - селектора во время работы программы уже нельзя определить, какой вариант должен использоваться в данный момент и так обычно работают при

необходимости разнотипного представления одних и тех же данных:

```
TYPE
CharArrayType = Array[1..4] of char;
VAR
V4:RECORD
case Boolean of
True : (C:CharArrayType);
False: (B1,B2,B3,B4 :Byte)
END;
```

Поля B1..B4 позволяют работать с ASCII - кодами символов, а поля C[1]..C[4] - с символами, не прибегая к явному преобразованию типов.

Переменные типа "запись" могут участвовать только в операциях присваивания, а их поля - в любых дозволенных для их типа операциях.

Для облегчения работы с полями записей в Паскале введен оператор присоединения WITH со следующим синтаксисом использования:

WITH ИмяПеременной_Записи DO Оператор;

Внутри оператора обращение к полям уже производится без указания имени переменной:

```
VAR
DemoRec : RECORD X,Y :Integer END;
WITH DemoRec DO
begin
 X:=0;Y:=129;
End;
```

При использовании внутри WITH "незаписных" переменных необходимо следить, чтобы их имена не совпадали с "записными". При необходимости "развязки" внутри WITH к совпадающим внешним именам необходимо приписать впереди через точку имя программы или модуля в которых они описаны:

```
PROGRAM MAIN;
VAR X,Y :Integer;
RecXY :RECORD X,Y :Integer END;
begin
WITH RecXY DO begin
X:=3.14*Main.X;
Y:=3.14*Main.Y end;
```

```
end;
```

Если одно из полей записи - тоже запись, можно распространить оператор присоединения на несколько полей вглубь, перечислив их через запятую, но внутри тела оператора можно будет обращаться только к последним полям:

```
WITH ИмяЗаписи, Поле_Запись DO
begin
 Обращения к именам полей Поля_Запись
end;
```

7.7. Сложные типы - множества.

Говоря "множество", всегда необходимо указывать "чего?". В Паскале множества - это наборы значений скалярного перечислимого типа не более чем из 256 элементов. Это, например, встроенные перечислимые типы - целочисленные `Byte`, символьный `Char` или созданный нами скалярный тип, например:

```
TYPE
VAType = (MDA, Hercules, CGA, EGA, VGA, Other, NotDetected);
VAR
VideoAdapter :VAType;
```

и далее можно ввести переменную - множество перечисленных в типе значений.

В описании множества как типа используется конструкция

```
Set of
```

с последующим указанием базового типа - скалярного типа, из элементов которого составляется множество.

Есть несколько способов задания множеств:

```
TYPE
SetChar = Set of Char;
SetByte = Set of Byte;
SetDigit = Set of 0..9;
SetDChar = Set of '0'..'9';
SetVA = Set of Cga..Vga;
```

В задании типа множества можно "урезать" базовый тип, задавая поддиапазон его значений. Если перечислимый тип вводится только для подстановки его имени `Set of`, то можно перечислять значения сразу в конструкции `Set of`, не забывая круглые скобки:

```
TYPE
```

SetVideo = Set of (Mda, Hercules, Cga, Ega, Vga, Other, NotDetected);

Можно опустить фазу задания типа в разделе TYPE и сразу задавать его в разделе VAR:

```
VAR
V1 : Set of ...
```

Каждый элемент множества имеет сопоставимый номер - для типа Byte это значение числа, для Char - это код символов, нумерация идет всегда от 0 до 255. Множества имеют весьма компактное машинное представление - 1 бит на 1 элемент, поэтому для хранения 256 элементов расходуется 32 байта. Переменные типа множество подчиняется специальному синтаксису. Элементы этой переменной должны заключаться в квадратные скобки:

```
SByte :=[1,2,3,4,10,20,30,40];
SChar :=['a','b','c'];
Sdiap:= [1..4];
Empty:=[];        {Пустое множество}
```

Порядок следования элементов внутри скобок не имеет значения, число повторений - тоже, храниться будет одно значение. В качестве элементов множеств в скобки могут включаться константы, переменные соответствующих базовых типов и выражения с совпадающим типом результата.

К множествам применимы операции сравнения =, <>, <=, >= и операции, создающие производные множества:

in - проверка вхождения элемента в множество;

+ - объединение множеств;

- - разность множеств;

* - пересечение множеств.

Операция изъятия элемента из множества в Паскале отсутствует.

7.8. Управляющие структуры.

Мы уже обсуждали управляющие структуры при освоении словесных описаний алгоритмов и блок - схем. К управляющим структурам, требующим специального синтаксиса записи, относятся условное выполнение одного или группы операторов

(ветвление) и циклическое (повторяющееся) выполнение одного или группы операторов.

Различают двух и многоальтернативные ветвления, их синтаксис будет рассмотрен для конкретных исполнителей.

Различают также несколько видов циклов:

-циклы с предусловием, работающие по схеме

ПОКА УсловиеВыполняется ВыполнятьБлокОператоров

-циклы с постусловием, реализующие схему

ВыполнятьБлокОператоров ПОКА УсловиеВыполняется

Отличия их очевидны: цикл с предусловием может не выполниться ни разу, если не выполняется предварительно проверяемое условие; цикл с постусловием всегда выполнится хотя бы 1 раз, т.к. проверка условия вынесена в конец цикла.

-бесконечные циклы, в которых условие прекращения повторения блока операторов формируется непосредственно в блоке операторов, а выход из цикла осуществляется с помощью специальных операторов или подпрограмм (break в C и Паскале); таким образом, бесконечный цикл реализует схему:

ВЫПОЛНЯТЬ_БЕСКОНЕЧНО БлокОператоров

-циклы с фиксированным количеством итераций, представляют собой частный случай циклов с предусловием и дополнительном формировании счетчика "оборотов"; работают они так: в специальную переменную-счетчик заносится некоторое начальное положительное или отрицательное значение, сравнивается с заданным числом и принимается решение о выполнении блока операторов; после каждого прохода блока операторов это число уменьшается (или увеличивается) на заданный шаг. Выход из цикла осуществляется, когда счетчик станет больше (или меньше) заданного числа. Циклы этого типа в различных инструментальных системах имеют различные модификации схем функционирования.

7.8.1. Условные операторы в Паскале.

Условный оператор в Паскале имеет следующую структуру:

if Условие then БлокОператоров1 else БлокОператоров2;

и, как и в других языках, служит для организации процесса вычислений в зависимости от какого - либо логического условия. Например:

```
if x>5 then begin x:=x+5;y:=1 end else y:=-1;
```

В Паскале между ключевыми словами не ставятся точки с запятой, но в конце всего оператора она конечно обязательна, чтобы отделить условный оператор от следующих за ним по тексту. Альтернативная ветвь else может отсутствовать, в таком операторе при невыполнении условия ничего не делается, а осуществляется переход к следующему оператору.

Условные операторы могут быть вложенными друг в друга - при этом важно не запутаться в вариантах сочетаний условий, надо помнить, что альтернатива else всегда принадлежит ближайшему if без else. Так, в фрагменте

```
if Условие1 then if Условие2 then Оператор1 else Оператор2;
```

Оператор2 относится к условию2 и если необходимо отнести его к условию1, то следует использовать скобочные слова begin - end:

```
if Условие1 then
begin if Условие2 then Оператор1 end else Оператор2;
```

Многовариантный оператор ветвления в Паскале имеет структуру:

```
CASE УправляющаяПеременнаяИлиВыражение OF
СписокЗначений1 : БлокОператоров1;
...
СписокЗначенийN : БлокОператоровN
ELSE АльтернативныйБлокОператоров
END;
```

Тип управляющей переменной или значение выражения может быть только перечислимым, диапазоном или целочисленным Byte, Shortint, Integer, Word. Ветвь ELSE - необязательна.

```
CASE Err OF
0       : Writeln('Нормальное завершение');
```

```
2,4,6 : Writeln('Ошибка выполнения файловых операций');
7..99 : Writeln('Ошибка кодирования')
ELSE Writeln('Неопознанная ошибка')
END;
```

7.8.2. Циклы в Паскале.

7.8.2.1. Цикл с предусловием в Паскале:

WHILE Условие DO БлокОператоров

Цикл выполняется, пока условие истинно.

Пример:

```
VAR Factorial,N :Integer;
begin Factjrial:=1; N:=1;
WHILE N<=10 DO begin
Factorial:=Factorial*N; N:=N+1
end;
Writeln(Factorial);
end.
```

7.8.2.2. Цикл с постусловием:

REPEAT БлокОператоров UNTIL Условие

Цикл REPEAT - UNTIL выполняется в Паскале, пока условие ложно. Если в качестве условия поставить логическую константу False, цикл превращается в бесконечный и выйти из него можно будет только оператором выхода из подпрограммы Exit или оператором завершения программы в целом Halt. В 7-й версии Паскаля появились подпрограммы выхода из циклов Break и досрочного перехода на проверку условия Continue, аналогичные соответствующим операторам языка Си.

7.8.2.3. Цикл FOR... DO.

В Паскале он имеет ограниченные возможности простого цикла с параметром - счетчиком со следующим синтаксисом:

FOR Параметр:= МладшееЗнач TO СтаршееЗнач DO БлокОператоров;

или

FOR Параметр:= СтаршееЗнач DOWNTO МладшееЗнач DO БлокОператоров;

Шаг наращивания счетчика в Паскале "нерегулируемый" и равен +1 при TO или -1 при DOWNTO. Циклы FOR с вещественными параметрами в Паскале невозможны - их приходится организовывать через цикл while.

7.8.3. Процедуры break и continue в теле циклов.

Если циклы организованы как бесконечные, например с константами в условиях циклов WHILE true DO и REPEAT - UNTIL false, то выход из них в младших версиях Паскаля был крайне неудобен. В 7-й версии этот недостаток устранен с помощью процедуры break "насильственного" выхода за пределы тела цикла по аналогии с таким же оператором C.

Другая модернизация языка - процедура continue - позволяет из любой точки тела цикла перейти к анализу необходимости его продолжения (проверке условия между WHILE и DO или после UNTIL), оставив невыполненной всю последовательность операторов, стоящих после вызова этой процедуры.

7.9. Подпрограммы.

7.9.1. Разновидности подпрограмм, способ определения.

Подпрограммы - это языковая реализация вспомогательных алгоритмов, на которые как правило распадается решение общей задачи в процедурном программировании. В самом деле, даже решение достаточно узкого класса задач на компьютере, например, вычисление числа Фибоначчи с произвольным номером, потребует от нас выполнения ряда вспомогательных алгоритмов.

В частности основная программа должна узнать, какой номер числа Фибоначчи интересует пользователя нашей программы. Можно ориентировать нашу программу на диалог с пользователем по этому поводу, то есть вывести на экран запрос типа : "Введите номер интересующего вас числа Фибоначчи" и принимать ввод пользователя с запоминанием введенного номера. Можно ориентировать программу на ввод пользовате-

лем номера числа сразу после имени программы в командной строке.

В первом случае нам понадобится подпрограмма (вспомогательный алгоритм) диалога, во втором - подпрограмма извлечения заданного номера из префикса программного сегмента. В свою очередь подпрограмме диалога могут понадобиться вспомогательные алгоритмы сохранения, очистки, восстановления содержимого экрана, чтобы не вести экранный ввод - вывод на фоне предыдущих записей, алгоритмы установки курсора для эстетического оформления вопросов и ответов в диалоге и т д.

Таким образом, решение общей задачи даже в простейших случаях представляет собой многоступенчатый процесс расчленения на все более простые действия. Конечно, можно составить весь алгоритм без явного выделения вспомогательных действий, но такую программу будет не только неудобно составлять и отлаживать, ее будет чрезвычайно сложно читать и практически невозможно совершенствовать.

Другой причиной необходимости подпрограмм в алгоритмических языках является наличие в разных программах большого количества однотипных, шаблонных действий - прием ввода пользователя с клавиатуры, вывод строк на экран или печатающее устройство, сортировки чисел по возрастанию в числовых массивах или сортировка списков фамилий по алфавиту, различные алгоритмы поиска необходимых данных в массивах - этот перечень стереотипных работ можно продолжать долго. Совершенно очевидна целесообразность однократного составления таких шаблонных, часто употребляемых вспомогательных алгоритмов, хранение их на диске и последующее использование в самых разнообразных задачах. Но тогда программа должна будет состоять из главной подпрограммы (она нужна для определения начала и конца общего алгоритма), которая что-то делает сама, а что-то поручает вызываемым на выполнение вспомогательным подпрограммам, которые, в свою очередь, делают что-то сами и, возможно, вызывают на выполнение другие вспомогательные подпрограммы и т.д.

Каждая вспомогательная подпрограмма должна быть достаточно универсальной, т.е. выполнять свою работу для различных вариантов данных, а, следовательно, при ее вызове на выполнение ей необходимо будет передать данные или их адреса для обработки. После завершения подпрограмма должна возвратить управление в точку вызова - точнее на оператор, следующий за вызовом подпрограммы. Поэтому механизм ра-

боты с подпрограммами требует запоминания точки ухода в подпрограмму и возврата из нее.

В Паскале поддерживается 2 вида подпрограмм - процедуры и функции.

Определение процедуры начинается с ключевого слова PROCEDURE, за которым следует ее имя со списком параметров в круглых скобках, если таковые необходимо передавать процедуре при ее вызове на выполнение. После такого заголовка следуют при необходимости принятые в Паскале разделы объявлений LABEL, CONST, TYPE, VAR и далее тело процедуры, обрамленное словами begin и end;

```
PROCEDURE ИмяПроцедуры(Парам1 : ТипПарам1;
                       Парам2 : ТипПарам2;
                  VAR Парам3 : ТипПарам3;
                  VAR Парам4 : ТипПарам4);
LABEL
 Перечисление локальных меток процедуры
CONST
 Описание локальных констант
TYPE
 Описание локальных типов
VAR
 Описание локальных переменных и вложенных подпрограмм-процедур и/или функций.
BEGIN
 Тело процедуры
END;
```

Модификатор VAR в списке параметров перед 3-м и 4-м параметрами указывает, что процедуре при вызове на выполнение будут передаваться не значения параметров, а ссылка на существующую переменную и процедура в этом случае может изменять значения такой переменной. При передаче параметров по значению в стек заносится копия значения и все изменения остаются внутри процедуры, не затрагивая область памяти самой переменной, переданной в процедуру по значению.

Структура определения функций совпадает с процедурами, кроме заголовка он содержит ключевое слово FUNCTION и дополнительно через двоеточие тип возвращаемого функцией значения - скалярного:

```
FUNCTION ИмяФункции (Список параметров) :ИмяСкалярногоТипа;
```

Под именем функции скрывается при вызове на выполнение скалярное значение возвращаемого ею типа и имя функции вместе со списком своих параметров может использоваться в выражениях наравне с переменными.

В младших версиях Турбо Паскаля вызов функции на выполнение был возможен с обязательным приемом в вызывающей подпрограмме возвращаемого значения, например, в правой части оператора присваивания. Начиная с 7-й версии это требование снято - функцию можно вызвать на выполнение и без использования возвращаемого значения, как процедуру. Если бы язык дополнили еще "пустым" типом, аналогичным типу void в C, то можно было бы обходиться одним типом подпрограмм - функциями, но этот последний шаг так и не был сделан.

Глобальными называют те константы, типы и переменные, которые описаны вне процедур или функций, а локальные - объявленные либо в списке параметров (переменные), либо в разделах подпрограмм. Чтобы подпрограмма могла работать с глобальными объектами, их описание должно предшествовать определению подпрограммы.

Процедуры и функции в Паскале могут быть вложенными друг в друга с большим числом уровней вложенности, хотя на практике он не превышает 2. Вложенная подпрограмма относится к охватывающей ее подпрограмме так же как подпрограмма к основной программе. Вложенные подпрограммы могут вызываться только внутри охватывающей, их переменные - локальны, но внутри вложенной подпрограммы "видны" все объекты охватывающей подпрограммы и все по настоящему глобальные.

Область действия меток всегда локальна, но вам следовало бы составлять программы так, чтобы ни метки, ни операторы GOTO никогда не понадобились.

7.9.2. Опережающие объявления подпрограмм.

Паскалевский компилятор транслирует программный текст последовательно сверху вниз, не осуществляя предварительного полного просмотра с составлением таблиц идентификаторов, как это делает, например, компилятор Си. Поэтому все программные объекты - типы, константы, переменные и подпрограммы должны в Паскале быть описаны до их использования. В противном случае компилятор будет считать их неизвестными и вам придется перемещать тексты определения подпрограмм вверх по тексту. Чтобы избежать этого и иметь возможность со-

ставлять текст программы с подпрограммами не "задом наперед", а в естественном порядке разработки от главной программы к подпрограммам вниз по тексту, используют опережающие объявления подпрограмм директивой FORWARD:

```
PROCEDURE ИмяПроцедуры(параметры); FORWARD;
FUNCTION ИмяФункции(параметры) :ИмяТипа;FORWARD;
          ....
PROCEDURE ИмяПроцедуры; {список параметров уже не нужен}
...
begin
 тело процедуры
end;

FUNCTION ИмяФункции; {достаточно имени}
Тело функции
...
```

Директива FORWARD помогает также развязать закольцованные вызовы, когда 2 подпрограммы вызывают друг друга и их взаимное расположение не может решить проблему предварительного определения до использования.

7.9.3. Объявление внешних подпрограмм.

Сам Паскаль - не слишком коммуникабельный язык, он не поддерживает генерацию объектных файлов в формате OBJ и не поставляет написанные на нем подпрограммы в программы на других языках. Но использовать при компиляции и компоновке внешние подпрограммы в виде OBJ - файлов, созданных другими трансляторами, он не отказывается, если удовлетворяются при этом требования к используемой модели памяти и способу передачи параметров.

Команды подстыковки объектных файлов в Паскаль - программу задаются директивами компилятора {$L ИмяФайла.obj} в подходящих местах программы. Реализованные в этих файлах подпрограммы должны быть объявлены специальным словом EXTERNAL.

```
{$L memlib.obj}
PRICEDURE MemProc1; EXTERNAL;
```

7.9.4. Подпрограммы как параметры других подпрограмм.

Чтобы имя подпрограммы можно было передать в другую подпрограмму в виде параметра, вводится специальный процедурный тип - это тип заголовка подпрограммы в целом:

```
TYPE
RealFuncType = FUNCTION(t:Real):Real;
```

Подпрограммы этого типа (функции с теми же параметрами и возвращаемым типом) могут использоваться в описаниях параметров подпрограмм, но компилироваться должны в режиме разрешения дальних вызовов {$F+}.

```
{$F+}
FUNCTION SinExp(tt: Real) : Real;
begin
 SinExp :=Sin(tt) * Exp(tt)
end;
{$F-}
```

Такая функция может быть подставлена в вызов подпрограммы:

```
Integral(0,1,Res1,SinExp)
```

Для получения в переменной RES1 значения интеграла в пределах [0,1], если процедура Integral определена примерно так:

```
PROCEDURE Integral(LowerLimit, UpperLimit:Real; VAR Result
:Real; FuncT :RealFunctinType);
VAR t:Real;

begin
 Численное интегрирование функции FuncT с результатом в Result
end;
```

7.9.5. Переменные процедурного типа.

По формату они совместимы с переменными типа Pointer и после приведения типов могут обмениваться с ними значениями. Чтобы переменная - подпрограмма понималась как указа-

тель на адрес подпрограммы в памяти, она должна предваряться оператором @.

```
TYPE
DemoProcType = procedure(A,B:Word);
DemoRecType = RECORD
          X,Y:Word;
          Op:DemoProcType;
          END;

VAR P1,P2 :DemoProcType;
...
```

В примере приведена возможность хранения в записи не только данных , но процедур их обработки.

7.9.6. Обмен данными между подпрограммами через общие области памяти.

Можно, используя директиву совмещения адресов, объявить в нескольких подпрограммах области данных, совмещенные с некоторой глобальной переменной:

```
TYPE
Vector100Type = Array[1..100] of Real;
MatrixType =Array[1..10,1..10] of Real;
Matrix2Type =Array[1..50,1..2] of Real;
VAR
V:Vector100Type;

PROCEDURE P1;
VAR M : MatrixType absolute V;
begin end;

PROCEDURE P2;
VAR M2 :MatrixType absolute V;
begin end;

PROCEDURE P3;
VAR V3 : Vector100Type absolute V;
begin end;

begin end.
```

7.9.7. Статические локальные переменные.

Иногда необходимо определить переменные с локальной областью видимости (то есть объявленные внутри подпрограммы), но глобальным временем жизни (то есть сохраняющие свои значения между отдельными вызовами подпрограмм, в которых они объявлены).

Такие переменные вводятся в разделе CONST подпрограмм как типизированные константы (переменные со стартовым значением) и будут размещены компилятором в сегменте данных, как и их глобальные аналоги:

```
PROCEDURE XXX;
VAR ...
CONST
A:Word = 240;
B:Real=41.3;
begin {Здесь значения A и B могут изменяться с сохранением значений до следующих вызовов процедуры}
end;
```

7.9.8. Бестиповые параметры - переменные.

Паскалевские подпрограммы могут иметь в своих заголовках бестиповые параметры - переменные (передача по ссылке), с упоминанием только имени переменной без двоеточий и типов

```
PROCEDURE PDemo(VAR V1,V2);
FUNCTION FDemo(A: Integer; VAR V) : Real;
```

Через них в подпрограмму можно передавать адреса переменных любых типов - строк, массивов, записей и т д, но подпрограмма сама должна явно задавать тип, с которым она собирается работать. Это дает возможность составлять, например, программы обработки массивов произвольной длины - при использовании типизованных параметров это невозможно, т.к. размер массива входит в описание типа. В качестве примера составим функцию, суммирующую N элементов произвольных числовых массивов:

```
VAR
B1 :Array[-100..100] of Byte;
B2 :Array[0..999] of Byte;
B3 :Array['a'..'z'] of Byte;
S :String;
```

```
{$R-}     {Выключим проверку индексов массивов}
FUNCTION  Sum(VAR X;N:Word):LongInt;
TYPE
XType = Array[1..1] of Byte;
VAR
Summa:LongInt; i:Word;

begin
 Summa:=0;
 for i:=1 to N do Summa:=Summa + XType(X)[i];
 Sum:=Summa;
end;
{$R+}

begin
 S:='123456789';
 Writeln(Sum(B1,201));
 Writeln(Sum(B2[100],101));
 Writeln(Sum(B3['b'],10));
 Writeln(Sum(S[1],9));
end.
```

Функция Sum не боится несовместимости типов, но будет возвращать корректные результаты только для массивов с элементами типа `Byte` - так мы определили тип `XType`, к которому приводим все, что передается подпрограмме через параметр X. Обратите внимание на диапазон `XType` 1..1 - он не несет смысловой нагрузки и при отключенной проверке индексов может быть произвольным. Рассмотренный пример можно распространить и на записи, указатели, числовые переменные - рискованно однако передавать через бестиповый параметр множества или элементы создаваемых нами перечислимых типов из-за особенностей их машинного представления.

7.9.9. Рекурсия.

Как мы уже знаем, под рекурсией понимается вызов подпрограммы из тела этой же подпрограммы, что часто используется при программировании многих математических формул, имеющих рекурсивные определения - например, формула вычисления факториала

n! = n*(n-1)! , если n>0

или степени целого числа

x^n = x*x^(n-1) , если n>0

т. е. тогда, когда для вычисления следующего значения необходимо использовать предыдущее.

В Паскале рекурсия записывается так же, как в формулах - проиллюстрируем это на примере того же факториала:

```
FUNCTION Fact(n:Word):LongInt;
begin
 if n=0 then Fact:=1 else Fact:=n*Fact(n-1);
end;
```

и степени числа:

```
FUNCTION Pow(x:Real;n:Word):Real;
begin
 if = 0 then Pow:=1 else Pow:=x*Pow(x,n-1);
end;
```

При n>0 происходит следующее: запоминаются значения членов выражения в ветви else (n для факториала ,x для степени), для вычисления неизвестных вызываются те же функции, но с предшествующими аргументами. При этом вновь запоминается в другом месте стека значения членов и происходят вызовы до тех пор, пока выражение не станет полностью определенным - у нас это выполнение ветви then. После этого идет раскрутка в обратную сторону с изъятием из памяти отложенных значений до получения конечного результата.

Несмотря на наглядность рекурсивных описаний, те же задачи чаще всего более эффективно решаются обычными итерациями :

```
FUNCTION Pow(x:Real;n:Word):Real;
VAR i:Word;m:Real;
begin m:=1;
for  i:=1 to n do m:=m*x;
Pow:=m
end;
```

Рекурсивные описания целесообразно применять для компиляторов, которые при возможности сами преобразуют рекурсивные вызовы в итеративные процедуры (Turbo Prolog), а также в специфических задачах типа грамматического разбора символьных конструкций. Отметим также трудность принудительного выхода из рекурсивных подпрограмм - оператор Exit срабатывает только на 1 уровень глубины рекурсии.

7.10. Модули и библиотеки модулей в Паскале.

7.10.1. Структура модулей.

Модуль (UNIT) в Паскале - это совокупность определений типов, констант, переменных и подпрограмм, оформленная специальным образом. Модуль является единицей компиляции в Паскале, как и программа, но, в отличие от программы, он не может быть запущен на выполнение самостоятельно - его компоненты предназначены для построения программ или других модулей. Если любые фрагменты текстов программ, размещенные в других файлах, могут подключаться к текущей программе директивой {$I ИмяФайла}, то модули предварительно компилируются независимо от использующих их программ с получением файла с расширением .TPU. Чтобы включить содержащиеся в модуле подпрограммы в любую программу, достаточно указать его имя в директиве USES:

```
USES DOS,GRAPH;
```

и далее свободно использовать содержимое соответствующей библиотеки путем, например, вызова на выполнение необходимых подпрограмм.

Модули удобны как для создания собственных библиотек, так и для построения программ практически любого размера - сумма размеров используемых в программе модулей ограничена лишь физическим объемом памяти компьютера, да и то если не используется оверлейная структура. Общая структура модуля имеет вид:

```
UNIT ИмяМодуля;
INTERFACE      {раздел объявлений или интерфейсный}
USES СписокМодулей;
CONST          {Блок объявления библиотечных констант}
TYPE       {Типов}
VAR        {Переменных}
```

Заголовки подпрограмм, входящих в модуль

```
IMPLEMENTATION {Раздел реализации модуля}
USES
CONST
TYPE
```

```
VAR
LABEL

BEGIN
 Блок инициализации модуля.
END.
```

Все блоки внутри разделов необязательны. Если отсутствует раздел инициализации, то после тела последней подпрограммы просто ставится `END`. В этом разделе формируются стартовые значения переменных и какие-нибудь одноразовые действия. Выполнение любой программы начинается с выполнения блоков инициализации используемых ею модулей. Подключение модулей происходит в порядке их перечисления, в этом же порядке работают и их блоки инициализации, поэтому порядок подключения модулей может оказаться небезразличным. В случае возможности совпадения имен в разных модулях, в модулях и программе развязку можно осуществить указанием имени модуля перед именами переменных или подпрограмм через точку:

```
U1.t, U1.proc8(...);
```

Возможно возникновение "закольцованности" при взаимном использовании модулей в разделе `INTERFACE`. Проблема решается в этом случае выносом всех ссылающихся друг на друга элементов из обеих модулей в третий модуль с подключением его к программе. Модуль может не содержать подпрограмм, а состоять только из описаний типов, констант, переменных:

```
UNIT Common;
INTERFACE
ComArrayType = Array[0..99] of LongInt;
CONST
MaxSize = 100;
VAR
ComArray : ComArrayType;
Flag: Byte;
IMPLEMENTATION
END.
```

7.10.2. Система библиотечных модулей Паскаля (краткие справочные данные).

В Турбо Паскале определен ряд стандартных модулей TPU, обеспечивающих функции ввода - вывода, работы со строками, управления экраном дисплея, принтером и другими ресурсами компьютера. При компиляции новый программный код генерируется только для оригинальных фрагментов текста программы, а вызовы библиотечных функций на стадии компоновки вызывают включение в код компонентов библиотек.

7.10.2.1. Подпрограммы обработки символов.

Они содержатся в библиотеке поддержки языка TURBO.TPL.

Chr(X: Byte) : Char {Возвращает символ ASCII - кода X}
Ord(C: Char) : Byte {Возвращает ASCII - код символа C}
Pred(C: Char) : Char {Выдает предшествующий символ в алфавите}
Succ(C: Char) : Char {Следующий символ}
UpCase(C: char) : Char{Переводит символы латиницы в верхний регистр}

7.10.2.2. Подпрограммы обработки строк.

Length(S: String) : Byte {Возвращает длину строки}
Concat(S1, S2, . . , Sn) : String {Возвращает слитую строку}
Copy(S:String;Start, Len:ineger) : String {Возвращает подстроку длиной Len, начинающуюся с позиции Start строки S}
Delete(VAR S:String;Start, Len:Integer) {Удаляет подстроку}
Insert(VAR S:String;Subs:String;Start:Integer) {Вставляет подстроку}
Pos(SubS, S:String):Byte {Возвращает номер первого символа SubS в S или 0}
Str(X[:F[:n]]; VAR S:String) {Преобразует число X в строку S}
Val(S:String;VAR X; VAR ErrCode: Integer) {Преобразует строку S в число X}

7.10.2.3. Математические подпрограммы.

Abs(X){Убирает знак минус у целого или вещественного аргумента}, Pi, Sin(X), Cos(X), ArcTan(X), Sqrt(X), Sqr(X), Exp(X), Ln(X)
Ttunc(X) {Целая часть}
Frac(X) {Дробная часть}
Int(X) {Целая часть}
Round(X) {Округление до ближайшего целого}
Random {Случайное число 0..1}
Randomize {Случайное старт генераторов случайных чисел}
Random(X) {Случайное Word 0..X}
Odd(X) {True если X нечетный}
INC(VAR X:Целое), INC(VAR X:Целое;N:Целое),
DEC(VARX:Целое), DEC(VAR X:Целое;N:Целое) {Увеличение или уменьшение на 1 или на N}

7.10.2.4. Подпрограммы для работы с адресами.

CSeg:Word, DSeg:Word, SSeg:Word, Sptr:Word {Возвращают содержимое сегментных регистров процессора CS, DS, SS, SP}

Addr(X) :Pointer {Возвращает указатель на начало объекта X в памяти}
Seg(X):Word {Сегментный адрес объекта X}
Ofs(X):Word {Смещение в сегменте для X}
Ptr(S,O:Word):Pointer {Сборка указателя по сегменту и смещению}
SizeOf(X):Word {Размер объекта в байтах}
@X:Pointer {Аналог Addr}

7.10.2.5. Подпрограммы работы с динамически распределяемой на стадии выполнения программы памятью - "кучей".

Управление кучей осуществляет монитор кучи, являющийся частью системной библиотеки Турбо Паскаля и реализующий следующие подпрограммы:

New(VAR P :Pointer) {Отводит место для переменной P^ и присваивает ее адрес указателю P}
Dispose(VAR P:pointer) {Освобождает память, отведенную New}

GetMem(VAR P: Pointer; Size:Word) {Отводит Size байтов в куче, присваивая адрес места P}
FreeMem(VAR P: Pointer;Size :Word) {Освобождает то, что было выделено GetMem}
Mark(VAR P:Pointer) {Запоминает по адресу P текущее состояние кучи}
Release(VAR P:Pointer) {Возвращает кучу в состояние, запомненное в P через Mark}
New(Тип Указателя):Pointer {Альтернативная форма создания динамической переменной типа заданного базового типа}
MaxAvail:LongInt {Самый длинный свободный участок в куче}
MemAvail:LongInt {Сумма длин всех свободных кусков кучи}

Переменная HeapError позволяет вам реализовать функцию обработки ошибки динамически распределяемой области памяти. Эта функция вызывается каждый раз, когда программа динамического распределения памяти не может выполнить запрос на выделение памяти. HeapError является указателем, который ссылается на функцию со следующим заголовком:

```
FUNCTION HeapFunc(Size: Word): Integer; far;
```

Функция обработки ошибки динамически распределяемой области реализуется путем присваивания ее адреса переменной HeapEror:

```
HeapError := @HeapFunc;
```

Функция обработки ошибки динамически распределяемой области памяти получает управление, когда при обращении к процедурам New или GetMem запрос не может быть выполнен. Параметр Size содержит размер блока, для которого не оказалось области памяти соответствующего размера. Возвращаемое значение 0 приводит к возникновению ошибки времени выполнения. Возвращаемое значение 1 вместо ошибки приводит к тому, что процедуры GetMem или FreeMem
возвращают указатель nil.

```
FUNCTION HeapFunc(Size: Word): Integer; far;
begin HeapFunc := 1; end;
```

Присутствие подобной функции в ваших программах позволит вам анализировать результат выполнения ваших операций по работе с памятью и в случае необходимости выдавать корректную диагностику.

7.10.2.6. Подпрограммы для работы с файлами.

Турбо Паскаль поддерживает так называемый буферизованный обмен с файлами - через область оперативной памяти, отводимую при открытии файла. Поэтому при записи в файл

физическое перемещение информации на диск произойдет либо после заполнения всего буфера, либо после закрытия файла.

Турбо Паскаль вводит и поддерживает 3 файловых типа:

- текстовые файлы (тип Text);
- компонентные файлы (тип File of ТипКомпонетов);
- бестиповые файлы (тип File);

Перед началом работы с файлом в памяти должна быть размещена переменная одного из перечисленных файловых типов:

```
ft1:Text;
fk1:File of Real;
fb1:File;
```

Подпрограммы, общие для всех типов файлов:

```
Assign(VAR f; FileName:String) {Связывает файловую переменную f с именем FileName физического файла}
Reset(VAR f) {Открытие файла для чтения}
Rewrite(VAR f) {Открытие файла для записи}
Close(VAR f) {Закрытие открытого файла}
Rename(VAR f;NewName :String) {Переименовывает только закрытый файл}
Erase(VAR f ) {Стирает только закрытый физический файл }
EOF(VAR f):Boolean {True если достигнут конец файла}
```

Подпрограммы для работы с текстовыми файлами (ASCII-коды, конец строки - #13, конец файла - ^Z - код 26, автоматическое преобразование в символьную форму при записи в файл и в машинную форму при чтении из файла)

```
SetTextBuf(VAR f:Text; VAR Buf[;BufSize:word]) {Установка буфера перед открытием файла}
Append(VAR f:Text) {Открытие для дополнения в конец файла}
Flush(VAR f:Text) {Перепись буфера в файл до полного заполнения}
EOLn(VAR f:Text):Boolean {True если конец строки или конец файла}
SeekEOLn(VAR f:Text):Boolean {True если после указателя только пробелы, табуляторы или символы конца строк}
SeekEOF(VAR f:Text):Boolean {True если конец файла или после указателя нет ничего значащего }
Read(f,X),Read(f,X!,X2...Xn),ReadLn {Чтение полных или частичных строк}
```

Write(f,X), Write(f,X1,X2..Xn), WriteLn {Запись в файл}

Типизированные (компонентные) файлы.

Процедуры Reset и Rewrite - обе устанавливают для типизированных и бестиповых файлов режим "для чтения и записи" по значению 2 предопределенной системной переменной FileMode. Чтобы установить "только запись", этой переменной необходимо присвоить значение 1, а для "только чтение" - 0.

После открытия файла ввод и вывод осуществляются стандартными подпрограммами Read(f,X) и Write(f,X) - только переменная X или их список того же типа, что и тип файлового компонента. К типизированным файлам нет смысла применять ReadLn или WriteLn - в них просто нет понятия строки или ее конца и нет признака конца файла.

Преимуществом типизированных файлов является эффективное хранение числовой информации и возможность обмена сложными и громоздкими структурами одной командой:

```
TYPE dim100x20 = Array[1..100,1..20] of Real;
VAR XX,YY:dim100x20; fIn,fOut:File of dim100x20;
begin
...
Read(fIn,XX); ... Write(fOut,YY);
end.
```

Бестиповые файлы.

С их помощью можно осуществлять обмен с дисками участками рабочей памяти произвольной длины, а считанные из файла данные можно преобразовывать в любой формат посредством приведения типов. При открытии файла буфер передачи данных устанавливается неявно в 128 байт. Но явно его можно установить любой размер при открытии файла расширенной записью процедур:

Reset (VAR f:File;BufSize:Word) или
Rewrite (VARf:File;BufSize:Word)

Подпрограммы ввода - вывода имеют вид:

BlockRead (VAR f:File;VAR Destin;Count:Word [;VAR ReadIn:Word])
BlockWrite (VAR f:File;VAR Source;Count:Word[;VARWriteOut:Word])
- Чтение и запись Count блоков размером буфера каждый. Необязательные параметры содержат число фактически считанных или записанных блоков, что можно использовать для проверки правильности выполнения процедур.

Подпрограммы прямого доступа к типизованным и бестиповым файлам:

```
FileSize(VAR f):LongInt {размер файла в компонентах или блоках}
FilePos(VAR f):LongInt {Номер последнего уже обработанного компонента или блока}
Seek(VAR f;N:LongInt) {Установка текущего компонента или блока считая от 0}
Truncate(VAR f) {Усечение файла после текущей позиции}
```

7.10.2.7. Подпрограммы для работы с каталогами.

```
GetDir(drive:Byte;VAR S:String) {Имя текущего каталога на диске drive}
ChDir(S:String)                 {Установка текущего каталога}
MkDir(S:String)                 {Создание каталога}
RmDir(S:String)                 {Удаление пустого каталога}
```

Обработка ошибок ввода - вывода может быть выполнена в нашей программе при отключении системной проверки ключом {$I-} и использовании функции IOResult:Integer, возвращающей код ошибки в диапазоне 2..200 (2..99 - коды ошибок DOS, 100..149 - диагностируемые программой, 150..200 - критические аппаратные)

7.10.3.Модуль CRT.

Содержит специальные подпрограммы для работы с текстами на дисплее, организовывать окна вывода на экран, настраивать цвета символов, управлять курсором, динамиком, опрашивать клавиатуру. Модуль размещен в системной библиотеке TURBO.TPL Этот модуль есть смысл подключать всегда директивой USES CRT, так как его блок инициализации переключает вывод непосредственно через видеопамять, минуя DOS и BIOS.

При подключенном CRT можно выводить на дисплей управляющие символы с кодами 0..31 без выполнения ими управляющих функций (кроме символов с кодами 7,8,10,13).

Во-вторых, расширяются возможности редактирования строк при использовании подпрограмм Read , ReadLn:

Esc - Стирает набранную строку

^D - Восстанавливает символ из стертой строки ввода
^F - Восстанавливает всю стертую строку
^Z - Завершает ввод и вырабатывает признак конца файла, если системная переменная CheckEOF = `True`

Переменные CRT:

```
CheckSnow:Boolean {Снятие снега на CGA-дисплеях}
DirectVideo:Boolean {Максимальная скорость вывода на экран}
CheckBreak:Boolean {Установка ^Break }
CheckEOF:Boolean  {Разрешение ввода ^Z как конца файла}
TextAttr:Byte    {Установка цвета букв и фона}
LastMode:Word  {Для работы с процедурой TextMode}
WindMax, WindMin:Word {Параметры текущего окна}
```

Подпрограммы CRT:

```
Window (X1, Y1, X2, Y2:Byte) {Задание текущего окна}
ClrScr                {Очистка окна}
TextMode (M:Word)     {Установка текстового режима}
GotoXY (X, Y:Byte)    {Установка курсора}
WhereX:Byte, WhereY:Byte {Номер столбца и строки курсора}
ClrEOL                {Стирание строки от тек позиции}
InsLine               {Вставка пустой строки вместо текущей}
DelLine               {Удаление строки}
TextColor (C:Byte)    {Цвет символов}
TextBackGround (C:Byte) {Цвет фона}
HighVideo             {Яркость включить}
LowVideo              {Яркость отключить}
NormVideo                  {Восстановить цветовой режим}
Sound (Hz:Word)            {Звук частотой Hz герц}
NoSound               {Выключить звук}
Delay (ms:Word)            {Пауза в мс}
KeyPressed:Boolean {Нажата ли клавиша}
ReadKey:Char          {Символ нажатой клавиши}
AssignCRT (VAR f:Text) {Связать текстовый файл с CRT}
```

7.10.4. Модуль DOS.

В модуле DOS Турбо Паскаля реализованы подпрограммы для работы с операционной системой MS DOS, многие из них - это просто оформленные в Паскалевском синтаксисе вызовы подпрограмм MS DOS, транслируемые в команды

INT НомерПрерывания

после заполнения регистров номерами подпрограмм и необходимыми для выполнения данными.

7.10.4.1. Подпрограммы опроса и установки параметров MS DOS.

```
DOSVersion:WORD          {Возвращает код версии MS DOS}
GetCBreak(VAR B:Boolean) {Читать параметр BREAK}
SetCBreak(B:Boolean)     {Установить параметр BREAK}
GetVerify(V:Boolean)     {Читать параметр VERIFY, управляющий режимом записи на диск}
SetVerify(V:Boolean)     {Установит параметр VERIFY}
EnvCount:Integer {Число системных переменных ОС - имена и значения, заданные командами SET, PATH, PROMPT}
EnvString(N:Integer) {Получить строку переменной номер N}
GetEnv(E:String):String      {Значение переменной E}
```

7.10.4.2. Подпрограммы для работы с часами и календарем.

GetDate, SetDate, GetTime, SetTime, PackTime, UnPackTime, GetFTime, SetFTime - мы приводим только названия подпрограмм, а полный синтаксис их использования вы найдете в Help - системе Турбо Паскаля

7.10.4.3. Анализ ресурсов дисков.

```
DiskFree(D:Word):LongInt {Размер свободного пространства на диске с кодом D : 0 - текущий, 1 - диск A, 2 - диск B и т д}
DiskSize(D:Word)     {Общая вместимость диска}
```

7.10.4.4. Подпрограммы для работы с каталогами и файлами.

Для работы с файлами вводятся специальные типы и константы:

```
CONST
ReadOnly = $01;Hidden = $02;SysFile = $04; VolumeId = $08;
Directory = $10; Archive = $20; AnyFile = $3F;

{При использовании эти константы можно складывать.}
```

```
fmClosed = $D7B0; {Файл закрыт}
fmInput =$D7B1; fmOutPut = $D7B2; fmInOut = $D7B3;

TYPE
SearchString = RECORD
Fill:Array[1..21] of Byte; {Системное поле}
Attr:Byte; Time: LongInt; Size: LongInt; Name: String[12];
END;

FileRec = RECORD
Handle :Word;
Mode:Word; {Состояние файла}
RecSize :Word; {Длина записи компонентного файла}
Private :Array[1..26] of Byte; {Не используется}
UserData :Array[1..16] of Byte; {Любой комментарий к файлу}
Name :Array[0..79] of char; {Полное имя дискового файла }

TextBuf = Array[0..127] of char;

TextRec = RECORD
Handle :Word;
Mode:Word;                    {Состояние файла}
BufSize :Word;
Private :Array[1..26] of Byte; {Не используется}
BufPos:Word; BufEnd: Word; BufPtr :^TextBuf;
OpenFunc : Pointer; InOutFunc: Pointer;
FlushFunc :Pointer; CloseFunc: Pointer;{Адреса драйверов}
UserData :Array[1..16] of Byte; {Любой комментарий к файлу}
Name :Array[0..79] of char;  {Полное имя дискового файла }
Buffer :TextBuf;
END;
```

Переменная DosError типа Integer содержит номер ошибки, возвращаемой MS DOS при неправильной операции; 0 - отсутствие ошибки, остальные расшифровки мы приводить не будем.

Подпрограммы:

```
FindFirst(Path:String; Attr:Word; VAR SR:SearchRec) {Поиск первого файла подходящего запросу}
FindNext(VAR SR : SearchString) {Вызывается после FindFirst для поиска следующего подходящего}
```

FSearch(Path :PathStr; DirList :String) : PathStr {Поиск файла в списке каталогов}
GetFAttr(VAR f: File; VAR FA :Word)
SetFAttr(VAR f: File; VAR FA :Word)
FSplit(Path: PathStr; VAR Dir : DirStr; VAR Name : NameStr;
VAR Ext : ExtStr) {Разбить полное имя на составляющие}
FExpand(Path :pathStr): PathStr {Приписать к имени текущий маршрут}

7.10.4.5. Подпрограммы для работы с прерываниями MS DOS.

В операционной системе вызов ее услуг реализован в виде подпрограмм, активизируемых по команде прерывания процессора

INT НомерПрерывания

где НомерПрерывания по существу является номером некоторого комплекса подпрограмм. Комплексы сгруппированы в основном по аппаратному признаку: прерывание с номером 16H дает доступ к подпрограммам опроса клавиатуры на уровне операционной системы, прерывание с номером 25H - к подпрограммам управления диском и т.д. Особую роль играет прерывание 21H, открывающее доступ к нескольким десяткам подпрограмм, образующим собственно операционную систему.

В работе на машинно-ориентированном языке Ассемблера, программист для вызова подпрограмм ОС должен загрузить в регистры процессора номер подпрограммы и ее аргументы и вызвать прерывание, открывающее доступ к этой подпрограмме. В Турбо Паскалевском модуле DOS есть подпрограммы поддержки прямых обращений к подпрограммам DOS:

GetIntVec(N:Byte;VAR Adress:Pointer) {Адрес подпрограммы прерывания с заданным номером N}
SetIntVec(N:Byte;Adress:Pointer) {Установить по прерыванию N подпрограмму с адресом Adress}
Intr(N:Byte; VAR R:Registers) {Вызвать прерывание N , передав номер подпрограммы и параметры в записи R}
MsDos(VAR R:Registers) {Специализированный вызов прерывания 21H}

Новый тип Registers определен в модуле DOS как вариантная запись:

```
TYPE Registers = RECORD
case Integer of
0:(AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags:Word);
1:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
                END;
```

Переменные этого типа служат для доступа к регистрам процессора при вызовах прерываний, причем вариант 0 открывает доступ к 16 - разрядным регистрам, а вариант 1 - к 8 - разрядным:

```
USES DOS;
VAR R1,R2 :Registers;
begin R1.AX:=$01FF; R2.BL:=$CA; ... END.
```

7.10.5. Модуль Strings.

В Паскале строки обычного типа (String) хранятся как байт длины, за которым следует последовательность символов. Строки с завершающим нулем не содержат байта длины. Вместо этого они состоят из последовательности ненулевых символов, за которыми следует символ NULL (#0). DOS ограничивает их размер 65535 символами.

Строки с завершающим нулем хранятся в виде символьных массивов, начинающихся с 0:

```
Array[0..X] of Char;
```

Более всего строки Паскаля и строки с завершающим нулем отличаются интенсивностью использования указателей. В модуле System имеется встроенный тип PChar, который представляет собой указатель на ASCIIZ-строку. Переменной этого типа можно присвоить строку:

```
var
 P: PChar;
 ...
 P := 'Привет...';
```

В результате такого присваивания указатель указывает на область памяти, содержащую ASCIIZ-строку, являющуюся копией строкового литерала. Компилятор записывает строковые литералы в сегмент данных:

```
const
 TempString: Array[0..14] of Char = 'Привет...'#0;
var
 P: PChar;
 ...
```

```
P := @TempString;
```

Типизированная константа типа `PChar` может инициализироваться строковой константой или адресом:

```
const
 Message: PChar = 'Stack overflow!';
 GrVBuf:PChar=PChar($a0000000);
```

Там, где используется тип `PChar`, может использоваться и символьный массив. В этом случае компилятор преобразует символьный массив в константный указатель на первый элемент массива. Символьный указатель можно индексировать аналогично символьному массиву.

Подпрограммы, предназначенные для работы со строками с завершающим нулем, сосредоточены в модуле `Strings`:

StrCat Добавляет исходную строку к концу целевой строки и возвращает указатель на целевую строку.

StrComp Сравнивает две строки S1 и S2. Возвращает значение < 0, если S1 < S2, равное 0, если S1 = S2 и > 0, если S1 > S2.

StrCopy Копирует исходную строку в целевую строку и возвращает указатель на целевую строку.

StrECopy Копирует исходную строку в целевую строку и возвращает указатель на конец целевой строки.

StrIComp Сравнивает две строки без различия регистра символов.

StrLCat Присоединяет исходную строку к концу целевой строки. При этом обеспечивается, что длина результирующей строки не превышает заданного максимума. Возвращается указатель на строку-результат.

StrLComp Сравнивает две строки с заданной максимальной длиной.

StrLCopy Копирует заданное число символов из исходной строки в целевую строку и возвращает указатель на целевую строку.

StrEnd Возвращает указатель на конец строки, то есть указатель на завершающий строку нулевой символ.

StrDispose Уничтожает ранее выделенную строку.

StrLen Возвращает длину строки.

StrLIComp Сравнивает две строки с заданной максимальной длиной без различия регистра символов.

StrLower Преобразует строку в нижний регистр и возвращает указатель на нее.

StrMove Перемещает блок символов из исходной строки в целевую строку и возвращает указатель на целевую строку. Два блока могут перекрываться.

StrNew Выделяет для строки память в динамически распределяемой области.

StrPas Преобразует строку с завершающим нулем в строку Паскаля.

StrPCopy Копирует строку Паскаля в строку с завершающим нулем и возвращает указатель на строку с завершающим нулем.

StrPos Возвращает указатель на первое вхождение заданной подстроки в строке, или `nil`, если подстрока в строке не содержится.

StrRScan Возвращает указатель на последнее вхождение указанного символа в строку, или `nil`, если сим вол в строке отсутствует.

StrScan Возвращает указатель на первое вхождение указанного символа в строку, или `nil`, если символ в строке отсутствует.

StrUpper Преобразует строку в верхний регистр и возвращает указатель на нее.

7.11. Использование ресурсов в прикладных программах.

7.11.1. Обработка "хвоста" командной строки

"Хвостом" командной строки называют все слова, которые должен (или может) ввести пользователь после имени исполняемого файла при запуске программы - обычно в них содержатся самые необходимые для начала работы прикладной программы сведения - например, имя файла с конкретными данными, количество строк и столбцов в таблице для программ обработки таблиц, для простых и коротких данных это могут быть и сами данные - например, номер интересующего пользователя члена числового ряда для программ вычисления членов различных числовых последовательностей (прогрессий, чисел ряда Фибоначчи и пр.). Допустимая длина хвоста программной строки 127 байт и размещается сам хвост операционной системой в префиксе программного сегмента начиная с относительного адреса 81H. Предыдущий байт по адресу 80H содержит число ударов пользователя по клавишам после имени запускаемой программы. Хвост командной строки не имеет никакого завершающего

символа - после последнего осмысленного символа будет лежать просто "мусор".

Само имя запущенной программы находится в блоке окружения, адрес которого тоже есть в PSP по относительному адресу 2CH, так что собственное имя тоже доступно нашей программе, что бывает актуально в связи с предоставляемой ОС свободой в переименовании любых файлов.

Турбо Паскаль предоставляет для доступа к командной строке 2 подпрограммы - ParamCount, возвращающую количество слов в командной строке без учета имени самой программы и ParamStr(c:Integer), возвращающую строку параметра с номером c; ParamStr(0) возвращает имя исполняемого файла.

```
VAR s:String;
begin
Writeln('Имя нашей программы :', ParamStr(0));
if ParamCount >0 then S:=ParamStr(1)
else begin
Write('Введите параметр - имя файла');
Readln(s);
if s='' then s:='default.dat';
end;
...
end.
```

7.11.2. Организация субпроцессов в Турбо Паскале

MS DOS имеет мощные средства организации субпроцессов и Турбо Паскаль предоставляет подпрограммы для доступа к этим средствам.

Программа с субпроцессами - такая программа, которая запускает на выполнение другую программу, которая в свою очередь может запустить еще одна и т.д. Запускающая программа (процесс) как бы замирает, оставаясь в оперативной памяти компьютера, а субпроцесс выполняется в свободной от родителя памяти как любая другая программа. После его завершения вновь "оживает" породившая его программа более высокого уровня. Типичным примером программы, активно использующей субпроцессы, является NORTON COMMANDER, запускающая различные файлы на выполнение вроде dbview.exe.

При составлении программ с субпроцессами в Турбо Паскале всегда необходимо предварительно рассчитывать требо-

вания к памяти для конкретной программы. Для управления памятью служит ключ компилятора $M, который должен стоять до первой строки текста программы в блоке основной программы :

{$M Размер_стека, Минимум_Кучи, Максимум_Кучи}

Размер стека указывается в пределах 1024..65520, минимум кучи обычно 0, но может быть до 655360 байт (если в момент запуска программа имеет свободной памяти меньше минимума кучи, работать она не сможет и остановится с выдачей сообщения), максимум кучи - тоже не более 655360. Если свободной памяти меньше, чем заявленный максимум кучи , то выделится столько, сколько реально есть. Необходимый размер стека можно оценить, просуммировав размеры локальных переменных в подпрограммах с учетом вложенности их вызовов. Для определения необходимых размеров кучи можно просуммировать размеры динамических переменных, созданных с помощью New и GetMem (заботясь при этом о том, чтобы куча не была фрагментированной). После определения размеров используемой родителем памяти можно использовать специальные подпрограммы для запуска субпроцессов.

При запуске любой программы может происходить предусмотренная в ней замена содержимого ряда системных векторов прерываний, в частности это делает среда Турбо Паскаля, запоминая заменяемые векторы в специальных переменных типа Pointer. При запуске субпроцесса разумно предоставить ему возможность использовать системные векторы, а не замененные. Подпрограмма SwapVectors восстанавливает сохраненные в переменных с именами SaveIntNN векторы, записывая туда же текущие, поэтому при повторном вызове подпрограмма снова восстановит отключенные подпрограммы прерываний - ее используют до и после подпрограммы Exec.

SwapVectors {Восстановление системных или временных векторов прерываний до и после запуска субпроцессов.0 - нормальное завершение}

Exec(ExeFile,ComLine:String) {Запуск выполняемого файла ExeFile (субпроцесса) со строкой параметров ComLine}

Для использования в качестве субпроцессов внутренних (встроенных) команд MS DOS необходимо запускать

COMMAND.COM и передавать ему текст команды в виде параметров:

```
EXEC(‘command.com’,’/c copy a.txt + b.txt’);
```

Важно не забывать включить в командную строку первым по счету ключ /c для командного процессора - если это не сделать, то получится выход в ДОС и вернуться из субпроцесса можно будет только подачей команды EXIT с клавиатуры.

При запуске командного процессора через подпрограмму EXEC лучше всего использовать его полное имя, которое можно, в свою очередь, получить через подпрограмму модуля DOS GetEnv:

```
Exec(GetEnv(‘COMSPEC’),’/c format a: /s’);
```

При таком запуске трудно проанализировать, чем завершилось выполнение субпроцесса, так как анализируется выполнение командного процессора, который редко дает сбои. Для анализа завершения субпроцессов можно использовать подпрограмму :

```
DosExitCode: Word {Возвращает код завершения субпроцесса}
```

0 - нормальное завершение с возвратом управления
1 - субпроцесс был прерван через нажатие Ctrl+Break
2 - субпроцесс был прерван по ошибке устройства
3 - субпроцесс остался резидентным после завершения процедурой Keep

7.11.4. Некоторые типовые подпрограммы работы с клавиатурой в прикладных Паскаль - программах.

7.11.4.1. Общие сведения о клавиатуре и средствах ОС для обработки клавиатурного ввода.

Основным средством ввода для пользователя является клавиатура - она служит для ввода командной строки при запуске программ из ОС, обеспечивает подготовку текстов в текстовых редакторах, позволяет управлять ходом программы нажатием всевозможных комбинаций клавиш, осуществлять выборы в меню программных услуг и многое другое.

Клавиатура ПК содержит встроенный микропроцессор; он при каждом нажатии и отпускании клавиши определяет ее порядковый номер, называемый скэн-кодом, и помещает его в порт 60H микросхемы связи с периферией. 1-е 7 битов скэн-кода - это номер клавиши, а 8-й сигнализирует нажатие (0) или отпускание (1). При удержании клавиши в нажатом состоянии дольше установленного времени задержки микропроцессор клавиатуры начнет генерировать коды нажатия с заданной частотой. Задержка и частота могут устанавливаться в нужные значения либо через порты клавиатуры, либо использованием функции AH=03H прерывания 16H BIOS. Стандартный обработчик аппаратного прерывания от клавиатуры имеет номер 9 - по его вектору помещается BIOS - программа обработки, анализирующая и обрабатывающая скэн-коды клавиш. В зависимости от типа клавиши применяется свой алгоритм обработки скэн-кода:

1. Шифт-клавиши Shift, Alt, Ctrl.
2. Триггерные клавиши NumLock, ScrollLock, CapsLock
3. Клавиши с буферизацией расширенного кода
4. спецклавиши

За каждой шифт- и триггерной клавишей закреплен свой бит в ячейках памяти по адресам 40:17H и 40:18H; при каждом нажатии и отпускании эти биты инвертируются и состояние бита рассказывает о том, нажата или отпущена клавиша. За триггерными клавишами закреплены 2 бита, 1 из них инвертируется при нажатии, а 2-й при нажатии и отпускании. Текущее состояние этих клавиш используется при выборе правил преобразования скэн-кодов от других клавиш.

При нажатии большинства клавиш и их комбинаций в специальный буфер помещается 2-байтовый BIOS-код ,у которого младший байт равен ASCII-коду либо 0, а старший - скэн-коду или так называемому расширенному скэн-коду. 2- байтовый 0 + расширенный скэн-код записывается в буфер клавиатуры при нажатии функциональных клавиш, Ins, Del, клавиш управления курсором и их комбинаций с Alt, Ctrl, Shift и пр комбинациях Alt с ASCII-клавишами. Клавишная комбинация PrnScr вызывает прерывание 5, Alt+Ctrl+Del передает управление программе начальной загрузки BIOS, а Ctrl+C записывает по адресу 00471H значение 80H, используемое как флаг желания пользователя остановить выполнение операций ввода-вывода.

Буфер клавиатуры.

Буфер BIOS для кодов клавиш занимает 32 байта с адреса 40:1EH до 40: 3EH, запись в буфер выполняет BIOS-обработчик 9-го прерывания, а чтение - функции BIOS прерывания 16H. Буфер рассчитан на 15 нажатий клавиш + 2 холостых байта для Enter. Буфер организован как циклическая очередь, доступ к которой осуществляется с помощью указателя "головы" по адресу 40:1AH и указателя "хвоста" по адресу 40:1CH. Оба указателя содержат смещение от сегмента 40H. При каждом нажатии по указателю головы записывается 2-байтовый код и указатель смещается на 2 байта, перепрыгивая при необходимости через холостую пару байтов и может "обогнать" содержимое указателя хвоста. При чтении из буфера смещается указатель хвоста. Если содержимое указателей равно, буфер пуст.

Буфер клавиатуры - классический пример использования кольцевого буфера для организации асинхронного взаимодействия 2-х программ по схеме "производитель-потребитель" - запись данных в буфер и считывание из буфера происходят в случайные, не связанные между собой моменты времени. При переполнении буфера производитель блокируется, пока потребитель не освободит чтением одно или более мест в буфере.

7.11.4.2. Подпрограммы для обработки клавиатурного ввода.

Очистка буфера клавиатуры выполняется при необходимости принять ожидаемый символ от пользователя.

```
procedure ClrKeyBuf;
VAR ch:Char;
begin while KeyPressed do ch:=ReadKey; end;
```

Ожидание нажатия произвольной клавиши используется при задержках например для просмотра пользователем предыдущего вывода.

```
Procedure Wait;
begin REPEAT UNTIL KeyPressed end;
```

Прием символа нажатой клавиши.

```
USES crt;
{$I clrkeybuf.INC}
{$I wait.INC}
VAR c:char;
begin
ClrScr;
Writeln('Нажмите любую символьную клавишу');
```

```
ClrKeyBuf;
c:=ReadKey;{Теперь в c - ASCII - код клавиши}
...
end.
```

Если надо ожидать ввода не любых, а конкретных символов, то поступают так:

```
USES crt;
{$I ClrKey.INC}
VAR c:char;
begin
...
Writeln('Нажмите А, Б или В');
ClrKeyBuf;
REPEAT
c:=ReadKey
UNTIL(c in ['A'..'B','a'..'в']);
clrkeybuf;
case c of
'A','a':begin ...end;
'Б','б':begin ...end;
'В','в':begin ...end;
end;
ClrKeyBuf;
...
end.
```

Опрос расширенных кодов и функциональных клавиш.

При использовании функции ReadKey приходится осуществлять повторное чтение после получения нулевого кода при первом обращении к функции. В принципе при работе со спецклавишами, имеющими нулевое значение в младшем байте двухбайтового кода, а в особенности при смешанном приеме кодов символьных и спецклавиш удобнее пользоваться непосредственно полными двухбайтовыми кодами, но библиотечные модули Паскаля не содержат таких функций. Позже мы покажем, как самим составить такую функцию с помощью услуг BIOS.

```
USES crt;
VAR ch:char;
{$I clrkey.INC}
begin
...
ClrKeyBuf;
```

```
Writeln('Нажмите функциональную клавишу');
ch:=ReadKey;
if ch =#0 then ch:=ReadKey;
...
end.
```

Если ждать нажатия клавиши нежелательно, то из программы надо убрать вызов ClrKeyBuf:

```
if KeyPressed then begin
ch:=ReadKey;
if ch=#0 then ch:=ReadKey;
```

Запись в клавиатурный буфер. Обычно выполняется при создании демонстрационных программ. При этом используют абсолютные адреса клавиатурного буфера и указатели на его голову и хвост.

Для записи в буфер клавиатуры символьных кодов можно использовать такой вариант:

```
{$M 4096,0,0)
USES crt,dos;

Procedure UnReadKey(KeyW:Word);
CONST
KbdStart =$1E;
KbdEnd =$3c;
VAR
KbdHead:Word absolute $40:$1A;
KbdTail:Word absolute $40:$1C;
OldTail:Word;
begin
OldTail:=KbdTail;
if KbdTail=KbdEnd then
KbdTail:=KbdStart
else INC(KbdTail,2);
if KbdTail=KbdHead then KbdTail:=OldTail
else MemW[$40:OldTail]:=KeyW;
end;
```

Для записи расширенных кодов клавиш:

```
Procedure UnReadExtCode(ExtCode:Word);
begin UnReadKey(Swap(ExtCode)) end;
```

Запись строк осуществляется посимвольно:

```
Procedure UnReadString(S:String);
VAR i:Byte;
```

```
begin
for i:=1 to Length(S) do UnReadKey(Ord(S[i])); end;
end.
```

7.11.5. Управление выводом на дисплей средствами языка Паскаль

7.11.5.1. Еще о модуле CRT.

Аббревиатура CRT может быть раскрыта по-русски как "электронно-лучевая трубка" - в этом модуле реализованы специальные подпрограммы для работы с текстом на дисплее, позволяющие управлять текстовыми режимами, организовывать текстовые окна, настраивать цвета символов, управлять курсором.

Этот модуль есть смысл подключать всегда директивой USES CRT, так как его блок инициализации переключает вывод непосредственно через видеопамять, минуя DOS и BIOS, а, следовательно, делает вывод значительно быстрее.

Обычно же вывод совершается по схеме: Процедура WRITE -->функция MS DOS вывода строки --> подпрограмма BIOS --> видеопамять адаптера. При подключении CRT перед выполнением основного блока программы происходит переназначение стандартных файлов на фиктивное устройство CRT и весь вывод начинает выполняться с помощью подпрограмм этого модуля. Если же нужно по каким либо причинам отказаться от услуг подключенного модуля CRT, то в программе должны появиться операторы:

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

Пустая строка в операторе Assign означает стандартное устройство CON.

Использование процедур модуля CRT демонстрируют пара простых примеров:

1. Пример процедур синтеза звуков:

```
USES CRT;
Procedure Phone;
VAR i:Word;
begin
REPEAT
 for i:=1 to 100 do begin Sound(1200);Delay(10); Nosound end;
 Delay(800)
UNTIL KeyPressed
```

```
end;

Procedure Bell;
begin
REPEAT
Sound(1800);Delay(2);
Sound(2000);Delay(2);
Sound(2200);Delay(2);
Sound(2400);Delay(2);
UNTIL KeyPressed;
Nosound
end;

Procedure Sirena;
VAR i:Word;
begin
REPEAT
 for  i:=400 t0 800 do begin Sound(i);Delay(3) end;
 Nosound
UNTIL KeyPressed;
end;

Procedure Pause;
VAR ch:Char;
begin while KeyPressed do ch:=ReadKey;
Delay(200)
end;

BEGIN
ClrScr;
Write('Нажмите клавишу'#10#10#13);
Write('Звук телефона'#13); Phone; Pause;
Write('Звук зуммера'#13); Bell; Pause;
Write('Звук сирены'#13); Sirena; Pause;
ClrScr;
end.
```

2. Процедура закраски квадратной области экрана с диагональю X0, Y0 - X, Y символами Ch с задержкой при закраске ms:

```
USES CRT;
Proceedure Spiral(x0,y0,x,y:Byte;ms:Word;Ch:Char);
VAR height,width,j:Byte; c:Integer;
begin
c:=1; width:=x-x0+1; height:=y-y0+1;
```

```
REPEAT
for j:=1 to width do begin
 gotoxy(x,y); Write(Ch);
 if(y>Hi(WindMax)) and (x>Lo(WindMax)) then begin
 gotoxy(1,1); InsLine       end;
Delay(ms); x:=x-c end;
x:=x+c; DEC(height);
for j:=1 to height do begin
y:=y-c; gotoxy(x,y); Write(ch); Delay(ms)
end;
DEC(width); x:=x+c; c:=-1*c
UNTIL (height<1) or (width<1);
gotoxy(1,1);
end;

VAR i:Byte; {Пример использования процедуры}
begin
ClrScr;
Spiral(1,1,80,25,2,#176);
for i:=1 to 10 do begin
TextAttr:=i;
Spiral(2*i,i,5*i,2*i,4,Chr(47+i));
 end;
Readln;
end.
```

7.11.5.2. Нестандартные методы работы с текстовыми изображениями.

В огромном числе задач использование стандартных средств вывода на экран малоэффективно или просто громоздко - заполнение окон каким - либо символом, сохранение изображений в файлах и пр.

1. Программный опрос текстовых режимов дисплея.

Видеопамять как мы уже знаем располагается в адресах от $A000:$0000 до $BFFF:$0000. Монохромные режимы используют память начиная с адреса $B000:0000, а цветные - с B800:0000. Реальный размер используемой видеопамяти зависит от режима работы; его минимальное значение при 40 столбцах и 25 строках равно 40x25x2=2000 байт, а максимальное при 80 столбцах и 50 строках равно 8000 байт.

Хорошая программа должна сама определять, с каким режимом она имеет дело. Определение текущего режима решает-

ся опросом фиксированных адресов памяти, в которых хранятся сведения о конфигурации.

Рассматриваемая ниже функция опрашивает такой специальный адрес, по которому MS DOS хранит данные о текущей конфигурации системы; функция возвращает адрес начала видеопамяти текстового режима:

```
FUNCTION GetScreenPtr:Pointer;
begin
if (Mem[0:$0410] and $30) = $30
then GetScreenPtr:=Ptr($B000,0)
else GetScreenPtr:=Ptr($B800,0)
end;
```

Номер текущего текстового режима получить легко - он всегда есть в младшем байте переменной модуля CRT LastMode, при этом 0-й бит старшего байта содержит признак 43 или 50 строк. С использованием этой переменной построены 2 функции опроса видеорежимов:

```
FUNCTION CurrentMode:Byte;
begin CurrentMode:=Lo(LastMode) end;

FUNCTION Font8x8YES:Boolean;
begin Font8x8YES:=(LastMode and Font8x8) = Font8x8 end;
```

2. Определение длины видеопамяти:

```
FUNCTION GetScreenSize :Word;
VAR
R: Byte absolute $0000:$0484;
C: Byte absolute $0000:$044A;
begin
ifHi(LastMode)=1 then GetScreenSize:=Succ(R)*C*2
                 else GetScreenSize:=25*C*2;
end;
```

3. Определение числа столбцов:

```
FUNCTION GetColNum:Byte;
begin GetColNum:=Mem[0:$44A] end;
```

4. Определение числа строк:

```
FUNCTION GetRowNum:Byte;
begin GetRowNum:=GetScreenSize div GetColNum div 2 end;
```

7.11.5.3. Организация доступа к видеопамяти.

Все четные адреса видеобуфера, начиная с 0, содержат коды символов, а нечетные - цветовые атрибуты. Доступ к видеобуферу можно получить, наложив на него массив 2-байтовых элементов:

```
TYPE
VideoWord = RECORD
            Symbol:Char;
            Attrib:Byte;
            END;
VideoText=Array[1..50*80] of VideoWord;
VideoTextPtr=^VideoText;
```

Непосредственный доступ к видеопамяти производится через переменную указательного типа VideoTextPtr. Если известно текущее число столбцов M, то преобразование координат X,Y в номер элемента массива осуществляется легко: M*(Y-1)+X.

Запоминание и восстановление окон.

Описанную методику работы с видеопамятью можно использовать для работы с экранными окнами - в этом случае надо считывать, запоминать или заполнять не всю видеопамять, а набор строчных фрагментов окна, длина которых равна ширине окна. Начало 1-го фрагмента Start для окна, заданного координатами X1, Y1, X2, Y2 можно вычислить так:

Start = M *(Y1-1) +X1, где M - число столбцов текущего текстового режима.

Начало каждой следующей строки получается добавлением числа M к началу предыдущего. Длина строки определяется просто:

Width=(X2-X1) +1

а число строк Height = (Y2 -Y1) +1

Для запоминания окна потребуется память размером:
Size = Width*Height*2

Обычно при сохранении окон их строки записываются в память последовательно и при восстановлении необходимо вновь вычислять положение каждой из них.

7.11.5.4. Управление формой курсора.

Проще всего - вызовом функции AH=1 прерывания 10H с указанием начальной и конечной строк курсора в регистрах процессора. Формально номера строк принимаются от 0 до 31, при задании начальной строки с номером 32 курсор невидим. Примеры процедур для управления формой курсора:

```
USES CRT,DOS;
```

1. Установка формы курсора.

```
PROCEDURE SetCursorSize(c_Start,c_End:Byte);
VAR Regs:Registers;
begin
with Regs do begin
AH:=$01; CH:=c_Start; CL:=c_End;
 end;
Intr($10,Regs)
END;
```

2. Курсор по умолчанию.

```
PROCEDURE SetNormalCursor;
VAR SE:Word;
begin
if(LastMode>=Font8x8) then SE:=$0507
else if(LastMode=Mono) then SE:=$0b0c else SE:=$0607;
SetCursorSize(Hi(SE),LO(SE));
END;
```

3. Крупный блок-курсор.

```
PROCEDURE SetBlockCursor;
VAR c_End:Byte;
begin
if(LastMode>=Font8x8) or (LastMode<>Mono)
then c_End:=7 else c_end:=13;
SetCursorSize(0,c_End)
END;
```

4. Отключение курсора

```
SetCursorSize(32,0);
```

7.11.5.5. Графический вывод.

В Турбо Паскале он с помощью драйверов BGI не отличается от рассмотренного для Турбо С, конечно, с коррекцией на паскалевский синтаксис.

8. Практическое процедурное программирование в (Турбо, Borland) Паскаль.

8.1. Извлечение аргументов из командной строки.

Обычно первой программой выбирают вывод на экран строки "Hello World! - Привет Мир!". Вы можете это проделать - программа выглядит так:

```
begin
  writeln('Hello World! - Привет Мир!');
end.
```

В теле главной программы, ограниченном парой слов begin...end. вызывается на выполнение библиотечная функция форматированного вывода writeln, которой мы даем в качестве аргумента строку - константу (литерал).

Основной нашей задачей в самом начале обучения программированию будет научиться извлекать из командной строки DOS, сформированной при вызове нашей программы на выполнение, имеющиеся в ней слова - аргументы. Дело в том, что подавляющее большинство программ всегда знают "что делать", но, начиная работу, хотят узнать, с какими объектами надо это делать. Только программа, реализующая некоторый алгоритм для целого класса однотипных объектов обработки, имеет какую-то потребительскую ценность. Поэтому большинство программ требуют после начала выполнения уточняющих задачу аргументов и эти аргументы удобнее всего задать сразу после имени программы в командной строке.

Как мы уже сообщали, Турбо Паскаль предоставляет такую возможность - в виде библиотечных функций: ParamCount, которая возвращает количество аргументов в командной строке, и ParamStr, которая возвращает строку, соответствующую заданному ее аргументом номеру слова в командной строке.

Функции EnvCount и EnvStr, в свою очередь, возвращают количество и строки среды окружения, формирующиеся командами DOS path,set.

Итак, первая программа, в которой мы учимся получать слова из командной строки. Директивой uses мы объявляем о намерении использовать данные и подпрограммы библиотечных модулей DOS и CRT. В CRT находится прототип подпрограммы очистки экрана clrscr. Наберите эту программу и испытайте ее, четко уясните себе понятия "командная строка", "слова команд-

ной строки", "строки окружения" - без этого трудно будет осознанно программировать что бы то ни было.

```
uses crt,dos;
var         {Раздел объявления глобальных переменных}
  i:integer;
begin
(*Очистим экран, чтобы вывод был на 'чистую поверхность'*)
  clrscr;
  (*Выводим количество слов в командной строке*)
  writeln('В командной строке ',ParamCount+1,' слов,
  разделенных пробелами');
  (*Выводим сами слова командной строки*)
  writeln('Слова командной строки:'); (*Что-то вроде заголовка*)
  for i:=0 to ParamCount do     (*Цикл вывода слов*)
  writeln(ParamStr(i));
  (*Выводим строки среды, сформированные командами DOS
    path, set*)
  writeln('Строки среды окружения:'); (*Опять заголовок*)
  for I := 1 to EnvCount do (*Опять цикл вывода строк*)
    Writeln(EnvStr(I));
  readkey; (*Ожидаем нажатия произвольной клавиши
          без отображения ее на экране*)
end.
```

8.2. Процедура, решающая квадратные уравнения с вещественными коэффициентами, заданными в командной строке.

Здесь нет незнакомого вам материала и мы ограничимся комментариями по тексту программы; единственное, о чем напомним - это о необходимости преобразования цифровых слов командной строки в вещественные числа - мы это делаем использованием библиотечной функции VAL().

```
uses crt;
(*Подпрограмма для вывода на экран решения квадратных
уравнений*)

procedure quadr(a,b,c:real);
var
  d,sqrd,r,i:real;
begin
(*Дискриминант и квадратный корень из его абсолютного значения*)
  d:=b*b-4*a*c;
  sqrd:=sqrt(abs(d));
  r:=-b/(2*a);
  i:=sqrd/(2*a);
```

```
(*Если дискриминант положителен или равен нулю*)
if(d>=0)then
writeln('Корни вещественные:'#$D#$A'x1 =
',(r+i):5:3,#$D#$A'x2 = ',(r-i):5:3)
  else
    writeln('Корни комплексные:'#$D#$A'x1 = ',r:5:3,'+',i:5:3,
           'i'#$D#$A'x2 = ',r:5:3,'-',i:5:3,'i');
end;

var
  a,b,c:real;
  Code:integer;
begin
  if(ParamCount<3) then
  begin
    writeln('Неполадки с коэффициентами уравнения');
    exit;
  end;
  Val(ParamStr(1),a,Code);
  Val(ParamStr(2),b,Code);
  Val(ParamStr(3),c,Code);
  clrscr;
  quadr(a,b,c);
end.
```

8.3. Простые числа.

Задача: вывести на экран все простые числа, не превышающие заданного в командной строке.

Алгоритм решения в разделе 4.9.1.

Прежде всего нам придется проверить, задал ли пользователь нашей программы предел для последовательности простых, и если да - "достать" из командной строки цифровое слово, преобразовать его в целое число и далее использовать в программе, которая выглядит при этом следующим образом:

```
uses crt;(*Очистка экрана и вывод*)
var
i,j,limit:longint;    (*Для предела простых*)
Code:integer;
flag:boolean;(*Для признака простое =true или не простое
=false*)
const beg=3; (*С этого числа начнем список простых*)

begin
  (*Если не задано предельное число*)
  if(ParamCount<1)then
  begin  writeln('Не задано предельное число'); exit;  end;
```

```
(*Цифровое слово в 1-м элементе массива слов командной
строки преобразуем в длинное целое с помощью библио-
течной процедуры Val*)
Val(ParamStr(1),limit,Code);
clrscr;           (*Очистим экран*)
(*Пока не превышен предел для делимого i с шагом 2*)
i:=beg;
while i<=limit do
begin
flag:=true;        (*Вначале предполагаем простое*)
                    (*Пробуем делители j пока не дойдем
                    до квадратного корня делимого*)
    j:=beg-2;
    while j<=sqrt(i) do
    begin
      if i mod j=0  (*Если разделилось без остатка*)then
      begin  flag:=false; break; end;
      inc(j,2);
    end;
    (*Если ни одного безостаточного деления - печатаем*)
    if flag then  write('  ',i); inc(i,2); end;
end.
```

8.4. Определение длины строки, заданной указателем на начало и "закрытой" нулем.

Мы уже рассматривали эту и последующие задачи на модели оперативной памяти, в библиотеке такие подпрограммы тоже есть, но для понимания процессов обработки строк этот материал важен и мы его приведем.

Пусть для начала исследуемая строка задана литералом (в реальных программах она читается из файла или вводится с клавиатуры - одним словом, поступает извне на стадии выполнения программы).

Алгоритм очень прост: выделим память для счетчика символов в строке, обнулим его, и, наращивая указатель на строку, пока не встретим завершающий нуль-байт, будем наращивать счетчик.

```
uses crt;
const
  st:PChar='Надо определить длину этой строки';
  len:byte=0;
begin
  while st^<>#0 do
  begin st:=@st[1]; inc(len); end;
  clrscr;
  writeln('Длина строки = ',len);
end.
```

8.5. Копирование строки, заканчивающейся двоичным нулем.

Алгоритм очевиден - наращиваем индекс, пока не скопируем нулевое значение в буфер назначения.

```
uses crt;
const
  src:PChar='Надо скопировать эту строку ';
  i:integer=0;
var
dest:array [0..255] of char;  (*Буфер назначения - копируем сюда*)
begin
clrscr;
dest[i]:=src[i];
while bytebool(dest[i]) do
begin inc(i); dest[i]:=src[i]; end;
writeln('Строка - оригинал: ',src,#$D#$A'Строка - копия: ',dest);
end.
```

8.6. Сравнение 2-х строк.

Алгоритм решения в разделе 4.9.2.

```
uses crt;
const
  str1:PChar='Наш дядя самых честных правил';
  str2:PChar='Наш дядя самых честных правил';
  str3:PChar='Наш дядя не самых честных правил';

(*Подпрограмма сравнения строк*)
function compstr(s1,s2:PChar):integer;
var
  r:integer;
begin
r:=integer(byte(s1^))-byte(s2^);
while bytebool(byte(s1^)+byte(s2^)) and not wordbool(r) do
begin s1:=@s1[1];s2:=@s2[1];r:=integer(byte(s1^))-byte(s2^);end;
compstr:=r;
end;
(*Главная ф-ция тестирует подпрограмму
сравнением str1 с str2 и str3*)

begin
  clrscr;
  if not wordbool(compstr(str1,str2))
  then writeln('Строки 1 и 2 равны');
  if(compstr(str1,str2)>0)then writeln('Строка 1 больше 2-й ');
  if(compstr(str1,str2)<0)then writeln('Строка 1 меньше 2-й ');
  if(compstr(str1,str3)<0)then writeln('Строка 1 меньше 3-й ');
```

```
  if(compstr(str1,str3)>0)then writeln('Строка 1 больше 3-й ');
  if not wordbool(compstr(str1,str3))then
  writeln('Строка 1 равна 3-й ');
end.
```

8.7. Выделение и преобразование в целое цифровой подстроки в строке, заданной указателем на начало и завершающейся нулем.

Алгоритм решения в разделе 4.9.3.

```
uses crt;
function str_digit(s:PChar):integer;
const
(*Здесь будет формироваться число из цифровой подстроки*)
d:integer=0;
begin
  while(s^<'0') or (s^>'9') and bytebool(s^) do
  s:=@s[1];(*Пока не цифра и не нуль*)
  while(s^>='0') and (s^<='9') do      (*Пока цифра*)
  begin (*Преобразуем в число с учетом позиции цифры*)
    d:=(byte(s^)-byte('0'))+10*d;
    s:=@s[1];
  end;
  str_digit:=d;
end;

(*Теперь тестирование в главной подпрограмме*)
const
  st:PChar='Цена этого изделия 1860 рублей';
begin
  clrscr;
  writeln('Выделенная подстрока содержит число ',str_digit(st));
end.
```

8.8. Печать отрезка ряда чисел Фибоначчи с длиной, заданной в командной строке.

Каждый член этого ряда - натуральное число, равное сумме 2-х предшествующих членов ряда. Очевидно, что 2 первых члена придется определить "волевым порядком" - например, как 1 и 1.

```
uses crt;
(*Процедура, печатающая заданное количество членов ряда Фибоначчи*)

procedure fib(n:word);
var
  i,sum:longint;
```

```
(*Два первых члена ряда Фибоначчи*)
const  fa:array [0..1] of longint=(1,1);
begin
  write(#$D#$A,fa[0],#$D#$A,fa[1]);
  for i:=2 to n-1 do
  begin
    sum:=fa[0]+fa[1];  (*Вычисляем следующее*)
    write(#$D#$A,sum);     (*Печатаем*)
    fa[0]:=fa[1];fa[1]:=sum;
  end;                  (*Сдвигаем влево на 1 число*)
end;

var
  n,Code:integer;
begin
  if ParamCount<1 then
  begin writeln('Не задано факториальное число');exit;end;
  Val(ParamStr(1),n,Code);(*Определяем количество чисел*)
  clrscr;
  fib(n);         (*Вызываем специалиста по Фибоначчи*)
end.
```

8.9. Подпрограмма обмена значениями между двумя областями памяти.

Эти области заданы указателями на их начало и размером (одинаковым для обеих в нашем примере) и используются в алгоритме Евклида для определения наибольшего общего делителя. Подпрограмма обмена выполнена универсальной, пригодной для любых типов любых размеров.

Сам алгоритм Евклида выполнен в вызываемой функции nod() в классическом варианте последовательным вычитанием до тех пор, пока числа не сравняются, а для упорядочения чисел по убыванию nod(), в свою очередь, вызывает swapobj().

```
uses crt;
(*Подпрограмма, меняющая местами содержимое 2-х  объектов в
памяти, заданных своими указателями и размером*)

procedure swapobj(obj1,obj2:pointer;size:integer);
var
  tmp:pointer;
begin
(*выделим память для временного сохранения одного из объектов*)
  GetMem(tmp,size);
  move(obj1^,tmp^,size); (*перегрузим в "запасник" один объект*)
  move(obj2^,obj1^,size);(*второй на место первого*)
  move(tmp^,obj2^,size); (*из запасника в область памяти второго*)
  FreeMem(tmp,size);  (*освободим арендованную память*)
```

```
end;

    (*Подпрограмма, определяющая наибольший общий де-
литель двух целых чисел, использующая swapobj*)

function nod(d1,d2:longint):longint;
var
  r:longint;
begin
while d1<>d2 do
begin if(d1<d2) then  swapobj(@d1,@d2,sizeof(d1));d1:=d1-d2; end;
nod:=d1;
end;

var
  n1,n2,r:longint; Code:integer;
begin
  clrscr;
  if ParamCount<>2 then
  begin writeln('Непорядок с аргументами');exit; end;
  Val(ParamStr(1),n1,Code); Val(ParamStr(2),n2,Code);
  writeln('НОД чисел ',n1,' и ',n2,' равен ',nod(n1,n2));
end.
```

8.10. Программирование рекурсивных алгоритмов.

Вызов подпрограммы из другой подпрограммы - обычный прием в процедурном программировании. При составлении алгоритмов часто бывает удобно на некоторой фазе обработки данных из тела текущего алгоритма вызвать на выполнение "самого себя", уже с измененными аргументами - такие алгоритмы называются рекурсивными. Различают прямую или непосредственную рекурсию (вызов алгоритмом самого себя) и опосредованную рекурсию, при которой из алгоритма 'А' вызывается алгоритм 'Б', а он в свою очередь вызывает алгоритм 'А'.

Паскаль поддерживает программирование рекурсивных алгоритмов, любая подпрограмма может вызвать и быть вызванной из любой подпрограммы. При рекурсивных вызовах все происходит так, как будто обращение осуществляется к другому экземпляру этой же подпрограммы. Рекурсий в программной реализации следует избегать - они приводят к большим затратам времени на вызовы и возвраты и усиленно расходуют память стека. Но рекурсия часто позволяет с меньшими затратами усилий составить алгоритм решения, а потом, после уяснения его особенностей, переписать с помощью циклов - это всегда возможно, но не всегда легко.

8.10.1. Простейший пример использования рекурсии - вычисление факториала натурального числа.

Он равен произведению числа на факториал числа, уменьшенного на 1 и просто напрашивается вызов из подпрограммы вычисления факториала N самой себя для вычисления факториала N-1.

В приведенной ниже программе есть 2 подпрограммы - вычисление факториала через рекурсивные вызовы и вторая - без рекурсии. Обе вызываются из функции main() прямо из подпрограммы печати.

```
uses crt;
(*Подпрограмма рекурсивного алгоритма вычисления факториала*)
function rfact(n:word):longint;
begin
  if(n=1)then rfact:=1  else  rfact:=n*rfact(n-1);
end;

(*Подпрограмма нерекурсивного алгоритма вычисления
факториала*)
function nrfact(n:word):longint;
const  f:longint=1;
begin for n:=n downto 2 do f:=f*n; nrfact:=f; end;
var
  n:word; Code:integer;
begin
  if(ParamCount<1)then
  begin writeln('Не задано факториальное число'); exit; end;
  Val(ParamStr(1),n,Code);
  clrscr;
  writeln('Факториал числа ',n,' равен ',rfact(n));
  writeln('Факториал числа ',n,' равен %ld',nrfact(n));
end.
```

8.10.2. Рекурсии - задача о Ханойских башнях.

Задача: на одном из 3-х колышков надето N дисков с убывающими к вершине радиусами; необходимо переложить диски на другой колышек в том же порядке с использованием 3-го как промежуточного, но нельзя класть больший диск на меньший.

Алгоритм решения в разделе 4.9.4.

```
uses crt;

(*Рекурсивный алгоритм "ханойские башни"*)
procedure hanojrec(n:word;a,b,c:char);
begin
  if(n=1)then
```

```
    write(a,'->',b,' ')
  else
  begin
  hanojrec(n-1,a,c,b); hanojrec(1,a,b,c); hanojrec(n-1,c,b,a);
  end;
end;

type
  abyte=array[0..0]of byte;
  pabyte=^abyte;

procedure hanoj(n:byte);
var
  a,b,c:pabyte;  btop,ctop:byte;  i:integer;
const
  atop:byte=0; min:char='a';
begin
  {$R-}
  (*Выделим память в куче под все массивы сразу
  как для утроенного массива a*)
  GetMem(a,3*(n+1));
  (*Определим указатели на массивы b и c*)
  b:=a; inc(b,n+1); c:=b;  inc(c,n+1);
  (*Начальные присвоения переменным*)
  btop:=n; ctop:=n; b^[btop]:=n; c^[ctop]:=n;
  for i:=0 to n do
    a^[i]:=i;
  (*Реализуем описанный алгоритм перекладывания дисков*)
  (*Пока все диски не окажутся на стержне 'c' или 'b'*)
  while(btop<>0)and(ctop<>0)do
  begin
    if(btop=0)or(ctop=0)then  break;
    (*В зависимости от того, где сейчас самый малень-
кий диск, перекладываем его по часовой стрелке*)
    case(min)of
      'a':
      begin
      write(min,'->',char(byte(min)+1),' ');
      inc(min); dec(btop); b^[btop]:=a^[atop]; inc(atop);
      end;
      'b':
      begin
      write(min,'->',char(byte(min)+1),' ');
      inc(min); dec(ctop); c^[ctop]:=b^[btop]; inc(btop);
      end;
      'c':
      begin
        write(min,'->',char(byte(min)-2),' ');
        dec(min,2);dec(atop);a^[atop]:=c^[ctop];inc(ctop);
      end;
    end;
```

```
      (*Минимальный диск изменил позицию и мы занимаем-
ся парой стержней, на которых его нет*)

    case(min)of
     'a':
     begin
       if(b^[btop]<c^[ctop])then
       begin
       write('b->c '); dec(ctop);c^[ctop]:=b^[btop]; inc(btop);
       end
       else
        if(c^[ctop]<b^[btop])then
        begin
        write('c->b '); dec(btop);b^[btop]:=c^[ctop];inc(ctop);
        end;
     end;
     'b':
     begin
       if(a^[atop]<c^[ctop])then
       begin
       write('a->c ');dec(ctop);c^[ctop]:=a^[atop];   inc(atop);
       end
       else
        if(c^[ctop]<a^[atop])then
        begin
        write('c->a '); dec(atop); a^[atop]:=c^[ctop];inc(ctop);
        end;
     end;
     'c':
     begin
       if(a^[atop]<b^[btop])then
       begin
       write('a->b '); dec(btop);b^[btop]:=a^[atop];    inc(atop);
       end
       else
        if(b^[btop]<a^[atop])then
        begin
        write('b->a ');dec(atop); a^[atop]:=b^[btop];  inc(btop);
        end;
     end;
    end;
  end;
  FreeMem(a,3*(n+1));
  {$R+}
end;

    (*В основной программе вызываются на выполнение по
очереди рекурсивный и нерекурсивный варианты для срав-
нения результатов*)

var
  n:byte;
```

```
  Code:integer;
begin
  if(ParamCount<1)then
  begin
  writeln('Не задано количество дисков в командной строке');exit;
  end;
  Val(ParamStr(1),n,Code);
  clrscr;
  hanojrec(n,'a','b','c'); writeln; hanoj(n);
end.
```

8.10.3. Рекурсии - программный генератор перестановок N попарно различных чисел.

Алгоритм решения в разделе 4.9.5.

Получение количества переставляемых элементов, выделение памяти под массив их индексов и начальное присвоение значений элементам этого массива выполнено в главной программе, оттуда же вызываются на выполнение оба варианта генератора.

```
uses crt;
type
  impossible=array[0..0]of word;
  pimpossible=^impossible;
var
NN:word;
AN:pimpossible; (*Для количества чисел и указателя на
начало массива индексов*)
     (*Рекурсивный генератор перестановок*)
procedure permute_rec(a:pimpossible;n:word);
var
  i,j:integer; temp,number:word;
begin
{$R-}
if(n>1)then
begin
permute_rec(a,n-1);
(*Переставляем значения элементов a[n-1] и a[i-1]*)
for i:=n-1 downto 1 do
begin temp:=a^[n-1]; a^[n-1]:=a^[i-1]; a^[i-1]:=temp;
permute_rec(a,n-1);
(*Переставляем значения элементов a^[n] и a^[i]*)
temp:=a^[n-1]; a^[n-1]:=a^[i-1]; a^[i-1]:=temp;
end;
end
else
begin
for j:=1 to NN do  write(a^[j-1]);  writeln;  end;
{$R+}
```

```
end;

var
  p:word; (*Для счетчика количества перестановок;*)

(*Генератор перестановок без использования рекурсии*)
procedure permute(a:pimpossible;n:word);
var
  tmp,j,l,k:word; i:integer;
begin
{$R-}
  while 1 shl 1 = 2 do
  begin
    (*Ищем правую границу упорядоч по возрастанию*)
    i:=n-2;
    while(a^[i]>a^[i+1]) and (i>=0) do
     dec(i);
    if(i<0)then break;(*Закончили - перестановки все*)
    (*Ищем самый правый больший граничного*)
    j:=n-1;
    while(j>i) and (a^[j]<a^[i]) do  dec(j);
    tmp:=a^[i];a^[i]:=a^[j];a^[j]:=tmp; (*Переставляем*)
    k:=1;
    for j:=i+1 to ((i+1+n) div 2)-1 do
    (*Реверсируем хвост массива*)
    begin
     tmp:=a^[j];a^[j]:=a^[n-k]; a^[n-k]:=tmp;inc(k);
    end;
    for l:=0 to n-1 do write(a^[l]); writeln; inc(p); end;
    gotoxy(20,12);
    writeln('Перестановки без рекурсии всего = ',p);
{$R+}
end;

var
  i,Code:integer;
begin
{$R-}
 (*Берем из командной строки количество элементов*)
  if ParamCount<1 then
    writeln('Не задано количество перестановок')
  else
  begin
    Val(Paramstr(1),NN,Code);
(*Просим память под массив для индексов переставляемых
объектов*)
    GetMem(AN,sizeof(word)*NN);
    clrscr;
    (*заполняем числами натурального ряда*)
    (*и печатаем стартовую последовательность*)
    for i:=0 to -1+NN do
    begin  AN^[i]:=i; write(i); end;
```

```
    writeln;  inc(p);
    permute(AN,NN);
    readkey; (*Пауза до нажатия клавиши*)
    clrscr;
    permute_rec(AN,NN);
    gotoxy(20,12);
    writeln('Рекурсивные перестановки');
    readkey; FreeMem(AN,sizeof(word)*NN);
  end;
{$R+}
end.
```

8.10.4. Рекурсия и алгоритмы перебора вариантов с возвратом.

Задача: на шахматной доске размером NxN найти все возможные расстановки N не бьющих друг друга ферзей.

Алгоритм решения в разделе 4.9.6.

```
uses crt;
type
  ossible=array[0..0]of word;
  possible=^ossible;
{$R-}
var
  F,(*Адрес массива для хранения варианта расстановки*)
  COLARR,   (*то же для индикации занятости столбца*)
  DS,    (*для индикации занятости диагонали, у которой*)
      (*постоянна сумма индексов столбцов и строк*)
  DR:possible;   (*то же для постоянной разности индексов*)
N,           (*для размерности матрицы   *)
ND,          (*для количества диагоналей *)
row,        (*для текущего индекса строки*)
number:word;

(*Нам будет удобно составить предварительно  несколько
вспомогательных подпрограмм - это улучшит читабель-
ность программы*)

(*Ф-ция, определяющая, свободно ли поле для установки*)
function isfree(row,col:word):boolean;
begin
  if wordbool(COLARR^[col])and wordbool(DS^[row+col]) and
    wordbool(DR^[N-1+row-col])then
    isfree:=true
  else
    isfree:=boolean(0);
end;

(*Процедура,занимающая поле*)
procedure setfild(row,col:word);
```

```
begin
F^[row]:=col;COLARR^[col]:=0;DS^[row+col]:=0;DR^[N-1+row-col]:=0;
end;

(*Процедура, освобождающая поле*)
procedure freefild(row,col:word);
begin
COLARR^[col]:=1;DS^[row+col]:=1;DR^[N-1+row-col]:=1;
end;

(*Процедура печати очередной расстановки*)
procedure printconf;
var
  i:byte;
begin
write('Номер расстановки: ',number,' ');
for i:=0 to N-1 do write(F^[i]);writeln; end;

(*Рекурсивная процедура генерации расстановок*)
procedure generation;
var
  col:word;
begin
  col:=0;
  repeat
  begin
  (*Если поле^[row]^[col] свободно*)
  if(isfree(row,col))then
  (*займем его занеся номер столбца в массив расстановки и обозначим нулями занятость столбца и диагоналей*)
  begin
    setfild(row,col);
    inc(row);
    (*Если все строки заняты*)
    if(row=N)then
    (*печатаем очередную конфигурацию искомых элементов*)
    begin inc(number); printconf; end
     (*иначе рекурсивно вызываем саму себя,
     но уже с новой исследуемой строкой*)
    else  generation;
    (*Возвращаемся на строку назад для
    исследования в ней следующего столбца*)
    dec(row);
    (*Освобождаем ранее занятое в ней поле^[row]^[col]*)
    freefild(row,col);
  end;
  (*Исследуем следующий столбец в строке row*)
  inc(col);
  end
  until(col>=N);
end;
```

```
procedure newgeneration;
var
  col_temp:word;
const
  col:word=0;
begin
while true <> false do
begin
repeat
begin
if(isfree(row,col))(*Если поле^[row]^[col] свободно*)
then
(*займем его занеся номер столбца в массив расстановки
и обозначив нулями занятость столбца и диагоналей*)
begin setfild(row,col);inc(row); col:=0;
if(row=N)then (*Если все строки заняты*)
begin (*печатаем очередную конфигурацию искомых элементов*)
inc(number); printconf;
(*возвращаемся на строку с неисчерпанным списком столбцов*)
while(row>0)do
begin  dec(row); col_temp:=F^[row];freefild(row,col_temp);
if(col_temp<(N-1))then  break;
end;
(*Если такой строки нет - все конфигурации отпечатаны*)
if(row=0 )and( col_temp=N-1)  then exit;
col:=col_temp+1;
end;(*if по все строки заняты *)
end(*if по если поле свободно*)
(*Исследуем следующий столбец в строке row*)
else
inc(col);
end
until(col>=N);
(*возвращаемся на строку с неисчерпанным списком столбцов*)
while(row>0)do
begin  dec(row); col_temp:=F^[row];freefild(row,col_temp);
if(col_temp<(N-1))then break;
end;
(*Если такой строки нет - все конфигурации отпечатаны*)
if(row=0 )and( col_temp=N-1) then  exit;
col:=col_temp+1;
end;(*forever*)
end;

var
  i,Code:integer;
begin
  clrscr;
  if(ParamCount<>1)then
  begin
```

```
    writeln('Неполадки с параметрами -
            задайте только размер строки');
    exit;
  end;
  Val(ParamStr(1),N,Code);     (*Размер матрицы*)
  ND:=2*N-1;                  (*Количество диагоналей*)
  (*Выделим память под все массивы сразу
   как под массив результата*)
  GetMem(F,(2*N+2*ND)*sizeof(word));
  (*Определим указатели на другие массивы в выделенной памяти*)
  COLARR:=possible(@F^[N]);
  DS:=possible(@COLARR^[N]);
  DR:=possible(@DS^[ND]);
(*'Объединичим' все массивы сразу -
т е обозначим свободными все поля*)
  for  i:=0 to -1+N+2*ND do
    COLARR^[i]:=1;
(*Начальная  подготовка  закончена - вызываем  функцию
генерации  расстановок*)
  writeln('Не рекурсивный генератор');
  newgeneration;
  readkey;
  for  i:=0 to -1+N+2*ND do
    COLARR^[i]:=1;
  writeln('Рекурсивный генератор');
  generation;
  FreeMem(F,(2*N+2*ND)*sizeof(word));
end.
```

8.10.5. Простые числа - решето Эратосфена и битовая упаковка булевских переменных.

Задача: вывести на экран все простые числа, меньшие заданного пользователем, используя алгоритм Эратосфена "просеивания" нечетных чисел.

Алгоритм решения в разделе 4.9.7.

```
uses crt;

{$R-}
type
  arh=array[0..0]of byte;
  parh=^arh;

var
  limit,           (*количество чисел*)
  (*индексы кандидатов в простые и непрост*)
  candidat,nonprost:word;
  bit:parh;      (*адрес массива для побитовой работы*)
  d,              (*для значения числа*)
  nbit:integer;   (*для номера бита в активном байте*)
```

```
  byte_ptr:parh;     (*для адреса активного байта*)

const
  first=3;            (*первое простое*)

     (*подпрограмма обнуления бита с заданным номером i
в массиве char-элементов с указателем bit*)

procedure sbros_bit(i:longint;b:parh);
var
  bptr:parh;
begin
  bptr:=parh(@b^[i shr 3]);(*номер байта с заданым битом char*)
  nbit:=7-(i and 7);     (*номер бита в байте*)
  bptr^[0]:=bptr^[0] and not ($0001 shl nbit);
end;

     (*Собственно алгоритм Эратосфена*)
procedure eratosfen;
var
  lbyte:word;
begin
     (*необходимая память в байтах для хранения битовой  модели
массива нечетных - примерно вдвое меньше заданного предельного
числа, т.к. нечетных - половина *)

  lbyte:=limit shr 2; (*адрес выделенного массива*)
  GetMem(bit,lbyte);  (*выделим память*)
  FillChar(bit^,lbyte,$ff); (*заполним массив единицами*)

  (*цикл перебора битовых элементов массива*)
for candidat:=0 to limit do
begin
byte_ptr:=parh(@bit^[candidat shr 3]);(*адрес активного байта*)
nbit:=7-(candidat  and  7); (*номер бита счетом справа налево*)
(*если число простое*)
if boolean(byte_ptr^[0] and (1 shl nbit)) then
    begin
      d:=2*candidat+first;  (*вычисляем его*)
      write(d,'  ');      (*и печатаем*)
(*находим индекс  непростого(кратного простому) *)
      nonprost:=d+candidat;
(*в цикле обнулим все кратные текущему простому*)
      while(nonprost<=limit)do
      begin
        sbros_bit(nonprost,bit);
        inc(nonprost,d);
      end;
    end; (*if end;*)
  end;
  FreeMem(bit,lbyte);   (*освободим память*)
```

```
end;

var
  Code:integer;
(*И теперь главная программа*)
begin
  if(ParamCount<1)then
  begin
    writeln('Не задан предел для простых');
    exit;
  end;
  (*Получаем колич нечетных чисел -их половина*)
  Val(ParamStr(1),limit,Code);
  limit:=limit shr 1;
  clrscr;
  eratosfen; (*Вызываем подпрограмму печати простых чисел*)
end.
```

8.11. Примеры простых программ, работающих с файлами.

8.11.1. Запись в файл: Вывести в файл с заданным в командной строке именем текущую таблицу кодировки символов.

План решения в разделе 4.9.8.

А вот так это будет выглядеть на языке Паскаль:

```
    var
 i,j:integer; kod:char; f:text;
begin
  if(ParamCount<>1)then
begin
writeln('Не задано имя файла под таблицу кодировки');
exit; end;
  {$I-}
  Assign(f,ParamStr(1));
  ReWrite(f);
  if wordbool(IOResult)then
  begin
    writeln('Неудача с открытием указанного файла');
    exit;
  end;
  (*Вначале выведем строку с номерами столбцов таблицы*)
  write(f,' ');
  for i:=0 to 15 do write(f,i:4); writeln(f);
  (*А теперь строки с символами*)
  for i:=0 to 15 do
  begin
    write(f,i:-4);  (*Выводим номер строки*)
```

```
    for j:=0 to 15 do   (*и символы*)
    begin  kod:=char(i*16+j);
      if(kod=#9)or(kod=#10)or(kod=#13)then  write(f,' УС')
      else  write(f,kod:4);
    end;
    writeln(f);
  end;
  close(f);
  {$I+}
end.
```

8.11.2. Чтение из файла.

Составить программу, подсчитывающую относительные частоты всех символов (частное от деления общего количества появлений каждого символа на суммарное число символов в анализируемом тексте) латинского алфавита в тексте из произвольного, предположительно англоязычного, файла, имя которого задано в командной строке ОС.

Алгоритм в разделе 4.9.9.

```
uses crt; (*Для очистки экрана*)

var
  freq:array[0..25]of longint; (*Массив счетчиков*)
  ch:char;
  i:integer;
  i1:real;
  f:file of char;
const
  count:longint=0;(*Общий счетчик букв*)
begin
  if(ParamCount<>1)then
  begin
    writeln('Не задано имя файла под таблицу кодировки'); exit;
  end;
  {$I-}
  Assign(f,ParamStr(1));  Reset(f);
  if wordbool(IOResult)then
  begin
    writeln('Неудача с открытием указанного файла'); exit;
  end;
  (*Посимвольная обработка содержимого файла*)
  while not eof(f) do
begin
read(f,ch);
if(ch>='A')and(ch<='z')then  (*Если прочитанный символ буква*)
    begin
      inc(count);        (*Наращиваем счетчик букв*)
      if(ch>='a')and(ch<='z')then (*Если буква маленькая*)
        (*делаем ее большой*)
```

```
        ch:=char(byte(ch)+byte('A')-byte('a'));
       (*Наращиваем соответствующий счетчик*)
       inc(freq[byte(ch)-byte('A')]);
     end;
  end;
  (*Вывод результата на экран в 2 колонки*)
  i1:=count;
  clrscr;
  for i:=0 to 25 do
  begin
    gotoxy(i div 13+40*(i div 13),i-12*(i div 13));
    writeln('Частота символа ',char(i+byte('A')),' - ',
           freq[i]/i1:1:3);
  end;
end.
```

8.12. Простые примеры непосредственной работы с видеобуфером.

Рассмотрим некоторые варианты "бегущих" строк, выполняемых прямой записью в видеобуфер:

1.Горизонтальная строка движется горизонтально.

(*Инициализируем указатель vb на видеобуфер в 3-м текстовом режиме значением числового адреса видеобуфера - и дальше обращаемся с ним как с обычным одномерным массивом.*)

```
uses crt,strings;

const
  vb:PChar=PChar($b8000000);
  fam:PChar='Фамилия';  (*Эта строка будет двигаться*)
var
  i,j:integer;

begin
  clrscr;
```

(*По видеопамяти будем двигаться с шагом 2 (индекс i) по четным байтам кодов символов, а по строке с шагом 1 (индекс j)*)

```
  for i:=0 to 1999 do
  begin
    for j:=0 to strlen(fam)-1 do
     vb[i*2+2*j]:=fam[j];(*Выводим строку*)
    delay(200);         (*Задержка 200 мс*)
    vb[i*2]:=' ';     (*Стираем букву*)
  end;
end.
```

Остальные варианты не будем подробно комментировать - ничем, кроме формирования индексов и процедур стирания "следа" от предыдущего вывода они не отличаются - разберитесь и составьте другие варианты движения строк, например по диагоналям.

2.Вертикальная строка движется вертикально

```
uses crt,strings;

const
  vb:PChar=PChar($b8000000);
  fam:PChar='Фамилия';  (*Эта строка будет двигаться*)
var
  i,j:integer;

begin
  clrscr;
  for i:=0 to 25 do
  begin
    for j:=0 to strlen(fam)-1 do
     vb[i*2*80+80*2*j]:=fam[j];(*Выводим строку*)
    delay(200);
    vb[i*2*80]:=' ';
  end;
end.
```

3.Горизонтальная строка бежит вертикально.

```
uses crt,strings;

const
  vb:PChar=PChar($b8000000);
  fam:PChar='Фамилия';  (*Эта строка будет двигаться*)
var
  i,j:integer;

begin
  clrscr;
  for i:=0 to 25 do
  begin
    for j:=0 to strlen(fam)-1 do
     vb[i*2*80+2*j]:=fam[j];(*Выводим строку*)
    delay(200);
    for j:=0 to strlen(fam)-1 do
     vb[i*2*80+2*j]:=' ';
  end;
end.
```

4.Вертикальная строка бежит горизонтально.

```
uses crt,strings;

const
```

```
  vb:PChar=PChar($b8000000);
  fam:PChar='Фамилия';  (*Эта строка будет двигаться*)
var
  i,j:integer;

begin
  clrscr;
  for i:=0 to 159 do
  begin
    for j:=0 to strlen(fam)-1 do
     vb[i*2+2*80*j]:=fam[j];(*Выводим строку*)
    delay(200);
    for j:=0 to strlen(fam)-1 do
     vb[i*2+2*80*j]:=' ';
  end;
end.
```

8.13. Пример работы с таблицей знакогенератора.

Задача: подменить таблицу знакогенератора своей, которая будет отличаться только изображением одного из символов псевдографики и вывести этот символ на экран для демонстрации, а после нажатия произвольной клавиши восстановить стандартную таблицу и вывести символ с тем же кодом.

Необходимая справочная информация в разделе 4.9.10.:

```
uses crt,dos;

var
  tz:PChar;  (*указатель на таблицу знакогенератора будет здесь*)
  h,     (*здесь будет высота символа*)
  s:byte;     (*число строк в рисунке символа*)
  r:registers;(*Структура для работы с регистрами через процедуру генерации прерывания intr()*)
  buf:PChar; (*Для указателя на буфер самодельной таблицы*)
  i,j:integer;

const
(*Рисунок нашего символа*)
ch:array [0..15]of char =(#$00, #$00, #$22, #$66, #$e7, #$24, #$24, #$24, #$a5,#$a5,#$a5,#$ff,#$ff,#$ff,#$ff,#$00);
begin
(*Заполняем регистровую структуру и вызываем прерывание *)
  r.ax:=$1130;
  r.bx:=$0600;
  intr($10,r);
  tz:=Ptr(r.es,r.bp);(*Получили адрес таблицы *)
  s:=r.cx;          (*Получили высоту символа*)
  (*Выделим память в куче под свою таблицу того же размера, что и стандартная и s байтов для своего рисунка*)
```

```
  GetMem(buf,256*s);
  (*Копируем стандартную таблицу в свой буфер*)
  for i:=0 to 255 do
    for j:=0 to 15 do
     buf[i*s+j]:=tz[i*s+j];
  (*Меняем рисунок символа с кодом $d4 на свой*)
  for i:=0 to s-1 do
    buf[s*$d4+i]:=ch[i];
  (*Очищаем экран и выводим символ стандартным рисунком*)
  clrscr;
  for i:=1 to 38 do
    write(#$d4);
  writeln;
  readkey;
  (*Подставляем адрес своей таблицы вместо стандартной*)
  r.ax:=$1100; r.es:=Seg(buf^); r.bp:=Ofs(buf^);
  r.cx:=256; r.dx:=0; r.bx:=$1000;
  intr($10,r);
  readkey;(*Любуемся проделанным*)
  (*Возвращаем старую таблицу*)
  r.ax:=$1104;r.bx:=0;intr($10,r);
  FreeMem(buf,256*s);
end.
```

8.14. Прикладное программирование в среде (Турбо, Борланд) Паскаль на более сложных примерах.

8.14.1. Обработка одномерных числовых массивов (векторов).

Общие сведения о задачах этого класса приведены в 4.9.11.

1.Мы начнем с одной простой задачи обработки файлового массива, затем создадим многофункциональную программу обработки и попытаемся постепенно совершенствовать диалог с пользователем для повышения комфортности в его работе с нашей программой.

```
(*Программа заполнения файла N случайными числами*)
const
  N=200;
  M=100;
var
  f:text;
  i:integer;
begin
```

```
  if ParamCount<>1then
  begin
  writeln('Задайте имя файла в командной строке');
exit;
  end;
  {$I-}
  Assign(f,ParamStr(1)); ReWrite(f);(*Открываем файл для записи*)
  if wordbool(IOResult)then
  begin
  writeln('Неудача при открытии файла ',ParamStr(1)); exit;
  end;
  for i:=0 to N-1 do
    write(f,M*(longint(random($7FFF))-($7FFF shr 1))/32767.,' ');
  close(f);
  {$I+}
end.
```

Теперь у нас есть файл для обработки с именем rndvect.txt - это имя мы и будем задавать в командной строке.

```
uses crt,strings,windos;

procedure proc0(fname:string;n:word);forward;
```

(*Это прототип функции обработки - ее определение мы составим позже. Функция не будет возвращать значение, а аргументами ее будут указатель на строку с именем файла и размер массива (количество чисел в нем). В объявлении прототипа должны присутствовать возвращаемый тип, имя функции и в круглых скобках список типов аргументов - имена аргументов (формальных параметров) не обязательны в прототипе, т.к. у прототипа нет тела, нет операторов обработки - сразу за круглой скобкой ставится точка с запятой или запятая (при объявлении списка прототипов функций, отличающихся только именами).*)

(*Пусть подпрограмма обработки определяет среднее арифметическое элементов массива - для этого мы можем читать по одному элементу и тут же его подсуммировать.*)

```
procedure proc0(fname:string;n:word);
var
  f:text;
  d:real;(*Для приема числа*)
  i:integer;
const
  sum:real=0;(*Для подсуммирования*)
begin
  (*Откроем файл для чтения*)
  {$I-}
  Assign(f,fname); Reset(f);
  if wordbool(IOResult)then
  begin
```

```
    writeln('Неудача при открытии файла ',fname); exit;
  end;
  (*Читаем слова из файла с преобразование к real и суммируем*)
  for i:=1 to n do
  begin read(f,d); sum:=sum+(d/n); end;
  (*Закрываем ненужный больше файл*)
  close(f);
  {$I+}
  (*Выводим результат обработки*)
  clrscr;
  writeln('Среднее арифметическое = ',sum:5:3);
end;

var
  r:word;(*Для размера массива*)
begin
  if ParamCount<1then
  begin
    write('Нет имени файла  в командной строке'); exit;
  end;
  (*Можно начинать диалог с пользователем*)
  while byte(false)=0 do
  begin
    clrscr;
    write(#$D#$A'Введите размер массива
    (0 - для выхода) и Enter : ');
    readln(r);   (*Принимаем размер массива в r*)
    if not wordbool(r)then
     break;  (*При 0-м размере прекращаем работу*)
    flush(input);
    proc0(ParamStr(1),r);  (*Иначе вызываем функцию обработки*)
    readkey;
  end;
end.
```

2.Многофункциональная программа обработки векторов.

Составлять отдельные программы на каждый вид обработки массивов неудобно, рациональнее предоставить весь комплекс услуг в одной программе. Вначале создадим массив текстовых строк - перечень выполняемых программой обработок массива, каждой такой строке будет поставлена в соответствие подпрограмма обработки.

Пусть таких различных функций в нашей программе будет например 15 - этот список вы можете расширить или сократить при необходимости. Мы не будем приводить решения всех перечисленных задач, они достаточно однообразны и вам предстоит для приобретения навыков решить их самостоятельно или с помощью преподавателя. Мы приведем реализацию 2-х функций - под номером 0 и 15, отличающихся наличием и отсутстви-

ем фрагмента выделения памяти под обрабатываемый массив; кроме того 15-я функция демонстрирует использование указателя на функцию в качестве параметра другой функции.

```
uses crt,strings,windos;
{$R-}

(*Прототипы 2-х функций обработки, которые мы определим позже*)

procedure proc0(fname:string;n:word);forward;
procedure proc15(fname:string;n:word);forward;

     (*Для остальных подпрограмм мы сразу приведем определения
- "пустышки" для последующей самостоятельной проработки*)

procedure proc1(fname:string;n:word);begin end;
procedure proc2(fname:string;n:word);begin
(*Подпрограмма - заглушка*)
  window(42,15,80,18);
  clrscr;
  writeln('Эту задачу вы должны программировать сами');
  readkey;
end;
procedure proc3(fname:string;n:word);begin end;
procedure proc4(fname:string;n:word);begin end;
procedure proc5(fname:string;n:word);begin end;
procedure proc6(fname:string;n:word);begin end;
procedure proc7(fname:string;n:word);begin end;
procedure proc8(fname:string;n:word);begin end;
procedure proc9(fname:string;n:word);begin end;
procedure proc10(fname:string;n:word);begin end;
procedure proc11(fname:string;n:word);begin end;
procedure proc12(fname:string;n:word);begin end;
procedure proc13(fname:string;n:word);begin end;
procedure proc14(fname:string;n:word);begin end;
procedure proc16(fname:string;n:word);begin end;
procedure proc17(fname:string;n:word);begin end;

     (* Для каждой задачи у нас есть ее изложение в виде строки
текста - позже мы собираемся использовать его при выводе списка
предоставляемых программой услуг по обработке одномерных массивов. К этим строкам надо "привязать" функции решения поставленных задач - и эту привязку мы осуществляем организацией массива структур с 2-мя полями - комментариями и указателями на функции.*)

type
  horror=procedure(fname:string;n:word);
  compfunc=function(var a,b):integer;
  preal=^real;
  areal=array[0..0]of real;
```

```
  pareal=^areal;
  pbyte=^byte;
  abyte=array[0..0]of byte;
  pabyte=^abyte;


type list_task=record
  _str:string;    (*Поле - указатель на строку названия задачи*)
  proc:horror;(*Поле указатель на функцию решения*)
end;

const
  task:array[0..18] of list_task=(
     (*Каждому полю мы сразу присваиваем значение, а
для удобства последующего вывода конец списка обозна-
чаем нуль - указателями на строку и на функцию*)

(_str:'Среднее арифметическое элементов массива';proc:proc0),
(_str:'Центрировать массив относительно среднего';proc:proc1),
(_str:'Среднее квадратов элементов центрированного массива';
proc:proc2),
(_str:'Среднее поэлементного произведения 2-х центрированных
массивов';proc:proc3),
(_str:'Скалярное произведение 2-х векторов';proc:proc4),
(_str:'Модуль массива как вектора и направляющие косинусы';
proc:proc5),
(_str:'Сумма и разность 2-х векторов';proc:proc6),
(_str:'Проекция 1-го вектора на направление 2-го';proc:proc7),
(_str:'Угол между 2-мя векторами в градусах';proc:proc8),
(_str:'Максимальное, минимальное значение и их индексы';
proc:proc9),
(_str:'Максимальное и минимальное абсолютных значений';
proc:proc10),
(_str:'Суммы положительных и отрицательных значений и их количе-
ства';proc:proc11),
(_str:'Произведение отличных от 0 положительных и
отрицательных';proc:proc12),
(_str:'Произведения отличных от нуля с четными и нечетными индек-
сами';proc:proc13),
(_str:'Отсортировать массив по неубыванию методом пузырька';
proc:proc14),
(_str:'Отсортировать массив быстрой сортировкой';proc:proc15),
(_str:'Новый массив циклическим сдвигом старого влево на 5 эле-
ментов';proc:proc16),
(_str:'Новый массив делением эл-тов исходного на максим по абс
знач';proc:proc17),
(_str:'';proc:proc17)
);
```

(*Теперь нам придется определить две функции обработки, у которых пока есть только прототипы - это функция определения среднего арифметического всех элементов массива и функция сортировки массива.*)

```
(*Среднее арифметическое элементов массива*)
procedure proc0(fname:string;n:word);
var
  f:text;  d:real;(*Для приема числа*)
  i:integer;
const
  sum:real=0;(*Для подсуммирования*)
begin
  (*Откроем файл для чтения*)
  {$I-}
  Assign(f,fname);  Reset(f);
  window(42,10,80,25);
  if wordbool(IOResult)then
  begin
  writeln('Неудача при открытии файла ',fname); exit;
  end;
  (*Читаем слова из файла с преобразование к real и суммируем*)
  for i:=1 to n do
  begin
    read(f,d); sum:=sum+(d/n);
  end;
  (*Закрываем ненужный больше файл*)
  close(f);
  {$I+}
  (*Выводим результат обработки*)
  clrscr;
  writeln('Среднее арифметическое = ',sum:5:3);
end;
```

(*Функция сравнения элементов типа double - указатель на нее надо дать аргументом функции пузырьковой сортировки bubblesort().*)

```
function cmpf(var a,b):integer;
begin
  if preal(a)^=preal(b)^then  cmpf:=0;
  if preal(a)^<preal(b)^then  cmpf:=-1;
  if preal(a)^>preal(b)^then  cmpf:=1;
end;
```

(*Подпрограмма, меняющая местами содержимое 2-х объектов в памяти, заданных своими указателями и размером*)

```
procedure swapobj(obj1,obj2:pointer;size:integer);
var
  tmp:pointer;
begin
```

```
 (*выделим память для временного сохранения одного из объек-
тов*)
  GetMem(tmp,size);
  move(obj1^,tmp^,size); (*перегрузим в "запасник" один объект*)
  move(obj2^,obj1^,size);(*второй на место первого*)
  move(tmp^,obj2^,size); (*из запасника в область памяти второго*)
  FreeMem(tmp,size);  (*освободим арендованную память*)
end;

procedure bubblesort(var a;n,size:word;cmpf:compfunc);
var
  i,j,t1,t2:longint;
begin
  for i:=0 to n-1 do
    for j:=i+1 to n-1 do
    begin
      t1:=longint(a)+i*size;  t2:=longint(a)+j*size;
      if cmpf(t1,t2)=1then
      swapobj(@pabyte(a)^[i*size],@pabyte(a)^[j*size],size);
    end;
end;


(*Отсортировать массив пузырьковой сортировкой*)
procedure proc15(fname:string;n:word);
var
  f:text;  a:pareal; i:integer;
begin
  (*Откроем файл для чтения*)
  {$I-}
  Assign(f,fname);  Reset(f);
  if wordbool(IOResult)then
  begin
    (*Окно для вывода результирующего массива*)
    window(42,10,80,23);
    writeln('Неудача при открытии файла ',fname);
    exit;
  end;
  (*Выделим память для массива*)
  GetMem(a,n*sizeof(real));
  (*Читаем слова из файла с преобразование к real в массив a*)
  for i:=0 to n-1 do  read(f,a^[i]);
  (*Закрываем ненужный больше файл*)
  close(f);
  {$I+}
  (*Сортируем с помощью ф-ции bubblesort()*)
  bubblesort(a, n, sizeof(real), cmpf);
  (*Выводим результат обработки*)
  window(42,10,80,23);
  clrscr;
  for i:=0 to n-1 do
    write(a^[i]:5:3,' ');
```

```
  FreeMem(a,n*sizeof(real));(*Освобождаем арендованную память*)
end;

{$I var2.inc}
```

(*В многофункциональной программе придется дополнить диалог с пользователем возможностью выбора выполняемой функции обработки. Для начала мы сделаем это так же, как принимали размер массива, а затем попытаемся улучшить сервис нашей программы *)

```
var
  r, (*Для размера массива int*)
  nf:word;(*Для номера функции*)
begin
  if ParamCount<1then
  begin
  writeln('Нет имени файла в командной строке'); exit;
  end;
  (*Можно начинать диалог с  пользователем*)
  while boolean(3456) do
  begin
    window(1,1,80,25);
    clrscr;
    write('Введите размер массива (0  -  для выхода) и Enter : ');
    read(r); (*Принимаем  размер  массива  в  r*)
    (*При 0-м размере прекращаем работу*)
    if not wordbool(r)then  break;
flush(input);  (*Очищаем буфер ввода (клавиатурный)*)
write('Введите N ф-ции (от 0 до 17, -1 для выхода) и Enter : ');
read(nf);  (*Принимаем номер функции в nf*)
if(nf<0)then break;  (*При отриц номере  прекращаем работу*)
    if nf>17then  continue;
    flush(input);  (*Очищаем буфер ввода  (клавиатурный)*)
    (*Временный вариант - если это уже решенная задача  0*)
    if(nf=0)then(*Вызываем функцию  обработки  proc0*)
     task[nf].proc(ParamStr(1),r)
    else         (*Иначе вызываем 'заглушку' proc1*)
     task[1].proc(ParamStr(1),r);
    readkey;
  end;
end.
```

2.1.Фильтрованный посимвольный прием ввода пользователя.

Человеку свойственно ошибаться - и пользователь вашей программы будет это делать при вводе запрашиваемых у него данных. Если, например, при наборе номера задачи или размера массива он введет нецифровой символ, функция форматированного приема строки не сможет преобразовать ее в целое и программа аварийно завершится.

В диалоговых программах это недопустимо - всегда надо давать возможность пользователю исправить свою ошибку. Это можно осуществить по разному - принимать, например строку вначале без преобразования в целое, анализировать наличие недопустимых (в нашем случае нецифровых) символов и возвращаться при необходимости к повторению ввода сначала.

В этом разделе мы познакомим вас с другим простым подходом - контролируемым или фильтрованным посимвольным приемом вводимых данных, при котором каждый символ принимается с клавиатуры отдельно без эха на экран, анализируется на допустимость и при положительном результате отображается на экране и заносится в строку. Если он ошибочен - то просто игнорируется и так до нажатия Enter, после которого строка преобразуется в число и далее все как и раньше. Так мы просто отсекаем попытки ввести недопустимый символ. Разумеется, это работает в простейших случаях, а в более сложных приходится комбинировать фильтрацию с анализом.

```
{$I var2.inc}
const
  Enter=#13; (*Фиксируем код клавиши Enter*)
var
  r, (*Для размера массива int*)
  nf:word;    (*Для номера функции*)
  ch:char;    (*Для приема символа без отображения*)
  s:string;   (*Массив для промежуточного приема строки*)
  i:integer;
begin
  if ParamCount<1then
  begin
    writeln('Нет имени файла в командной строке'); exit;
  end;
  (*Можно начинать диалог с  пользователем*)
  while boolean(3456) do
  begin
    window(1,1,80,25); clrscr;
    write(#$D#$A'Введите размер массива
         (0  -  для выхода) и Enter : ');
    (*Принимаем размер массива в s посимвольно до Enter*)
    i:=1;  ch:=readkey;
    while(ch<>Enter)do
    begin
     if(ch>='0')and(ch <='9')then (*Если принятый символ цифра*)
     begin
       s[i]:=ch;      (*Заносим его в строку s*)
        write(ch);           (*Выводим его на экран*)
        inc(i);     (*Наращиваем счетчик символов*)
      end;
      ch:=readkey;           (*Читаем следующий*)
```

```
    end;
    s[0]:=char(i-1);     (*Корректируем длину строки*)
    Val(s,r,i);                (*Преобразуем в целое*)
    if not wordbool(r)then
     break;   (*При 0-м размере прекращаем работу*)
    flush(input); (*Очищаем буфер ввода (клавиатурный)*)
    write(#$D#$A'Введите N ф-ции (от 0 до 17, -1 для выхода) и
Enter: ');
    (*Повторяем прием в s номера задачи до Enter*)
    i:=1;   ch:=readkey;
    while(ch<>Enter)do
    begin
     if(ch>='0')and(ch <='9')then (*Если принятый символ цифра*)
     begin
       s[i]:=ch;   (*Заносим его в строку s*)
       write(ch);           (*Выводим его на экран*)
       inc(i);   (*Наращиваем счетчик символов*)
     end;
     ch:=readkey;           (*Читаем следующий*)
    end;
    s[0]:=char(i-1);  (*Корректируем длину строки*)
    Val(s,nf,i);               (*Преобразуем в целое*)
    if(nf<0)then
     break;   (*При отриц номере  прекращаем работу*)
    if nf>17then  continue;
    flush(input);     (*Очищаем буфер ввода  (клавиатурный)*)
    (*Временный вариант - если это уже решенная задача 0*)
    if(nf=0)then (*Вызываем функцию обработки proc0*)
     task[nf].proc(ParamStr(1),r)
    else         (*Иначе вызываем 'заглушку' proc1*)
     task[1].proc(ParamStr(1),r);
    readkey;
  end;
end.
```

2.2.Выбор услуг программы из списка услуг (меню услуг).

Для набора номера задачи в рассмотренном варианте программы пользователю придется иметь допустимый перечень задач на бумаге - это не очень удобно; было бы желательно предварительно вывести ему на экран перечень видов обработки массивов и дать возможность выбора с помощью например курсорных клавиш. Для этого нам придется составить соответствующую подпрограмму, которая будет возвращать номер выбранной строки и этот номер будет использоваться основной программой для запуска соответствующей функции обработки. Этим и займемся.

```
{$I var2.inc}
```

(*Возвращает полный код нажатой клавиши, в отличие от readkey*)

```
function bioskey:integer;
var
  r:TRegisters;
begin
  r.ah:=0;
  intr($16,r);
  bioskey:=r.ax;
end;
```

(*Подпрограмма, возвращающая номер выбранной пользователем строки из экранного списка строк. Может использоваться как модель меню при запуске на выполнение различных подпрограмм, в том числе по индексу из массива подпрограмм (массива указателей на функции)*)

(*Нам понадобится работа со спецклавишами управления курсором - чтобы не осуществлять анализ типа нажатой клавиши и повторного чтения для получения старшего байта, мы будем работать с двухбайтовыми кодами и определим их в виде констант*)

```
const
UP=18432;DOWN=20480;HOME=18176;ENDKEY=20224;
ENTER=7181;ESC=283;
ROW_COUNT=4;  (*Количество строк в    окне*)
MAX_ELEM=50; (*Предельное количество  строк*)
STR_LEN=72;  (*Длина строк в окне*)
WNDYBEG=2; (*Начальная строка окна на экране*)
```

(*Шаблон оконных строк - вы можете создать его в любом текстовом редакторе, оснащенном макросами для рисования рамок*)

```
  strwnd:array[0..2]of string=(
'╔═══════════════════════════════════════════════════════╗',
'║                                                       ║',
'╚═══════════════════════════════════════════════════════╝'
);
```

(*Атрибуты для нормальной и выделенной строк и строка пробелов*)

```
  norm_attr:byte=$1f;
  spec_attr:byte=$3f;
  space_str:string=
  '                                                       ';
var
  count:byte;  (*Для общего количества подлежащих выводу строк*)

(*Функция вывода окна для списка строк*)
procedure str_wnd;
var
  i:integer;
begin
  window(1,WNDYBEG,STR_LEN+1,WNDYBEG+ROW_COUNT+3+1);
  textattr:=norm_attr;
```

```
  writeln(strwnd[0]); (*Выводим верхнюю рамочную*)
  for i:=0 to ROW_COUNT do
    writeln(strwnd[1]);(*Содержательные*)
  writeln(strwnd[2]);         (*Нижнюю рамочную*)
  window(2,WNDYBEG+1,STR_LEN{-1},WNDYBEG+ROW_COUNT+1);
end;

const
    index:word=0;
```
offs:word=0; (*index+offs - индекс в массиве структур задач task; считаем, что весь список задач не может поместиться по высоте окна и при 'наезде' на его нижнюю границу будет наращиваться offs, если список не исчерпан*)

```
var
  temp:word;

(*функция вывода массива строк*)
function strput:integer;
var
```
i,key,string_number:integer; (*для номера выбранной строки*)
```
begin
```
(*Необходимо подсчитать общее количество строк в массиве задач*)
```
  i:=0;
  while wordbool(Length(task[i]._str)) do
  inc(i); count:=i;
  (*Выводим массив строк меню*)
  index:=0;
  while(index<count)and(index<ROW_COUNT)do
  begin
    gotoxy(1,1+index mod ROW_COUNT);
    writeln(task[index]._str);
    inc(index);
  end;
  index:=0; temp:=0; offs:=0; (*Начальные позиции*)
  (*Перекрашиваем текущую строку*)
  textattr:=spec_attr;
  gotoxy(1,(1+index mod ROW_COUNT)); writeln(space_str);
  gotoxy(1,(1+index mod ROW_COUNT)); writeln(task[index]._str);
  gotoxy(1,(1+index mod ROW_COUNT)); textattr:=norm_attr;

  (*Теперь двигаемся по строкам в окне*)
  while true do
  begin
    key:=bioskey;  (*Получаем двухбайтовый код нажатой клавиши
*)
    case(key)of (*Анализируем его сравнением с константами*)
      ESC:
      begin
        strput:=-1; (*Клавиша ESC вызывает выход*)
        break;
```

```
      end;
      UP:       (*Клавиша стрелка вверх*)
      begin
(*Если строка 0 и вверх некуда, но индекс массива не 0*)
     if(index+offs>0)and(index mod ROW_COUNT=0)and(offs>0)then
     begin
          (*Вернем окраску текущей строке*)
writeln(space_str);            gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str); gotoxy(1,(1+index mod ROW_COUNT));
dec(offs); gotoxy(1,(1+index mod ROW_COUNT));   temp:=index;
(*Выводим - вначале строки пробелов для очистки прошлого*)
          for index:=0 to ROW_COUNT-1 do
          begin
 gotoxy(1,(1+index mod ROW_COUNT)); writeln(space_str);
          end;
          (*а затем строки*)
          index:=0;
          while(index+offs<count)and(index<ROW_COUNT)do
          begin
 gotoxy(1,(1+index mod ROW_COUNT));
 writeln(task[index+offs]._str);  inc(index);
          end;
          (*Перекрасим строку без изменения позиции курсора*)
          index:=temp; gotoxy(1,(1+index mod ROW_COUNT));
          textattr:=spec_attr;  writeln(space_str);
          gotoxy(1,(1+index mod ROW_COUNT));
          writeln(task[index+offs]._str);
          gotoxy(1,(1+index mod ROW_COUNT));
          textattr:=norm_attr;
        end
        else
          if(index+offs>0)then
          begin    (*В средней части окна и списка*)
            (*Вернем окраску текущей строке*)
writeln(space_str); gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
dec(index); gotoxy(1,(1+index mod ROW_COUNT));
textattr:=spec_attr;        (*Выделим цветом новую строку*)
writeln(space_str); gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
textattr:=norm_attr;
          end;
      end;
      DOWN:
      begin
(*Если достигнута нижняя граница, но список вывода не исчерпан*)
if(index+offs<count-1)and((index  mod  ROW_COUNT)=(ROW_COUNT-
1))
then   begin
writeln(space_str); gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
```

```
inc(offs);
gotoxy(1,(1+index mod ROW_COUNT));
textattr:=spec_attr;
writeln(space_str);gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
textattr:=norm_attr;
(*Выводим -вначале строки пробелов для очистки прошлого*)
temp:=index;
for index:=0 to ROW_COUNT-1 do
begin
gotoxy(1,(1+index mod ROW_COUNT));writeln(space_str);
end;
(*а затем массив строк*)
index:=0;
while(index+offs<count)and(index<ROW_COUNT)do
  begin
gotoxy(1,(1+index mod ROW_COUNT));writeln(task[index+offs]._str);
inc(index);
end;
(*Перекрасим строку без изменения позиции курсора*)
index:=temp;
gotoxy(1,(1+index mod ROW_COUNT));
textattr:=spec_attr;
writeln(space_str); gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
textattr:=norm_attr;
   end
       else
if(index+offs+1<count)and(ROW_COUNT+index           mod
ROW_COUNT+1<count)
then
    begin
writeln(space_str);gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
inc(index); gotoxy(1,(1+index mod ROW_COUNT));
textattr:=spec_attr; writeln(space_str);
gotoxy(1,(1+index mod ROW_COUNT));writeln(task[index+offs]._str);
gotoxy(1,(1+index mod ROW_COUNT));
textattr:=norm_attr;
    end;
     end;
     HOME:
     begin
writeln(space_str); gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
       if(offs>0)then
       begin
        offs:=0;
(*Выводим -вначале строки пробелов для очистки прошлого*)
        for index:=0 to ROW_COUNT-1 do
        begin
```

```
gotoxy(1,(1+index mod ROW_COUNT));writeln(space_str);
         end;
        end;
        (*а затем массив str*)
        index:=0;
        while(index<count)and(index<ROW_COUNT)do
        begin
gotoxy(1,(1+index mod ROW_COUNT));writeln(task[index+offs]._str);
inc(index);
        end;
index:=0;
        (*Перекрашиваем текущую строку*)
textattr:=spec_attr;gotoxy(1,(1+index mod ROW_COUNT));
writeln(space_str); gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
textattr:=norm_attr;
      end;(*HOME*)
      ENDKEY:
      begin
writeln(space_str);gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
        if(offs>0)or((offs=0)and(count>ROW_COUNT))then
        begin
          offs:=count-ROW_COUNT;
(*Выводим -вначале строки пробелов для очистки прошлого*)
          for index:=0 to ROW_COUNT-1 do
          begin
gotoxy(1,(1+index mod ROW_COUNT));writeln(space_str);
          end;
          (*а затем массив str*)
          index:=0;
          while(index<count)and(index<ROW_COUNT)do
          begin
gotoxy(1,(1+index mod ROW_COUNT));writeln(task[index+offs]._str);
            inc(index);
          end;
          index:=ROW_COUNT-1;
        end;
        if(count<=ROW_COUNT)then  index:=count-1;
        (*Перекрашиваем текущую строку*)
textattr:=spec_attr;gotoxy(1,(1+index mod ROW_COUNT));
writeln(space_str);gotoxy(1,(1+index mod ROW_COUNT));
writeln(task[index+offs]._str);gotoxy(1,(1+index mod ROW_COUNT));
textattr:=norm_attr;
      end;
      ENTER:
      begin
(*Записываем номер текущей строки в string_number и выходим*)
string_number:=index+offs;strput:=string_number; break;
      end;
    end;(*case*)
```

```
  end;
end;

{$I var4.inc}

(*Главная функция для этого варианта имеет вид:*)
var
  r, (*Для размера массива int*)
  nf:word;(*Для номера функции*)
  ch:char;    (*Для приема символа без отображения*)
  s:string;(*Массив для промежуточного приема строки*)
  i:integer;
begin
  if ParamCount<1then
  begin
  writeln('Нет имени файла в командной строке'); exit;
  end;
  (*Можно начинать диалог с пользователем*)
  while true do
  begin
    window(1,1,80,25);
    textattr:=$07;
    clrscr;
    write(#$D#$A'Введите размер массива
    (0  -  для выхода) и Enter : ');
    (*Принимаем размер массива в s посимвольно до Enter*)
    i:=1; ch:=readkey;
    while ch<>char(Enter)do
    begin
      if(ch>='0')and(ch <='9')then (*Если принятый символ цифра*)
      begin
        s[i]:=ch;           (*Заносим его в строку s*)
        write(ch);         (*Выводим его на экран*)
        inc(i);       (*Наращиваем счетчик символов*)
      end;
      ch:=readkey;         (*Читаем следующий*)
    end;
    s[0]:=char(i-1);   (*Корректируем длину строки*)
    Val(s,r,i);              (*Преобразуем в целое*)
    if not wordbool(r)then
      break;   (*При 0-м размере прекращаем работу*)
    flush(input); (*Очищаем буфер ввода (клавиатурный)*)
    str_wnd;  (*Шаблон окна для выбора задач*)
    nf:=strput;   (*Получение выбранной строки*)
    if(nf<0)then  break; (*При отриц номере прекращаем работу*)
    flush(input); (*Очищаем буфер ввода (клавиатурный)*)
    if nf>17then  continue;
    task[nf].proc(ParamStr(1),r); (*Вызываем функцию обработки*)
    readkey;
  end;
end.
```

2.3.Выбор файла из списка файлов.

Аналогичную списку функций услугу мы можем оказать пользователю, предоставив ему такой же метод выбора файла - для этого придется составить функцию, осуществляющую навигацию по каталогам и файлам в оконной таблице с помощью клавиш управления курсором. Количество файлов в каталоге может значительно превышать количество задач, но их имена короче - 12 символов максимум вместе с разделяющей имя и расширение точкой. Поэтому рационально сделать окно многоколонковым - мы сделаем его на 3 колонки, это как раз позволит занять половину экрана. Для перемещения по колонкам нам понадобятся горизонтальные клавиши - стрелки и мы добавим определение их кодов соответствующими константами LEFT, RIGHT. Имена остальных констант, имеющих тот же смысл, что и в предыдущей задаче, мы просто несколько изменим, чтобы избежать путаницы.

```
{$I var4.inc}

(*переводит символ в нижний регистр, возвращая результат*)
function tolower(c:char):char;
var
  d:byte;
begin
  d:=byte(c);
  if(d>=byte('A'))and(d<=byte('Z')) then inc(d,$20);
  if(d>=byte('A'))and(d<=byte('П')) then inc(d,$20);
  if(d>=byte('P'))and(d<=byte('Я')) then inc(d,$50);
  if(d=byte('Ё')) then  d:=byte('ё');tolower:=char(d);
end;


const
LEFT=19200;RIGHT=19712;COL_COUNT=3;ROW_CNT=12;MAX_EL=500;
  NAME_LEN=14;

okno:array[0..2]of string=(
'╔═══════════╦═══════════════════╦═══════════════════╗',
'║           ║                   ║                   ',
'╚═══════════╩═══════════════════╩═══════════════════╝');
space:string='          ';
var
fpath_name:array [0..128]of char;(*для имени выбранного файла*)

(*Вывод окна для списка файлов*)
procedure file_wnd;
var
  i:integer;
begin
```

```
  window(1,10,40+1,25);textattr:=norm_attr; writeln(okno[0]);
  for i:=0 to ROW_CNT-1 do writeln(okno[1]);writeln(okno[2]);
  gotoxy(2,2);
end;

var
  (*Для списка элементов каталога*)
  files:array[0..MAX_EL,0..NAME_LEN]of char;
  ffblk:TSearchRec; (*Структура для параметров файла*)
  s:array[0..80]of char;  (*Для названия текущего каталога*)
  dcount:integer;   (*Для количества элементов в каталоге*)

(*Нам придется отслеживать при перемещении по элементам в окне
-текущий индекс в массиве files;
-текущую строку и колонку в окне
-позицию курсора*)
const
  findex:word=0;
  foffs:word=0;
  ftemp:word=0;  (*findex+foffs -индекс в files*)

(*функция вывода содержимого текущей директории*)
procedure read_dir;
var
  chd,done,k,i,key:integer;
  tmp:array[0..NAME_LEN]of char;
begin
  s[0]:=#0;
  while boolean(random(10000)+1) do
  begin
    chd:=0;         (*признак смены текущего каталога*)
    gotoxy(1,1);
    writeln(okno[0]);(*Восстанавливаем верхнюю рамку*)
    (*определяем тек. директорию в s и выводим в рамку*)
    getcurdir(s,0);
    strcat(s,'\');
    if StrLen(s)>=Length(okno[0]) then
     s[Length(okno[0])]:=#0;
    gotoxy((Length(okno[0])-StrLen(s))div 2,1);
    writeln(s);
    (* занесем каталоги текущего каталога в files*)
    findfirst('*',$10 or $01,ffblk); findex:=0;
    while DosError=0 do
    begin
     StrCopy(files[findex],ffblk.name);
     files[findex][NAME_LEN-1]:=char(ffblk.attr);
     inc(findex); findnext(ffblk);
    end;
    findfirst('*.*',0,ffblk);
(* занесем файлы текущего каталога в files*)
    while DosError=0 do
```

```
    begin
      (*Попутно имена файлов - в нижний регистр*)
      for i:=0 to strlen(ffblk.name)-1 do
        ffblk.name[i]:=tolower(ffblk.name[i]);
      StrCopy(files[findex+foffs],ffblk.name);
      files[findex+foffs][NAME_LEN-1]:=char(ffblk.attr);
      inc(findex);
      findnext(ffblk);
    end;
    dcount:=findex;  (*Общее количество элементов в каталоге*)
    (*Остальное обнуляем*)
    while findex<MAX_EL do
    begin
      files[findex][0]:=#0;inc(findex);
    end;

    (*Для удобства использования сортируем по именам*)
    for i:=0 to dcount-2 do
      for k:=i+1 to dcount-1 do
        if(StrComp(files[i],files[k])>0)then
        begin
StrCopy(tmp,files[i]);
StrCopy(files[i],files[k]);
StrCopy(files[k],tmp);
        end;

    (*Выводим -вначале строки пробелов для очистки прошлого*)
    for findex:=0 to COL_COUNT*ROW_CNT-1 do
    begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(space);
    end;
    (*а затем массив files*)
findex:=0;
while(findex<dcount)and(findex<COL_COUNT*ROW_CNT)do
    begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex]);inc(findex);
    end;

foffs:=0; findex:=0; ftemp:=0; (*Начальные позиции*)

    (*Перекрашиваем текущую строку*)
textattr:=spec_attr;
gotoxy(2+(NAME_LEN-1)*(findex div ROW_CNT),2+findex mod
ROW_CNT); writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;

    (*Теперь двигаемся по именам в окне*)
```

```
    while true do
    begin
      key:=bioskey;
      Case(key)of
        ESC:exit;

        LEFT:
if(findex div ROW_CNT)>0then
begin (*Вернем окраску текущей строке*)
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);

          (*Переместимся в колонку левее*)
dec(findex,ROW_CNT);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);

          (*Перекрашиваем новую строку*)
textattr:=spec_attr; writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
        end;(*if  LEFT*)
        RIGHT:
if(findex div ROW_CNT<2)and
((findex     div     ROW_CNT+1)*ROW_CNT+findex     mod
ROW_CNT<dcount)then begin (*Вернем окраску текущей строке*)
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
          (*Переместимся в колонку правее*)
inc(findex,ROW_CNT);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
          (*Перекрашиваем новую строку*)
textattr:=spec_attr; writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
        end;(*if  RIGHT*)
        UP:
if(findex+foffs>0)and(findex div ROW_CNT=0)
and(findex mod ROW_CNT=0)and(foffs>0)then
        begin
          (*Вернем окраску текущей строке*)
```

```
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); dec(foffs);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); ftemp:=findex;
      (*Выводим -вначале строки пробелов для очистки
прошлого*)
for findex:=0 to COL_COUNT*ROW_CNT-1 do
         begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(space);
         end;
          (*а затем массив files*)
findex:=0;
while(findex+foffs<dcount)and(findex<COL_COUNT*ROW_CNT)do
         begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
writeln(files[findex+foffs]);inc(findex);
         end;

(*Перекрасим строку без изменения позиции курсора*)
findex:=ftemp;
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=spec_attr; writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
        end
        else
          if(findex+foffs>0)then
          begin
            (*Вернем окраску текущей строке*)
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); dec(findex);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=spec_attr;writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
          end;
        DOWN:
        if(findex+foffs<dcount-1)and(findex div ROW_CNT=2)
          and(findex mod ROW_CNT=ROW_CNT-1)then
        begin
```

```
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); inc(foffs);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=spec_attr; writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;

     (*Выводим -вначале строки пробелов для очистки прошлого*)
          ftemp:=findex;
          for findex:=0 to COL_COUNT*ROW_CNT-1 do
          begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(space);
          end;

          (*а затем массив files*)
          findex:=0;

while(findex<dcount)and(findex<COL_COUNT*ROW_CNT)do
          begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
inc(findex);
          end;

          (*Перекрасим строку без изменения позиции курсора*)
findex:=ftemp;
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=spec_attr; writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
        end
        else
if(findex+foffs+1<dcount)and ((findex div ROW_CNT)*ROW_CNT+findex
mod ROW_CNT+1<dcount)then
         begin
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); inc(findex);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=spec_attr; writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
```

```
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
          end;
         HOME:
         begin
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
       if(foffs>0)then
          begin
            foffs:=0;
       (*Выводим -вначале строки пробелов для очистки прошло-
го*)
            for findex:=0 to COL_COUNT*ROW_CNT-1 do
            begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(space);
            end;
            (*а затем массив files*)
            findex:=0;

     while(findex<dcount)and(findex<COL_COUNT*ROW_CNT)do
            begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
             inc(findex);
            end;
          end;
          findex:=0;
          (*Перекрашиваем текущую строку*)
textattr:=spec_attr;
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
writeln(space);gotoxy(2+(NAME_LEN-1)*(findex div
ROW_CNT),2+findex mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
        end;(* HOME*)
        ENDKEY:
        begin
writeln(space);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
     if(foffs>0)or((foffs=0)and(dcount>COL_COUNT*ROW_CNT))then
          begin
            foffs:=dcount-COL_COUNT*ROW_CNT;
      (*Выводим -вначале строки пробелов для очистки прошлого*)
            for findex:=0 to COL_COUNT*ROW_CNT-1 do
```

```
            begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); writeln(space);
            end;
            (*а затем массив files*)
            findex:=0;

     while(findex<dcount)and(findex<COL_COUNT*ROW_CNT)do
            begin
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex   mod
ROW_CNT); writeln(files[findex+foffs]);inc(findex);
            end;
            findex:=COL_COUNT*ROW_CNT-1;
          end;

         if(dcount<=COL_COUNT*ROW_CNT)then  findex:=dcount-1;
           (*Перекрашиваем текущую строку*)
textattr:=spec_attr;
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT);
writeln(space);gotoxy(2+(NAME_LEN-1)*(findex                div
ROW_CNT),2+findex mod ROW_CNT); writeln(files[findex+foffs]);
gotoxy(2+(NAME_LEN-1)*(findex   div   ROW_CNT),2+findex
mod ROW_CNT); textattr:=norm_attr;
        end;
        ENTER:
        begin
          (*Если стоим не на имени каталога то считаем
файлом и копируем его в fpath_name*)
     if not bytebool(byte(files[findex+foffs][13]) and $10)then
          begin
            StrCopy(fpath_name,files[findex+foffs]);
            (*и выходим*)
            exit;
          end;
          (*иначе делаем каталог текущим*)
if not wordbool(StrComp(files[findex+foffs],'.'))then chdir('\')
else chdir(files[findex+foffs]); chd:=1;
        end;
      end;(*Case*)
      if(chd<>0)then break;
    end;
  end;(*while boolean(random(10000)+1) do*)
end;

{$I var5.inc}

    (*При выборе имени файла из предоставляемого спис-
ка программа может выглядеть так:*)

var
```

```
  r,       (*Для размера массива int*)
  nf:word;     (*Для номера функции*)
  ch:char;     (*Для приема символа без отображения*)
  ps:string;  (*Массив для промежуточного приема строки*)
  i:integer;
begin
(*Можно начинать диалог с пользователем*)
  while boolean(1998) do
  begin
    window(1,1,80,25); textattr:=$07; clrscr;
    write(#$D#$A'Введите размер массива
         (0  -  для выхода) и Enter : ');
    (*Принимаем размер массива в ps посимвольно до Enter*)
    i:=1;
    ch:=readkey;
    while ch<>char(Enter)do
    begin
    if(ch>='0')and(ch <='9')then (*Если принятый символ цифра*)
     begin
       ps[i]:=ch;          (*Заносим его в строку ps*)
       write(ch);          (*Выводим его на экран*)
       inc(i);       (*Наращиваем счетчик символов*)
     end;
     ch:=readkey;          (*Читаем следующий*)
    end;
    ps[0]:=char(i-1);  (*Корректируем длину строки*)
    Val(ps,r,i);             (*Преобразуем в целое*)
    if not wordbool(r)then
     break;       (*При 0-м размере прекращаем работу*)
    flush(input); (*Очищаем буфер ввода (клавиатурный)*)
    str_wnd;      (*Шаблон окна для выбора задач*)

    nf:=strput;     (*Получение выбранной строки*)
    if(nf<0)then
     break;  (*При отриц номере прекращаем работу*)
    flush(input);(*Очищаем буфер ввода (клавиатурный)*)
    if nf>17then  continue;
    file_wnd;(*рисуем окно для вывода файлов*)
    read_dir;(*Выводим текущий каталог в окно*)
    (*Вызываем функцию обработки*)
    task[nf].proc(StrPas(fpath_name),r);
    readkey;
  end;
end.
```

2.4.Стандартный набор функций для работы с векторами.

Мы приведем определения наиболее употребительных общего характера функций для работы с одномерными массивами (векторами). Они оформлены в виде модуля - вы можете скомпилировать его в .tpu файл и вызывать в своих программах при необходимости.

{модуль, содержащий наиболее общие функции для работы с векторами}
```
unit vectors;
{$R-}
interface
(* Для работы с векторами нам понадобится два специальных типа :
безразмерный массив и указатель на него для создания векторов
любой размерности. По аналогии с языком Си, будем индексировать
такие вектора от 0 до n-1 *)

  type
    vector=array[0..0] of real;
    pvector=^vector;
  function CreateEmptyVector(n:longint):pvector;
  function CreateFilledVector(n:longint;whatcopy:pvector):pvector;
  function LoadVector(n:longint;f:string):pvector;
  procedure WriteVector(n:longint;vec:pvector;f:string);
  procedure PrintVector(n:longint;vec:pvector);


implementation

(*Эта функция пытается создать пустой вектор заданной
размерности, попутно очищая его*)
function CreateEmptyVector(n:longint):pvector;
var
  vec:pvector;
  i:longint;
begin
  CreateEmptyVector:=nil;
  if(n<=0) then(*при некорректном размере вектора*)
  begin
    writeln(n,' не может быть размерностью вектора');
    exit;(*возвращаем нулевой указатель*)
  end;
  (*пытаемся выделить память под запрашиваемое число
  элементов двойной точности *)
  GetMem(vec,n*sizeof(real));
  for i:=0 to n-1 do
    vec^[i]:=0;{обнуляем n составляющих вектора}
  CreateEmptyVector:=vec;  (*и возвращаем указатель на него*)
end;


(*Эта функция пытается создать пустой вектор заданной размерно-
сти, а затем заполнить его данными из другого вектора*)
function CreateFilledVector(n:longint;whatcopy:pvector):pvector;
var
  vec:pvector;
  i:longint;
begin
```

```
  vec:=CreateEmptyVector(n);(*создаём пустой вектор*)
  CreateFilledVector:=vec;
  if(vec=nil) then{если не удалось выделить запрошенный блок}
    exit;(*возвращаем нулевой указатель*)
  for i:=0 to n-1 do
    vec^[i]:=whatcopy^[i];(*копируем n составляющих вектора*)
  CreateFilledVector:=vec;(* и возвращаем указатель на него*)
end;

(*эта функция пытается загрузить вектор, лежащий в
файле в виде v2 ... vn, где vi - его составляющие*)
function LoadVector(n:longint;f:string):pvector;{f - файл с вектором}
var
  vec:pvector;  i:longint;  fp:text;
begin
  vec:=CreateEmptyVector(n);(*конструируем пустой вектор*)
  LoadVector:=vec;
  if(vec=nil) then(*если не удалось выделить запрошенный блок*)
    exit;(*возвращаем нулевой указатель*)
  {$I-}
  Assign(fp,f);(*пытаемся открыть файл*)
  if(IOResult<>0) then(*если не удалось открыть файл*)
  begin
    writeln('Не могу открыть файл ',f);
    FreeMem(vec,n*sizeof(real));(*освобождаем память *)
    LoadVector:=Ptr(0,0);(*возвращаем nil-вектор*)
    exit;
  end;
  Reset(fp);
  for i:=0 to n-1 do
    read(fp,vec^[i]);(*считываем n составляющих вектора*)
  close(fp);(*закрываем ненужный больше файл*)
  {$I+}
  LoadVector:=vec;(*возвращаем указатель на считанный вектор*)
end;

(*эта процедура записывает вектор в файл с заданным именем*)
procedure WriteVector(n:longint;vec:pvector;f:string);
var
  i:longint;  fp:text;
begin
  if(n<=0) then(*при некорректном размере вектора*)
  begin
  writeln(n,' не может быть размерностью вектора'); exit;
  end;
  {$I-}
  Assign(fp,f);
  if(IOResult<>0) then(*если не удалось открыть файл*)
  begin
    writeln('Не могу открыть файл ',f);  exit;
  end;
```

```
  ReWrite(fp);       (*пытаемся открыть файл*)
  for i:=0 to n-1 do (*записываем n составляющих вектора*)
    write(fp,'       ',vec^[i]:10:3);
  close(fp);      {закрываем ненужный больше файл}
  {$I+}
end;

(*Печать вектора на экран - это не что иное, как за-
пись его в файл, соответствующий экрану*)
procedure PrintVector(n:longint;vec:pvector);
begin
  WriteVector(n,vec,'con');
end;
end.
```

8.14.2. Обработка двумерных числовых массивов (таблиц или матриц или массивов векторов).

Общие сведения о задачах этого класса в 4.9.12.

Мы приведем ряд функций общего характера по конструированию матриц в оперативной памяти при различных исходных данных. Эти функции вы собраны в одном модуле - вы его можете скомпилировать и вызывать эти функции в вашей программе по необходимости.

```
{модуль, содержащий наиболее общие функции для работы
с матрицами}
unit matrixes;
{$R-}
interface

(*Для работы с матрицами нам понадобится два специальных типа :
безразмерный массив указателей на тип вектор и указатель на него
для создания матриц любой размерности.*)

uses vectors;

type
  matrix=array [0..0] of pvector;
  pmatrix=^matrix;

function CreateEmptyMatrix(m,n:longint):pmatrix;
function CreateFilledMatrix(m,n:longint;whatcopy:pmatrix):pmatrix;
function LoadMatrix(m,n:longint;f:string):pmatrix;
procedure WriteMatrix(m,n:longint;mtr:pmatrix;f:string);
procedure PrintMatrix(m,n:longint;mtr:pmatrix);
procedure ReleaseMatrix(m,n:longint;mtr:pmatrix);

implementation
```

```
    (*Эта функция пытается создать пустую матрицу за-
данной размерности, попутно очищая её*)

function CreateEmptyMatrix(m,n:longint):pmatrix;
var
  mtr:pmatrix;
  i,j:longint;
begin
  CreateEmptyMatrix:=nil;
  if(m<=0)or(n<=0) then(*при некорректном размере матрицы*)
  begin
    writeln(m,'x',n,' не может быть размерностью матрицы');
    exit;{возвращаем нулевой указатель}
  end;
  (*пытаемся выделить память под m строк матрицы -
    указателей на вектора вещественных чисел*)
  GetMem(mtr,m*sizeof(pvector));
  for i:=0 to m-1 do(*выделяем память под каждую строку *)
    GetMem(mtr^[i],n*sizeof(real));
  for i:=0 to m-1 do
    for j:=0 to n-1 do
      mtr^[i]^[j]:=0;   (*обнуляем mxn составляющих матрицы*)
  CreateEmptyMatrix:=mtr;(*и возвращаем указатель на неё*)
end;

(*Эта функция пытается создать пустую матрицу заданной размер-
ности, а затем заполнить её данными из другой матрицы*)

function CreateFilledMatrix(m,n:longint;whatcopy:pmatrix):pmatrix;
var
  mtr:pmatrix;
  i,j:longint;
begin
  mtr:=CreateEmptyMatrix(m,n);
  CreateFilledMatrix:=mtr;
  if(mtr=nil)then(*если не удалось создать*)
    exit;(*возвращаем нулевой указатель*)
  for i:=0 to m-1 do
    for j:=0 to n-1 do
      mtr^[i]^[j]:=whatcopy^[i]^[j];(*копируем mxn составляющих )
 CreateFilledMatrix:=mtr;          (*и возвращаем указатель  *)
end;


(*эта функция пытается загрузить матрицу, лежащую в файле в виде
m11 m12... m1n m21... mmn, где mij - её составляющие*)

function LoadMatrix(m,n:longint;f:string):pmatrix;
    (* f - имя файла с матрицей *)
var
  mtr:pmatrix;
```

```
  i,j:longint;
  fp:text;
begin
  mtr:=CreateEmptyMatrix(m,n);
  LoadMatrix:=mtr;
  if(mtr=nil)then(*если не удалось создать*)
    exit;(*возвращаем нулевой указатель*)
  {$I-}
  Assign(fp,f);
  Reset(fp);(*пытаемся открыть файл*)
  if IOResult<>0 then{если не удалось открыть файл}
  begin
    writeln('Не могу открыть файл ',f);
    ReleaseMatrix(m,n,mtr);
    LoadMatrix:=Ptr(0,0);
    exit;(*возвращаем nil-указатель*)
  end;
  for i:=0 to m-1 do
    for j:=0 to n-1 do
    read(fp,mtr^[i]^[j]);(*считываем mxn составляющих матрицы*)
  close(fp);(*закрываем ненужный больше файл*)
  {$I+}
  LoadMatrix:=mtr;(*возвращаем указатель на считанную матрицу*)
end;


{эта функция записывает матрицу в файл с заданным именем}
procedure WriteMatrix(m,n:longint;mtr:pmatrix;f:string);
var
  i,j:longint;
  fp:text;
begin
 if(m<=0)or(n<=0) then(*при некорректном размере матрицы*)
  begin
    writeln(m,'x',n,' не может быть размерностью матрицы');
    exit;{возвращаем нулевой указатель}
  end;
  {$I-}
  Assign(fp,f);  ReWrite(fp);(*пытаемся открыть файл*)
  if IOResult<>0 then(*если не удалось открыть файл*)
  begin
    writeln('Не могу открыть файл ',f); exit;
  end;
  for i:=0 to m-1 do
  begin
    for j:=0 to n-1 do
write(fp,#$9,mtr^[i]^[j]:5:3);(*записываем mxn составляющих*)
    writeln(fp);  (*добавляем в конце каждой строки*)
  end;
  close(fp);      (*закрываем ненужный больше файл*)
  {$I+}
end;
```

```
    (*Вывод матрицы на экран - это не что иное, как
запись её  в файл, соответствующий экрану*)

procedure PrintMatrix(m,n:longint;mtr:pmatrix);
begin
  WriteMatrix(m,n,mtr,'con');
end;
```

(*Для освобождения памяти из-под матрицы, представляющей собой массив указателей, мы должны вначале освободить память из-под каждого указателя на тип vector, и лишь затем - из-под указателя на pvector*)

```
procedure ReleaseMatrix(m,n:longint;mtr:pmatrix);
var
  i,j:longint;
begin
  for i:=0 to m-1 do
    FreeMem(mtr^[i],n*sizeof(real));
  FreeMem(mtr,m*sizeof(pvector));
end;

end.
```

Теперь мы приведем серию примеров прикладных программ, использующих вышеприведенные подпрограммы создания и разрушения матриц в оперативной памяти.

1.Программа сложения двух матриц

Необходимые сведения в 4.9.12.1.

```
uses matrixes;
var
  i,j,m,n:longint;
  Code:integer;
  mtr1,mtr2,mtr3:pmatrix;

begin
(*при недостаточном количестве аргументов в командной
строке*)
  if(ParamCount<5) then
  begin
    writeln('Используйте   :   ',ParamStr(0),'   m   n
file1.txt file2.txt fileres.txt,'#$D#$A#$8+'где mxn -
размерность матриц, лежащих в указанных файлах');
    halt(0);(*выход из программы при недостатке параметров*)
  end;
  Val(ParamStr(1),m,Code);
  Val(ParamStr(2),n,Code);(*определяем размерность матриц*)
  if(m<=0)or(n<=0)then(*при некорректном размере матрицы*)
  begin
```

```
    writeln(m,'x',n,' не может быть размерностью матрицы');
    exit;(*выходим в ОС*)
  end;
mtr1:=LoadMatrix(m,n,ParamStr(3));(*загружаем первую матрицу*)
if(pointer(0)=mtr1)then(*если не удалось загрузить, выходим*)
    halt(word(-1));
  mtr2:=LoadMatrix(m,n,ParamStr(4));(*загружаем вторую матрицу*)
  if(longint(mtr2)=0)then(*если не удалось загрузить*)
  begin   (*освобождаем память из под первой матрицы*)
    ReleaseMatrix(m,n,mtr1);   exit;      (*и выходим*)
  end;
  mtr3:=CreateEmptyMatrix(m,n);
  if(not LongBool(mtr3))(*если не хватило памяти*)then
  begin  (*освобождаем память из под первой матрицы*)
    ReleaseMatrix(m,n,mtr1);
 (*освобождаем память из под второй матрицы*)
    ReleaseMatrix(m,n,mtr2);   exit;   (*выходим*)
  end;
  writeln('Первая матрица:');  PrintMatrix(m,n,mtr1);
  writeln('Вторая матрица:');  PrintMatrix(m,n,mtr2);
  writeln('Сумма матриц:');
  for i:=0 to m-1 do
    for j:=0 to n-1 do  (*поэлементно суммируем матрицы*)
      mtr3^[i]^[j]:=mtr1^[i]^[j]+mtr2^[i]^[j];
  PrintMatrix(m,n,mtr3);
  (*записываем матрицу-результат в соответствующий файл*)
  WriteMatrix(m,n,mtr3,ParamStr(5));
  ReleaseMatrix(m,n,mtr1);(*освобождаем память из под матриц*)
  ReleaseMatrix(m,n,mtr2);
  ReleaseMatrix(m,n,mtr3);
end.
```

(*абсолютно аналогичной будет программа вычитания - меняется только знак*)

2.Программа умножения двух матриц
Необходимые сведения в 4.9.12.2.

```
uses matrixes;
var
  m1,n1,m2,n2,i,j,k:longint;
  Code:integer;
  mtr1,mtr2,mtr3:pmatrix;

begin
  (*при недостаточном количестве аргументов в командной строке*)
  if(ParamCount<7) then
  begin
    writeln('Используйте   :   ',ParamStr(0),'   m1   n1
file1.txt m2 n2 file2.txt fileres.txt,'+
    #$D#$A#$9'где mixni -
```

```
    размерность матриц, лежащих в указанных файлах');
    exit;(*выход из программы при недостатке параметров*)
  end;
  (*определяем размерность первой матрицы*)
  Val(ParamStr(1),m1,Code);
  (*определяем размерность второй матрицы*)
  Val(ParamStr(2),n1,Code);
  Val(ParamStr(4),m2,Code);
  Val(ParamStr(5),n2,Code);
  if(m1<=0)or(n1<=0)or(m2<=0)or(n2<=0)then
  (*при некорректном размере матрицы*)
  begin
    writeln('Некорректная размерность матрицы');
    exit;(*выходим в ОС*)
  end;
  if(n1 <> m2)then
  begin
    writeln('Условие умножения матриц - равенство количества
строк второй матрицы количеству столбцов первой - не выполняет-
ся');
    halt(0);
  end;
  (*загружаем первую матрицу*)
  mtr1:=LoadMatrix(m1,n1,ParamStr(3));
  if(mtr1=nil)(*если не удалось загрузить, выходим*)then
  halt(0);
  (*загружаем вторую матрицу*)
  mtr2:=LoadMatrix(m2,n2,ParamStr(6));
  if(longint(mtr2)=0)(*если не удалось загрузить*)then
  begin
    (*освобождаем память из под первой матрицы*)
    ReleaseMatrix(m1,n1,mtr1);
    exit;                        (*и выходим*)
  end;
  mtr3:=CreateEmptyMatrix(m1,n2);
  if not longbool(mtr3) then(*если не хватило памяти*)
  begin
    ReleaseMatrix(m1,n1,mtr1);(*освобождаем память *)
    ReleaseMatrix(m2,n2,mtr2);  exit;  (*выходим*)
  end;
  writeln('Первая матрица:');
  PrintMatrix(m1,n1,mtr1);
  writeln('Вторая матрица:');
  PrintMatrix(m2,n2,mtr2);
  for j:=0 to n2-1 do
    for i:=0 to m1-1 do
      for k:=0 to n1-1 do
        mtr3^[i]^[j]:=mtr3^[i]^[j]+mtr1^[i]^[k]*mtr2^[k]^[j];
  writeln('Произведение матриц:');
  PrintMatrix(m1,n2,mtr3);
  (*записываем матрицу-результат в соответствующий файл*)
  WriteMatrix(m1,n2,mtr3,ParamStr(7));
```

```
  ReleaseMatrix(m1,n1,mtr1);(*освобождаем память *)
  ReleaseMatrix(m2,n2,mtr2);
  ReleaseMatrix(m1,n2,mtr3);
end.
```

3.Операции над одной матрицей, традиционные для массивов: поиск минимума и максимума и их координат, а также абсолютного минимума (максимума).

```
uses matrixes;
var
  min,max,amin,amax:real;
  i,j,m,n,xmin,xmax,xamin,xamax,ymin,ymax,yamin,yamax:longint;
  mtr:pmatrix;
begin
  (*Для разнообразия введём элементы матрицы с клавиатуры*)
  readln(m,n);(*определяем размерность матрицы*)
  if(m<=0)or(n<=0)then(*при некорректном размере матрицы*)
  begin
    writeln(m,'x',n,' не может быть размерностью матрицы');
    exit;(*выходим в ОС*)
  end;
  mtr:=LoadMatrix(m,n,'con'); (*вводим матрицу с консоли*)
  if(not longbool(mtr))(*если не удалось загрузить, выходим*)
  then   halt(1);
  writeln(#$D#$A'Введенная матрица:');
  PrintMatrix(m,n,mtr);
```

(*устанавливаем значения минимумов и максимумов (обычных и абсолютных [т.е. по модулю]) в значение первого элемента вектора векторов - матрицы действительных чисел, а их соответствующие координаты в его номер, т.е. 0*)

```
  min:=mtr^[0]^[0];  max:=mtr^[0]^[0];
  amin:=abs(mtr^[0]^[0]); amax:=abs(mtr^[0]^[0]);
  xmin:=0;  xmax:=0;  xamin:=0;  xamax:=0;
  ymin:=0;  ymax:=0;  yamin:=0;  yamax:=0;
  for i:=0 to m-1 do(*в циклах по элементам матрицы*)
    for j:=0 to n-1 do
    begin
      if(mtr^[i]^[j]<min)then
      begin
        min:=mtr^[i]^[j]; xmin:=j; ymin:=i;
      end;
      if(mtr^[i]^[j]>max) then
      begin
        max:=mtr^[i]^[j];  xmax:=j;  ymax:=i;
      end;
      if(abs(mtr^[i]^[j])<amin) then
      begin
        amin:=abs(mtr^[i]^[j]); xamin:=j; yamin:=i;
```

```
      end;
      if(abs(mtr^[i]^[j])>amax) then
      begin
        amax:=abs(mtr^[i]^[j]); xamax:=j; yamax:=i;
      end;
    end;
  writeln(#$D#$A'min(mtr)=mtr^[',ymin,']^[',xmin,']=',min:5:3,#$D#$A+
'max(mtr)=mtr^[',ymax,']^[',xmax,']=',max:5:3,#$D#$A+'min(|mtr|)=|mtr^[',
yamin,']^[',xamin,']|=',amin:5:3,#$D#$A+'max(|mtr|)=|mtr^[',yamax,']^[',xa
max,']|=',amax:5:3);
ReleaseMatrix(m,n,mtr);(*освобождаем память из-под матрицы*)
halt(word(-1));  (*вернём -1 как код завершения программы*)
end.
```

4.Программа решения системы линейных алгебраических методом Гаусса

Изложение алгоритма в 4.9.12.3.

```
uses matrixes,vectors;
var
  i,j,k,num,m,n:longint;
  Code:integer; mtr:pmatrix; res:pvector; temp:real;
begin
  (*при недостаточном количестве аргументов в командной строке*)
  if boolean(ParamCount-4) then
  begin
writeln('Используйте : ',ParamStr(0),' m n file.txt fileres.txt,'+
#$D#$A#$8+'где mxn - размерность матриц,
          лежащих в указанных файлах');
halt(0);(*выход из программы при недостатке параметров*)
  end;
  Val(ParamStr(1),m,Code);
 Val(ParamStr(2),n,Code); (*определяем размерность матриц*)
 if(m<=0)or(n<=0)then  (*при некорректном размере матрицы*)
  begin
  writeln(m,'x',n,' не может быть размерностью матрицы');exit;
  end;
  if(n-m)<>1then
  begin
    writeln('Количество неизвестных отличается от
            количества уравнений!');
    exit;
  end;
  mtr:=LoadMatrix(m,n,ParamStr(3));(*загружаем матрицу*)
  if(pointer(0)=mtr)then(*если не удалось загрузить, выходим*)
    halt(word(-1));
(*Результатом решения системы уравнений является вектор неиз-
вестных*)
  res:=CreateEmptyVector(m);
  if(not longbool(res))(*если не хватило памяти*)then
```

```
  begin
    ReleaseMatrix(m,n,mtr); exit;
  end;
  writeln('Исходная матрица СЛАУ:');
  PrintMatrix(m,n,mtr);
  writeln(#$D#$A'Вектор решений:');
  (*Выбор главного элемента*)
  for i:=0 to m-1 do
  begin
    temp:=mtr^[0]^[i];  num:=i;
    for j:=i to m-1 do
     if(abs(mtr^[j]^[i])>temp)then
     begin
       temp:=abs(mtr^[j]^[i]); num:=j;
     end;
    if(num<>i)then
     (*Обмен num-той и i-той строк местами*)
     for k:=0 to n-1 do
     begin
temp:=mtr^[num]^[k];mtr^[num]^[k]:=mtr^[i]^[k];mtr^[i]^[k]:=temp;
     end;
  end;
  for i:=0 to m-1 do
    if(mtr^[i]^[i]=0) then
    begin
writeln('Решение системы при вырожденной матрице невозможно');
(*освобождаем память из под матрицы и вектора*)
ReleaseMatrix(m,n,mtr);FreeMem(res,m*sizeof(real));
halt(0);
    end;
  (*Прямой ход Гаусса*)
  for i:=0 to m-1 do
  begin
    temp:=mtr^[i]^[i];
    for j:=0 to n-1 do
     mtr^[i]^[j]:=mtr^[i]^[j]/temp;
    for k:=i+1 to m-1 do
    begin
     temp:=mtr^[k]^[i];
     for j:=0 to n-1 do
       mtr^[k]^[j]:=mtr^[k]^[j]-mtr^[i]^[j]*temp;
    end;
  end;
  (*Обратный ход Гаусса*)
  for i:=m-2 downto 0 do
  begin
    temp:=0;
    for j:=i+1 to m-1 do
     temp:=temp+mtr^[i]^[j]*mtr^[j]^[n-1];
    mtr^[i]^[n-1]:=mtr^[i]^[n-1]-temp;
  end;
  (*Переписываем из последнего столбца вектор-результат*)
```

```
  for i:=0 to m-1 do
    res^[i]:=mtr^[i]^[n-1];
  PrintVector(m,res);
  (*записываем вектор-результат в соответствующий файл*)
  WriteVector(m,res,ParamStr(4));
  (*освобождаем память из под матрицы и вектора*)
  ReleaseMatrix(m,n,mtr);FreeMem(res,m*sizeof(real));
end.
```

5.Программа нахождения детерминанта матрицы с помощью метода Гаусса.

Алгоритм решения в 4.9.12.4.

```
uses matrixes;
var
  temp,sw,c,det,max:real;
  i,j,k,how,num,m,n:longint;
  mtr:pmatrix;
  Code:integer;
begin
(*при недостаточном количестве аргументов в командной строке*)
  if(ParamCount<3)then
  begin
writeln('Используйте : ',ParamStr(0),' m n file.txt,'#$D#$A+#$9'где mxn -
размерность матрицы, лежащей в указанном файле');
exit;(*выход из программы при недостатке параметров*)
  end;
  (*определяем размерность матриц*)
  Val(ParamStr(1),m,Code);  Val(ParamStr(2),n,Code);
  if(m<=0)or(n<=0)or(m<>n)then(*при некорректном размере *)
  begin
writeln(m,'x',n,' не может быть размерностью матрицы'); exit;
  end;
  mtr:=LoadMatrix(m,n,ParamStr(3));(*загружаем матрицу*)
  if(pointer(0)=mtr)then(*если не удалось загрузить, выходим*)
    halt(0);
  writeln('Исходная матрица:');
  PrintMatrix(m,n,mtr);
  Code:=1;(*Для обхода range check error*);
  case m of
    1:det:=mtr^[0]^[0];
    2:det:=mtr^[0]^[0]*mtr^[Code]^[Code]-
    mtr^[0]^[Code]*mtr^[Code]^[0];
    else
 (*количество перестановок строк определяет знак детерминанта*)
      how:=0;
      for i:=0 to m-1 do
      begin
        max:=mtr^[0]^[i];  num:=i;
        for j:=i to m-1 do
         if(abs(mtr^[j]^[i])>max) then
```

```
          begin
            max:=abs(mtr^[j]^[i]); num:=j;
          end;
        if(num<>i)then
        begin
          for k:=0 to m-1 do
          begin
temp:=mtr^[num]^[k];mtr^[num]^[k]:=mtr^[i]^[k];mtr^[i]^[k]:=temp;
          end;
          inc(how);(*наращиваем количество перестановок*)
        end;
      end;
      (*Прямой ход Гаусса*)
      for i:=0 to m-1 do
      begin
        sw:=mtr^[i]^[i];
        for j:=i+1 to m-1 do mtr^[i]^[j]:=mtr^[i]^[j]/sw;
        for k:=i+1 to m-1 do
        begin
          c:=mtr^[k]^[i];
for j:=0 to m-1 do mtr^[k]^[j]:=mtr^[k]^[j]-mtr^[i]^[j]*c;
        end;
      end;
      (*После прямого хода нам остаётся только перемно-
жить элементы главной диагонали*)
      det:=1;
      for i:=0 to m-1 do
        det:=det*mtr^[i]^[i];
      (*и учесть количество проведенных перестановок строк*)
      if(wordbool(how and 1)) then
        det:=-det;
  end;
  writeln(#$D#$A'Детерминант: ',det:5:3);
  (*освобождаем память из под матрицы и вектора*)
  ReleaseMatrix(m,n,mtr);
end.
```

6.Программа ортогонализации матрицы (построения из строк матрицы ортогональной системы векторов)

Алгоритм решения в 4.9.12.5.

```
uses matrixes;
var
  s,l,dd,i,m,n:longint;
  Code:integer;
  mtr,orto:pmatrix;
  md:real;

begin
(*при недостаточном количестве аргументов в командной строке*)
if(ParamCount<4) then
```

```
  begin
    writeln('Используйте : ',ParamStr(0),' m n file.txt
fileres.txt,'+#$D#$A#9'где mxn -  размерность матриц, лежащих в ука-
занных файлах');
    exit;(*выход из программы при недостатке параметров*)
  end;
  Val(ParamStr(1),m,Code);
  Val(ParamStr(2),n,Code);(*определяем размерность матрицы*)
  if(m<=0)or(n<=0)then(*при некорректном размере матрицы*)
  begin
writeln(m,'x',n,' не может быть размерностью матрицы');exit;
  end;
  mtr:=LoadMatrix(m,n,ParamStr(3));  (*загружаем матрицу*)
  if(mtr=nil)then(*если не удалось загрузить, выходим*)
    halt(0);
  orto:=CreateFilledMatrix(m,n,mtr);
  if not longbool(orto)then(*если не хватило памяти*)
  begin  (*освобождаем память из под первой матрицы*)
    ReleaseMatrix(m,n,mtr);  exit;          (*выходим*)
  end;
  writeln('Исходная матрица:');
  PrintMatrix(m,n,mtr);
  writeln('Ортонормированная матрица:');
  (*Вычисляем сумму квадратов элементов первой строки матри-
цы*)
  md:=0;
  for i:=0 to n-1 do
    md:=md+orto^[0]^[i]*orto^[0]^[i];
  md:=sqrt(md);(*находим корень то есть модуль первой строки*)
  (*нормируем первую строку, деля каждый её элемент на модуль*)
  for i:=0 to n-1 do
    orto^[0]^[i]:=orto^[0]^[i]/md;
  (*Для надёжности повторяем процесс ортогонализации 3 раза*)
  for dd:=0 to 2 do
  begin
    for l:=1 to m-1 do
    begin
      for i:=0 to l-1 do
      begin
     (*Вычисляем скалярное произведение l-той строки на i-ую*)
        md:=0;
        for s:=0 to n-1 do
          md:=md+orto^[l]^[s]*orto^[i]^[s];
        (*Ортогонализируем l-тую строку к i-ой*)
        for s:=0 to n-1 do
          orto^[l]^[s]:=orto^[l]^[s]-md*orto^[i]^[s];
      end;
      (*Нормируем l-тую строку*)
      md:=0;
      for s:=0 to n-1 do
        md:=md+orto^[l]^[s]*orto^[l]^[s];
```

```
      md:=sqrt(md);
      for s:=0 to n-1 do
        orto^[l]^[s]:=orto^[l]^[s]/md;
    end;
  end;
  PrintMatrix(m,n,orto);
```

(*После вывода на печать матрицы для очистки совести выводим попарно скалярные произведения всех строк, дабы убедиться, что построенная нами система векторов действительно ортогональна*)

```
  for i:=0 to m-1 do
    for dd:=i+1 to m-1 do
    begin
      md:=0;
      for s:=0 to n-1 do
        md:=md+orto^[i]^[s]*orto^[dd]^[s];
      writeln('orto[',i,']*orto[',dd,']=',md:5:3);
    end;
  (*записываем матрицу-результат в соответствующий файл*)
  WriteMatrix(m,n,orto,ParamStr(4));
  ReleaseMatrix(m,n,mtr);(*освобождаем память из под матриц*)
  ReleaseMatrix(m,n,orto);
end.
```

7.Программа транспонирования матрицы (замены строк столбцами)

```
uses matrixes;
var
  m,n,i,j:longint;  Code:integer;  mtr,tran:pmatrix;

begin
(*при недостаточном количестве аргументов в командной строке*)
if(ParamCount<4) then
begin
  writeln('Используйте : ',ParamStr(0),' m n file.txt fileres.txt,'+#$D#$A#9'где mxn - размерность матриц, лежащих в указанных файлах');
    exit;(*выход из программы при недостатке параметров*)
  end;
  Val(ParamStr(1),m,Code);
  Val(ParamStr(2),n,Code);(*определяем размерность матрицы*)
  if(m<=0)or(n<=0)then(*при некорректном размере матрицы*)
  begin
writeln(m,'x',n,' не может быть размерностью матрицы'); exit;
  end;
  mtr:=LoadMatrix(m,n,ParamStr(3));(*загружаем матрицу*)
 if(mtr=nil)then(*если не удалось загрузить, выходим*)
    halt(0);
  tran:=CreateEmptyMatrix(n,m);
```

```
  if not longbool(tran)then(*если не хватило памяти*)
  begin  (*освобождаем память из под первой матрицы*)
    ReleaseMatrix(m,n,mtr);
    exit;       (*выходим*)
  end;
  writeln('Исходная матрица:');
  PrintMatrix(m,n,mtr);
  writeln('Транспонированная матрица:');
  for i:=0 to m-1 do
    for j:=0 to n-1 do
     tran^[j]^[i]:=mtr^[i]^[j];
  PrintMatrix(n,m,tran);
  (*записываем матрицу-результат в соответствующий файл*)
  WriteMatrix(n,m,tran,ParamStr(4));
  ReleaseMatrix(m,n,mtr);(*освобождаем память из под матриц*)
  ReleaseMatrix(n,m,tran);
end.
```

8.15. Работа с текстовыми строками.

В настоящем разделе мы приведем 2 примера работы со строками - сортировку строк и синтаксическую разборку строки на лексемы - условные неделимые единицы некоторого языка, принятого в контексте текущей задачи. Остальные случаи будут встречаться по текстам различных программ и там по возможности комментироваться.

8.15.1. Сортировка строк.

Пусть в текстовом файле находится массив слов, разделенных друг от друга пробелами или переводами строки - это может быть например неупорядоченный алфавитно список фамилий учащихся вашей группы и необходимо вывести его на экран или в файл в упорядоченном виде. Имя файла будет задано пользователем в командной строке.

Программа может выглядеть так (а может и иначе - вам надо привыкать к тому, что программирование - творческий процесс и создать "рецептурный справочник" по написанию программ невозможно):

```
uses crt,strings;
{$R-}
type
  unChar=array[0..0] of char;
  PunChar=^unChar;
  astring=array[0..0]of PunChar;
  pastring=^astring;
var
```

```
  j,i,wc:word;
  f:file of char;
  wptr:pastring;
const
  controls:set of #$00..#$FF=[#$20,#$D,#$A,#$9,#12,#11];
  temp:char=' ';
  maxlenword:PChar='                                    ';
  (*Для одного самого длинного слова из файла*)

begin (*В командной строке должно быть имя файла*)
  if(ParamCount<>1)then
  begin
writeln('Неполадки с параметрами в командной строке');exit;
  end;
  {$I-}
  Assign(f,ParamStr(1)); Reset(f);
  if boolean(IOResult)then
  begin
writeln('Неудача с открытием файла ',ParamStr(1)); exit;
  end;
  (*Сосчитаем количество слов в файле*)
  wc:=0;
  while (temp in controls) and not eof(f)do
    read(f,temp);
  while not eof(f) do
  begin
while not (temp in controls) and not eof(f)do
     read(f,temp); inc(wc);
while (temp in controls) and not eof(f)do  read(f,temp);
  end;
  close(f);
  Reset(f); (*возвратим указатель чтения - записи в начало*)
  (*Теперь можем попросить память под указатели на слова*)
  GetMem(wptr,wc*sizeof(pastring));
  (*Перепишем файл в память с попутным ее выделением
для каждого слов*)
  for i:=0 to wc-1 do
    GetMem(wptr^[i],sizeof(string));
  temp:=' ';
  while (temp in controls) and not eof(f)do
    read(f,temp);
  for i:=0 to wc-1 do
  begin
    j:=0; wptr^[i]^[j]:=temp; inc(j);
    while not (temp in controls) and not eof(f)do
    begin
      read(f,temp); wptr^[i]^[j]:=temp;inc(j);
    end;
if eof(f) and not (temp in controls) then wptr^[i]^[j]:=#0;
if(temp in controls)then  wptr^[i]^[j-1]:=#0;
while (temp in controls) and not eof(f)do  read(f,temp);
```

```
  end;
  close(f);
(*Только теперь мы добрались до задачи сортировки -
выполним ее  простейшим 'пузырьковым' методом*)
  for i:=0 to wc-1 do
    for j:=i+1 to wc-1 do
     if StrComp(PChar(wptr^[i]),PChar(wptr^[j]))>0then
     begin
       StrCopy(maxlenword,PChar(wptr^[i]));
       StrCopy(PChar(wptr^[i]),PChar(wptr^[j]));
       StrCopy(PChar(wptr^[j]),maxlenword);
     end;
  (*Осталось вывести результат*)
  clrscr;
  writeln('Отсортированные по алфавиту слова');
  for i:=0 to wc-1 do
    writeln(PChar(wptr^[i]));
  (*И освободить арендованную память*)
  for i:=0 to wc-1 do
    FreeMem(wptr^[i],sizeof(string));
  GetMem(wptr,wc*sizeof(pastring));
  {$I+}
end.
```

8.15.2. Синтаксический анализ выражений (формул), заданных символьной строкой.

Эта задача является вспомогательной в программах интерпретации формул, заданных пользователем в виде символьных строк, в частности:

◊ при построении графиков функций, задаваемых пользователем вводом с клавиатуры;

◊ при выполнении расчетов по часто и непредсказуемо изменяющимся формулам - например, при начислении налогов, в программах-калькуляторах и пр.

В общем случае в формуле могут встретиться 2 класса лексем - операции и операнды и дополнительные промежуточные классы, например "конец формулы", "строка", "выражение в скобках". К операциям мы отнесем не только очевидный набор арифметических символов (+,-,^,*,/), но и все функции, которые распознает и выполняет наш интерпретатор (например sin(выражение), cos(выражение) и пр.

Классы можем закодировать например так:

```
const OPERAND=1;OPERATION=2;SKOBKA=3; EOL=4;
```

Эти значения будем использовать для инициализации глобальной переменной token_type.

Допустимые для использования в формулах операции (подклассы класса OPERATION) тоже закодируем, чтобы ускорить операции обработки (это называется переводом во внутренний формат):

```
{Коды операций}
const
MUL=1;LDIV=2;POW=3;PLUS=4;MINUS=5;       {Арифметика}
LABS=6;  ACOS=7;ASIN=8;  ATAN=9;  LCOS=10;  COSH=11;
LEXP=12; LOG=13;
LOG10=14; LSIN=15; SINH=16; LSQRT=17; TAN=18; TANH=19;
```

Все обозначения (имена) операций и их коды сведем в таблицу (массив структур tf[]) по шаблону

```
type
 loperation=record
 on:array [0..9] of char; ov:integer;
 end;

const
 op:array [0..20] of loperation=
(
(on:'ABS';ov:LABS),  (on:'ACOS';ov:ACOS),  (on:'ASIN';ov:ASIN),
(on:'ATAN';ov:ATAN),  (on:'COS';ov:LCOS),  (on:'COSH';ov:COSH),
(on:'EXP';ov:LEXP),  (on:'LN';ov:LOG),  (on:'LOG';ov:LOG10),
(on:'SIN';ov:LSIN),  (on:'SINH';ov:SINH),  (on:'SQRT';ov:LSQRT),
(on:'TAN';ov:TAN),  (on:'TANH';ov:TANH),  (on:'+';ov:PLUS),  (on:'-
';ov:MINUS),  (on:'*';ov:MUL),  (on:'/';ov:LDIV),  (on:'^';ov:POW),
(on:'';ov:0));
```

причем в поля op[i].on занесли имена операций, а в поля op[i].ov значения их числовых кодов.

К операндам (подклассы класса OPERAND) отнесем переменные, именованные константы, непосредственно заданные числа и закодируем:

```
const LSTRING=1; {просто неклассифицированная еще строка}
      NUMBER=2; {числовая строка}
      VARIABLE=3; {переменная}
      CONSTANT=4; {константа}
      EXPRESS =5; {выражение в скобках}
```

Значения кодов подклассов будем заносить в глобальную переменную tok.

Будем предполагать, что допустимые для использования в формулах именованные константы сведены в константный массив структур вида:

```
type cnst=record
 cn:array [0..5] of char;
 cv:real;
end;   шаблон структуры

const
 c:array [0..2] of cnst=(
 (cn:'PI';cv:3.14159265358979323846),
(cn:'E';cv:2.71828182845904523536),
 (cn:'';cv:0)
 );константный массив структур
```

При этом в поля c[i].cn записаны имена констант, а в поля c[i].cv - их числовые коды.

Предварительная обработка формульной строки может осуществляться еще при клавиатурном вводе фильтрацией недопустимых символов, а если нет уверенности, что это выполнено достаточно эффективно, то необходимо в простейшем случае удалить из нее все недопустимые символы - пробелы, символы табуляции, возврата каретки, перевода строки, всевозможные разделители типа запятой или двоеточия, буквы нелатинского алфавита, оставив только предусмотренные к распознаванию знаки.

Кроме того, необходимо перед анализом привести все буквенные символы к верхнему регистру.

Таким образом, первое, что нам понадобится - это инструмент для выделения и классификации лексем (неделимых единиц формульного языка) из текста формулы. Остальные разъяснения алгоритма синтаксического анализа формульной строки мы дадим в комментариях к отдельным строкам приводимого ниже текста программы.*)

```
uses strings,crt;

(*служебные типы*)
type
  preal=^real;
  PPChar=^PChar;(*указатель на массив ASCIIZ-строк*)
```

(*Для хранения кода класса лексемы выделим глобальную переменную token_type, а для кода ее конкретного значения - tok*)

```
var
  token_type,tok:integer;
  token:array [0..9] of char; (*для хранения выделенной лексемы*)

(*Коды классов лексем*)
const OPERAND=1;OPERATION=2;SKOBKA=3;EOL=4;

(*Подклассы операндов*)
const LSTRING=1;  (*просто неклассифицированная еще строка*)
      NUMBER=2;    (*числовая строка*)
      VARIABLE=3;  (*переменная*)
      CONSTANT=4;  (*константа*)
      EXPRESS =5;  (*выражение в скобках*)
  WasError:boolean=false;  (*индикатор ошибки при синтаксическом
разборе позволяет быстро его закончить и начать новый без выхода
из программы, вернувшись к вводу данных*)

type cnst=record
  cn:array [0..5] of char;
  cv:real;
end; (*   шаблон структуры*)

const
  tc:array [0..2] of cnst=(
  (cn:'PI';cv:3.14159265358979323846),
  (cn:'E';cv:2.71828182845904523536),
  (cn:'';cv:0)
 );(*константный массив структур  *)

{Коды операций}
const
MUL=1;LDIV=2;POW=3;PLUS=4;MINUS=5;  {Арифметика}
LABS=6;   ACOS=7;ASIN=8;   ATAN=9;   LCOS=10;   COSH=11;
LEXP=12;LOG=13;  LOG10=14;  LSIN=15;  SINH=16;  LSQRT=17;
TAN=18;  TANH=19;

     (*Все  обозначения  (имена) операций  и  их  коды
сведем  в таблицу (массив структур tf[]) по шаблону*)
type

  loperation=record
    on:array [0..9] of char;
    ov:integer;
  end;

const
  op:array [0..19] of loperation=
(
(on:'ABS';ov:LABS),   (on:'ACOS';ov:ACOS),   (on:'ASIN';ov:ASIN),
(on:'ATAN';ov:ATAN),   (on:'COS';ov:LCOS),   (on:'COSH';ov:COSH),
```

(on:'EXP';ov:LEXP), (on:'LN';ov:LOG), (on:'LOG';ov:LOG10), (on:'SIN';ov:LSIN), (on:'SINH';ov:SINH), (on:'SQRT';ov:LSQRT), (on:'TAN';ov:TAN), (on:'TANH';ov:TANH), (on:'+';ov:PLUS), (on:'-';ov:MINUS), (on:'*';ov:MUL), (on:'/';ov:LDIV), (on:'^';ov:POW), (on:'';ov:0));

OOP:PChar='+-*/^'; (*Перечень арифметических операций*)

```
(*НЕКОТОРЫЕ СЛУЖЕБНЫЕ ПОДПРОГРАММЫ*)
(*Преобразование к верхнему регистру*)
function toupper(c:char):char;
var
  d:byte;
begin
  d:=byte(c);
  if(d>=byte('a'))and(d<=byte('z')) then   dec(d,$20);
  if(d>=byte('a'))and(d<=byte('п')) then   dec(d,$20);
  if(d>=byte('p'))and(d<=byte('я')) then   dec(d,$50);
  if(d=byte('ё'))                   then   d:=byte('Ё');
  toupper:=char(d);
end;

(*Тест на "низость" символа, в т.ч. для русских букв*)
function islower(key:char):boolean;
begin
  islower:=false;
  if (key='ё') or
     ( (byte(key)>=byte('a')) and (byte(key)<=byte('z')) ) or
     ( (byte(key)>=byte('a')) and (byte(key)<=byte('п')) ) or
     ( (byte(key)>=byte('p')) and (byte(key)<=byte('я')) ) then
     islower:=true;
end;


(*Ф-ция приведения латинских букв к верхнему регистру*)
procedure touppercase(f:PChar);
var
  j:integer;
begin
  for j:=0 to StrLen(f) do
    if islower(f[j]) then
      f[j]:=toupper(f[j]);
end;

     (*Тест на алфавитно-цифровые символы*)
function isalnum(key:char):boolean;
begin
  isalnum:=false;
 if ( (byte(key)>=byte('0')) and (byte(key)<=byte('9')) ) or
      ( (byte(key)>=byte('A')) and (byte(key)<=byte('Z')) ) or
      ( (byte(key)>=byte('a')) and (byte(key)<=byte('z')) ) or
```

```
     ( (byte(key)>=byte('A')) and (byte(key)<=byte('п')) ) or
     ( (byte(key)>=byte('р')) and (byte(key)<=byte('ё')) ) then
     isalnum:=true;
end;


(*Функция очистки формульной строки от излишков и неточностей*)
procedure clearformula(f:PChar);
var
  temp:PChar;
  i:integer;
begin
  temp:=@f[0];
  while boolean(temp^) do
    if isalnum(temp^) or longbool(StrScan(OOP,temp^))
     or(temp^='.')or(temp^='(')or(temp^=')') then
     inc(temp)(*пример адресной арифметики*)
    else
     (*Вначале удалим все недопустимые в формуле символы*)
     move(temp[1],temp[0],StrLen(@temp[1])+1);

(*Затем несколько раз на наличие сочетаний типа '+*' или '-*'*)
  temp:=@f[1];
  while boolean(temp^) do
  begin
    if((temp^='*')or(temp^='/')or(temp^='^'))and
     ((temp-1)^='+')or((temp-1)^='-')then
    begin
     move(temp[1],temp[0],StrLen(@temp[1])+1);
     temp:=@f[1];{перезапуск с начала строки}
    end;
    inc(temp);
  end;
end;


(*Ф-ция поиска по таблицам для определения типа строковых лексем; параметры ф-ции - искомая строка и адрес, по которому положить значение обнаруженной именованной константы*)

procedure look_up(s:PChar;cv:preal);
var
  i:integer;
begin
  (*Просмотрим таблицу операций*)
  i:=0;
  while wordbool(StrLen(op[i].on)) do
  begin
  (*Если нашли совпадение с допустимым именем операции
    возвращаем конкретный тип операции*)
    if not wordbool(StrComp(op[i].on,s)) then
```

```
    begin
      token_type:=OPERATION;
      tok:=op[i].ov;
      exit;
    end;
    inc(i);
  end;
  i:=0;
  (*Поиск в таблице констант*)
  while wordbool(StrLen(tc[i].cn)) do
  begin
(*Если нашли совпадение с  допустимым  именем  кон-
станты возвращаем индекс константы в  массиве  струк-
тур  и  ее  значение записываем по заданному адресу*)
    if not wordbool(StrComp(tc[i].cn,s)) then
    begin
token_type:=OPERAND;tok:=CONSTANT; cv^:=tc[i].cv; exit;
    end;
    inc(i);
  end;

  (*Если ничего не нашли*)
  token_type:=0;tok:=-1;
end;

(*Ф-ция для вывода сообщения об ошибке и возврата к вводу *)
procedure serror(error:integer);
const
  e:array [0..6] of string=(
'Синтаксическая ошибка',
'Непарные круглые скобки',
'Где-то не выражение',
'Где-то не переменная',
'Есть нераспознанная операция',
'Не распознанное имя константы',
'Получается деление на нуль');
begin
  gotoxy(1,5);
  write(e[error],'-any key to continue');
  readkey;
  WasError:=true;
end;

(*Тест на цифровые символы*)
function isdigit(key:char):boolean;
begin
  isdigit:=false;
  if  (byte(key)>=byte('0')) and (byte(key)<=byte('9'))then
    isdigit:=true;
end;
```

(*Тест на алфавитные символы - операция пересечения 2-х множеств*)

```
function isalpha(key:char):boolean;
begin
  isalpha:=isalnum(key) and not isdigit(key);
end;
```

(*Функция выделения и классификации очередной лексемы, возвращает класс лексемы. Ее параметры - указатель на указатель текущего текста формулы и адрес для значения операнда-константы - он транзитом передается в look_up(). Первый параметр мы вынуждены получать 'по ссылке', чтобы значение адреса текущей лексемы изменялось в вызывающей программе, а не в нашей подпрограмме*)

```
function get_token(cf:PPChar;cv:preal):integer;
var
  opr,temp:PChar;(*Временный указатель на лексему*)
begin
  token_type:=0;  tok:=0;
  (*Если конец формулы*)
  if(cf^^=#0)then
  begin
  token[0]:=#0; token_type:=EOL; get_token:=token_type; exit;
  end;

  (*Если это открытая круглая скобка*)
  if(cf^^='(')then
  begin
    temp:=token;
    temp^:=cf^^;   (*перепишем его в token*)
    inc(cf^);     (* переход на следующую позицию*)
    inc(temp);
    temp^:=#0;       (*token закроем нулем*)
    tok:=EXPRESS;
    token_type:=OPERAND;
    get_token:=token_type;
    exit;
  end;

(*Если это закрытая круглая скобка
  if(cf^^=')')then
  begin
    temp:=token;
    temp^:=cf^^;   (*перепишем его в token*)
    inc(cf^);     (* переход на следующую позицию*)
    inc(temp);
    temp^:=#0;       (*token закроем нулем*)
    tok:=EXPRESS;
    token_type:=SKOBKA;
    get_token:=token_type;
```

```
    exit;
*)

(*Если это символ арифметической операции*)
  opr:=StrScan(OOP,cf^^);
  if(opr<>nil)then
  begin
    temp:=token;
    temp^:=cf^^;   (*перепишем его в token*)
    inc(cf^);     (* переход на следующую позицию*)
    inc(temp);
    temp^:=#0;       (*token закроем нулем*)
    case opr^ of
    '+': tok:=PLUS;
    '-': tok:=MINUS;
    '*': tok:=MUL;
    '/': tok:=LDIV;
    '^': tok:=POW;
    end;
    (*сообщим что класс лексемы - операция*)
    token_type:=OPERATION;
    get_token:=token_type;
    exit;
  end;

(*Если встретили цифру*)
  if isdigit(cf^^)then
  begin
    (*то запишем всю числовую подстроку в token*)
    temp:=token;
    while isdigit(cf^^) or (cf^^='.')do
    begin
    temp^:=cf^^;  inc(temp);  inc(cf^);
    end;
    temp^:=#0;  tok:=NUMBER; token_type:=OPERAND;
    get_token:=token_type;
    exit;
  end;

(*Если встретили букву*)
  if isalpha(cf^^)then
  begin
(*то это переменная или операция-функция или именованная константа, пока все буквы-цифры перепишем и временно зарегистрируем просто строкой*)
    temp:=token;
    while isalnum(cf^^)do
    begin
    temp^:=cf^^; inc(temp);   inc(cf^);
    end;
    token_type:=LSTRING;
  end;
```

```
  temp^:=#0;

  (*Не отходя далеко проанализируем полученную строку*)
  (*Если это 1-буквенное имя независимой переменной*)
  if(token_type=LSTRING )and not wordbool(StrComp(token,'T'))then
  begin
tok:=VARIABLE;token_type:=OPERAND;get_token:=token_type;exit;
  end;

(*В противном случае поищем среди унарных операций и констант*)
  look_up(token,cv);
  if(tok<0)then  serror(4);
end;

(* Эта подпрограмма может использоваться как вспомогательная,
например, в интерпретаторе для вычисления значений формулы при
различных значениях независимой переменной. В следующих раз-
делах это будет продемонстрировано, а пока мы просто протестиру-
ем приведенные ф-ции в программе, выводящей на экран отдельные
лексемы заданной в командной строке формулы, их классы и под-
классы.*)

{$I gettok.inc}
var
  formula:array[0..80]of char;  (*Для сохранения текста формулы*)
  cv:real; t:PChar;
begin
  if(ParamCount<>1)then
  begin
  write('Неполадки с параметрами в командной строке');  exit;
  end;
  StrPCopy(formula,ParamStr(1));  (*Сохраняем текст формулы*)
  touppercase(formula);      (*Приводим к верхнему регистру*)
  clearformula(formula);     (*Убираем недопустимые символы*)
  t:=PChar(@formula);
  clrscr;
  get_token(@t,@cv);
  while(token_type<>EOL)and(not WasError)do
  begin
    writeln('Лексема     ',token,',     код     класса
',token_type,', код подкласса ',tok);
    get_token(@t,@cv);
  end;
end.
```

8.15.3. Рекурсивный интерпретатор формул, заданных на стадии выполнения программы в виде символьной строки.

Задача интерпретирующей подпрограммы - вернуть вычисленное значение при заданном значении аргументов. Для упрощения и без того достаточно громоздкого алгоритма мы будем предполагать, что формула представляет собой функцию f(t) одного аргумента с фиксированным именем 't' и в нее могут входить в качестве операндов числа, именованные константы (например 'pi', 'e' и пр.), аргумент 't' и набор различных математических операций и функций.

В качестве вспомогательного средства функция - интерпретатор будет использовать подпрограммы синтаксического анализатора строковых выражений, который будет вызываться для получения каждой следующей лексемы - неделимой единицы выбранного "формульного" языка, который у нас по возможности не будет отличаться от принятого в математике. Неизбежное отличие будет конечно состоять в том, что например произведение dcos(t) придется записывать в виде d*cos(t) с явным, а не подразумеваемым обозначением операции умножения. Подпрограммы реализации синтаксического анализатора поместим в файл gettok.inc и включим его в нашу программу директивой {$Ifilename.ext}. Собственно интерпретацию и вычисление значения формулы при заданном значении аргумента t будем осуществлять по следующему алгоритму: вначале просмотрим текст формулы, предварительно "очищенный" специальной подпрограммой от пробелов и явных ошибок в расстановке операций - например от знаков умножений, делений, возведения в степень сразу после знаков других операций; в процессе просмотра заполним массив структур, полями которых являются коды операций или операндов и (для операндовой структуры) значения операндов. При получении операнда класса EXPRESS (выражение в скобках) мы просто перепишем все внутрискобочное выражение в приватный массив подпрограммы и рекурсивно вызовем ее для вычисления внутрискобочного выражения - еще один пример использования рекурсии для упрощения метода решения задачи.

После построения структуры формулы в виде последовательности операций и операндов просто приступим к выполнению операций в порядке их приоритетов - сначала все унарные операции, включая математические функции от скобочных выражений, затем возведения в степень, потом умножения и деления и в последнюю очередь сложения и вычитания. После выполнения каждой операции количество элементов в массиве формульных структур будем сокращать после записи результата

операции в поле соответствующего операнда и перемещения влево области занимаемой структурами массива памяти.

После выполнения последней операции результат окажется в 0-м элементе массива структур - оттуда его и возвратит наша функция интерпретации выражений.

```
{$I gettok.inc} (*Выделение лексем*)

var
 formula:array[0..80]of char; (*Для сохранения текста формулы*)
  t:real;              (*Для значения аргумента*)
```

(* Нам понадобятся некоторые вспомогательные функции, в частности функция выполнения заданных двухместных математических операций - ее параметры при вызове - это код требуемой двухместной операции и значения левого и правого операндов.*)

```
function oper(co:integer;lo,ro:real):real;
begin
  case(co)of
    PLUS :oper:=lo+ro;
    MINUS:oper:=lo-ro;
    MUL  :oper:=lo*ro;
    LDIV  :
       if(ro=0) then   serror(6)
       else         oper:=lo/ro;
    POW  :
     begin
       if lo=0 then    oper:=0
       else         oper:=exp(ro*ln(lo));
     end;
    else           serror(0);
  end;
end;
```

(*Эта функция осуществляет унарные операции, к классу которых мы отнесли помимо унарных "плюс" и "минус" все математически функции, полагая находящееся в скобках после имени функции выражение единственным их операндом *)

```
function unary(co:integer;ro:real):real;
begin
  case(co)of
    PLUS :unary:=ro;
    MINUS:unary:=-ro;
    LSIN :unary:=sin(ro);
    LCOS :unary:=cos(ro);
    TAN  :unary:=sin(ro)/cos(ro);
    LSQRT:unary:=sqrt(ro);
    else  serror(0);
```

```
  end;
end;
```

(* Формула состоит из операндов и операций. Типы операндов - число, именованная константа, переменная, выражение в скобках.*)

```
type __formula=record   (*Для структуры формулы*)
  dtok,                (*Код типа операнда*)
  co:word;               (*Код операции*)
  z:real;                 (*значение операнда*)
end;
```

(* Это уже основная подпрограмма вычисления значений по тексту формулы, она возвращает вычисленное значение, а ее параметры - указатель на текст формулы и значение независимой переменной*)

```
var
result:real; (*для промежуточных результатов вычислений*)
(*для текущего фрагмента формулы*)
privat_form:array[0..80]of char;
function get_exp(f:PChar;t:real):real;
var
    tt,flag,Code,
    sc,            (*Счетчик скобок*)
    i,j,          (*рабочие переменные для циклов*)
    co:integer;      (*для кода операции*)
    cv:real;  (*для значения именованной константы*)
  tmp:PChar;       (*Исходный адрес формулы*)
  cnt:word;         (*Количество элементов в формуле*)
  frm:array[0..31]of __formula;
begin
  for i:=0 to 31 do
    FillChar(frm[i],sizeof(frm[i]),0);(*Обнулим*)
  tmp:=f; i:=0;  cnt:=0;  get_token(@tmp,@cv);
  (*Просмотрим текст формулы и заполним ее структуру*)
  while(token_type<>EOL)do
  begin
    (*Если получили операнд - число, константу или переменную*)
    if(token_type=OPERAND)then
    begin
      frm[i].dtok:=tok;
      if(tok=NUMBER)then    Val(token,frm[i].z,Code);
      if(tok=CONSTANT)then    frm[i].z:=cv;
      if(tok=VARIABLE)then    frm[i].z:=t;
      if(tok=EXPRESS)then
      begin
        tt:=token_type;(*Сохраним тип лексемы*)
        sc:=1;
```

(*Перепишем все после нее до парной ей закрытой в массив privat_form создавая таким образом другую, частную формулу, как

фрагмент общей - для нее мы можем использовать такой же механизм исследования*)

```
        j:=0;
        while wordbool(sc)and(j<78)do
        begin
          if(tmp^='(')then inc(sc);
          if(tmp^=')')then dec(sc);
          privat_form[j]:=tmp^; inc(j);inc(tmp);
        end;
        privat_form[j-1]:=#0;    (*Закроем нулем*)
if wordbool(sc)then  serror(1); (*Если не нашли парную скобку*)
        result:=get_exp(privat_form,t);
        token_type:=tt;     (*Восстановим тип лексемы*)
        frm[i].dtok:=NUMBER; frm[i].z:=result;
      end;(*if EXPRESS   *)
    end;  (*if OPERAND*)

    (*Если получили операцию*)
if(token_type=OPERATION)then  frm[i].co:=tok; inc(i);
get_token(@tmp,@cv);
end; (*while*)

cnt:=i;    (*Количество заполненных элементов*)
```

(*Теперь можем обрабатывать. Вначале необходимо выполнить все унарные операции - у них самый высокий приоритет. Признаком унарной операции является либо если она первая в структуре формулы, либо предыдущий в формуле элемент - операция, а последующий - операнд*)

```
  flag:=1;
  while wordbool(flag)do
  begin
    flag:=0; i:=0;
    while(i<cnt)do
    begin
if((i=0)and(frm[i].co<>0)and(frm[i+1].dtok<>0))or((i<>0)and
(frm[i-1].co<>0)and(frm[i].co<>0)and(frm[i+1].dtok<>0))then
      begin
        frm[i+1].z:=unary(frm[i].co,frm[i+1].z);
        (*Сомкнем ряды на выполненной операции*)
        move(frm[i+1],frm[i],(cnt-i-1)*sizeof(frm[i]));
        dec(cnt); inc(flag);
      end;
      inc(i);
    end;
  end;
  (*Теперь необходимо выполнить все возведения в степень *)
  i:=1;
  while(i<cnt)do
  begin
```

```
if(frm[i].co=POW)and(frm[i-1].dtok<>0)and(frm[i+1].dtok<>0)then
    begin
      frm[i-1].z:=oper(POW,frm[i-1].z,frm[i+1].z);
      (*Сомкнем ряды на выполненной операции*)
move(frm[i+2],frm[i],(cnt-i-2)*sizeof(frm[i]));dec(i);dec(cnt,2);
    end;
    inc(i);
  end;
  (*Теперь  выполнить все деления и умножения *)
  i:=1;
  while(i<cnt)do
  begin
    if((frm[i].co=MUL)or(frm[i].co=LDIV))and(frm[i-1].dtok<>0)
       and(frm[i+1].dtok<>0)then
    begin
     frm[i-1].z:=oper(frm[i].co,frm[i-1].z,frm[i+1].z);
      (*Сомкнем ряды на выполненной операции*)
move(frm[i+2],frm[i],(cnt-i-2)*sizeof(frm[i]));dec(i);dec(cnt,2);
    end;
    inc(i);
  end;
  (*Теперь  выполнить все сложения и вычитания *)
  i:=1;
  while(i<cnt)do
  begin
    if((frm[i].co=PLUS)or(frm[i].co=MINUS))and(frm[i-
1].dtok<>0) and(frm[i+1].dtok<>0)then
    begin
     frm[i-1].z:=oper(frm[i].co,frm[i-1].z,frm[i+1].z);
      (*Сомкнем ряды на выполненной операции*)
move(frm[i+2],frm[i],(cnt-i-2)*sizeof(frm[i]));dec(i);dec(cnt,2);
    end;
    inc(i);
  end;
  (*Теперь в 0-м элементе лежит результат*)
  get_exp:=frm[0].z;
end;

{$I interpr.inc}
```

(* Пусть текст формулы и значение аргумента, при котором она должна быть вычислена, задается в командной строке *)

```
var
  Code:integer;
  r:real;
begin
  if(ParamCount<>2)then
  begin
  writeln('Неполадки с параметрами в командной строке');exit;
  end;
```

```
  StrPCopy(formula,ParamStr(1));  (*Сохраняем текст формулы*)
  Val(ParamStr(2),t,Code);
  touppercase(formula);clearformula(formula);
  r:=get_exp(formula,t);
  clrscr;
  writeln('Вычисленное значение при t=',t:5:3,' равно ',r:5:3);
end.
```

8.16. Работа в графическом режиме с библиотекой Borland Graphics Interface (BGI) при построении графиков функций.

Предположим, что перед нами поставлена задача составить программу - графический интерпретатор функций, принимающую от пользователя уравнения плоских кривых в параметрической форме (x=fx(t); y=fy(t)) и диапазон изменения параметра t и строящая по этим уравнениям графики соответствующих функций.

Для решения этой задачи нам, очевидно, пригодятся рассмотренные в предыдущих разделах программы синтаксического анализа заданных в строковом виде формул, интерпретатор строковых выражений.

Составление любой программы лучше всего начинать с составления ее словесного плана-алгоритма, вначале крупноблочного, а затем постепенно детализируемого. Это даст возможность структурировать программу разбиением на отдельные подпрограммы (разложить сложную задачу на ряд простых), определить функциональное назначение всех подпрограмм.

Итак, начнем:

1.Сохраним содержимое экрана при входе в нашу программу, чтобы восстановить перед выходом из нее и "не наследить".

2.Оформим интерьер для ввода-вывода. Выведем приглашение пользователю для ввода уравнений кривых. Дадим ему возможность свободно редактировать текст вводимых формул, перемещаясь между предусмотренными для этого строками ввода. Результаты ввода и редактирования должны отображаться и на экране и в буфере памяти.

3.После завершения ввода вычислим значения переменных с определенным шагом и результаты вычисления сохраним в таблице для последующего построения графика. Для этого вызовем подпрограмму - интерпретатор текста формул для рас-

чета массивов значений x и y при различных значениях параметра t.

4.Инициализируем графический режим и определим основные параметры - максимальные значения координат и пр.

5.Оформим интерьер для вывода графика функции, просчитаем масштабы по осям координат.

6.Нарисуем график и когда пользователь им налюбуется и нажмет например Esc - предложим ему продолжить работу с другой кривой или с той же самой, но с другими коэффициентами. По желанию пользователя завершим работу программы.

Пункты с 2-го по 6-й будем повторять циклически, пока пользователь не введет признак завершения программы - нам этот признак тоже предстоит придумать.

8.Выйдем из графического режима и восстановим содержимое текстового экрана, из которого мы стартовали.

Вот крупным планом и все - теперь на очереди детализация плана.

Реализация этого плана на Си приведена в разделе 6.18. Так как библиотеки BGI - графики в Паскале и Си полностью аналогичны, Паскаль - реализацию этой программы мы оставляем на самостоятельную проработку с возможностью "консультаций" в приведенном Си - варианте.

8.17. Работа с массивами структур файлового хранения

Массивы или списки структур в файлах представляют собой обычно так называемые базы данных - многочисленные совокупности информационных блоков, содержащих возможно разнотипные, но связанные по смыслу и способам совместной обработки данные - это могут быть сведения о служащих некоторой фирмы, библиографический справочник, сведения об автомобилях и их владельцах в базе данных Госавтоинспекции региона, регистрационные данные о посетителях стоматологической поликлиники, операционные сведения за текущий день, с начала недели, месяца, года в коммерческом банке и многое другое.

В настоящем разделе мы рассмотрим методы работы с записями (структурами) файлового хранения на простом примере создания пополняемого толкового словаря. Реализующая его программа должна будет обеспечить предоставление пользователю следующих услуг: добавление записей в словарь, логическое и физическое удаление записей, поиск по ключевому сло-

ву, последовательный просмотр содержимого словаря и списка помеченных к удалению записей.

Каждая запись в словаре - это структура с двумя полями типа символьных массивов (строк), одно из которых содержит ключевое слово, а другое, большей длины, толкование этого слова.

Метод создания словаря будет ясен из приводимого ниже текста программы и сопровождающих его комментариев.

```
uses crt,dos;
(*Некоторые необходимые константы*)
const
  MAXWORDS=1000;(*максимальное количество слов в словаре*)
  KEYLENGTH=20;(*длина слова*)
  VALUELENGTH=255;(*длина толкования*)
  NUMPUNKTS=8;(*количество пунктов в меню*)

(*в этой структуре хранятся основные справочные сведе-
ния в памяти*)
type shortrec=record
  IsDel:char;(*если ==0, запись пустая или помечена к удалению*)
  key:string[KEYLENGTH];(*слово*)
end;
ashortrec=array [0..MAXWORDS-1] of shortrec;
pshortrec=^ashortrec;

(*эта структура для файлового хранения записей *)
type rec=record
  memrec:shortrec;(*признак занятости и слово-ключ*)
  value:string[VALUELENGTH];(*значение по данному ключу*)
end;
frec=file of rec;

(*прототипы используемых процедур и функций*)
procedure add;forward;
procedure del;forward;
procedure find;forward;
procedure view;forward;
procedure pack;forward;
procedure undel;forward;
procedure dellist;forward;
procedure quit;forward;
procedure init;forward;
function findinmemory(key:string):boolean;forward;
function rcompare(f,s:string):boolean;forward;
function getnumber(key:string):longint;forward;
function rtolower(c:char):char;forward;

(*этот массив структур содержит список возможных дей-
ствий и реакцию на них*)
```

```
  type unknown=record
  msg:string[30];(*сообщение - пункт меню*)
  (*указатель на функцию, соответствующую данному пункту*)
  func:procedure;
end;

const
message:array[0..NUMPUNKTS-1] of unknown=(
(*количество элементов в массиве равно количеству пунктов меню*)
(msg:'0. Добавить запись';func:add),
(msg:'1. Удалить запись';func:del),
(msg:'2. Найти запись';func:find),
(msg:'3. Просмотр словаря';func:view),
(msg:'4. Упаковать словарь';func:pack),
(msg:'5. Отменить удаление';func:undel),
(msg:'6. Список удалённых';func:dellist),
(msg:'7. Выход';func:quit));

var
(*временная запись, используемая для промежуточных действий*)
  temp:rec;
  dictionary:pshortrec;(*указатель на справочник в памяти*)

(*процедура начальной инициализации*)
procedure init;
var
  fp:frec;
  i,RecordCount:longint;
begin
  (*отводим память под справочник*)
  GetMem(dictionary,MAXWORDS*sizeof(shortrec));
  if not longbool(dictionary) then(*если не удалось отвести*)
  begin
  writeln('Не хватает памяти под справочник'); halt(1);
  end;
  (*очищаем память, выделенную под справочник*)
  FillChar(dictionary^[0],MAXWORDS*sizeof(shortrec),0);
  (*открываем двоичный файл для чтения и записи*)
  {$I-}
  (*проверка на существование*)
  Assign(fp,ParamStr(1)); FileMode:=2; Reset(fp);
  if ioresult<>0 then
  begin
    Assign(fp,ParamStr(1)); FileMode:=2; ReWrite(fp);
    if IOResult<>0 then(*нет места, защита от записи etc*)
    begin
      writeln('Не могу открыть файл ',ParamStr(1));
      (*освобождаем память из под справочника*)
    FreeMem(dictionary,MAXWORDS*sizeof(shortrec)); halt(0);
    end;
```

```
    close(fp);
    FileMode:=2;Reset(fp);(*переоткрываем вновь созданный файл*)
  end;
  (*определяем число пар слов-значение в словаре*)
  RecordCount:=FileSize(fp);
  for i:=0 to RecordCount-1 do
  begin (*и считываем слова в справочную часть*)
    read(fp,temp);
    dictionary^[i]:=temp.memrec;
  end;
  close(fp);(*закрываем файл*)
  {$I+}
end;

(*Процедура добавления записи*)
procedure add;
var
  fp:frec;  i:longint;
begin
  {$I-}
  Assign(fp,ParamStr(1));  FileMode:=2; Reset(fp);
  FillChar(temp,sizeof(rec),0);(*очищаем рабочую запись*)
  writeln('Введите слово : ');(*запрашиваем ключевое слово*)
  readln(temp.memrec.key);
  (*если его нет в словаре*)
  if( not findinmemory(temp.memrec.key))then
  begin
(*вводим с клавиатуры значение с ограничением на длину*)
writeln('Введите значение этого слова (Enter - конец ввода) :');
readln(temp.value);
temp.memrec.IsDel:='A';(*признак неудалённости записи*)
for i:=0 to MAXWORDS-1 do(*дрейф по справочнику в памяти*)
if not bytebool(dictionary^[i].IsDel) then
      (*в поисках пустой (удалённой) ячейки*)
     begin (*заносим слово в память*)
       dictionary^[i]:=temp.memrec;
       Seek(fp,i);
       write(fp,temp);(*а вместе со значением - на диск*)
       break;
     end;
    if i=MAXWORDS then
    writeln('Нет места в словаре для добавления нового слова');
  end
  else(*если нашли слово в памяти*)
  writeln('Такое слово уже есть в словаре. Попробуйте
изменить его значение');
  readkey;
  close(fp);
  {$I+}
end;
```

```
(*функция поиска слова в справочнике*)
function findinmemory(key:string):boolean;
var
  i:longint;
begin
(*слово считается найденным, если оно не помечено к удалению и совпадает с искомым с точностью до регистра*)
  for i:=0 to MAXWORDS-1 do
    if(bytebool(dictionary^[i].IsDel))and
    ( not rcompare(key,dictionary^[i].key)) then
    begin
      findinmemory:=boolean(1); exit;
    end;
  findinmemory:=false;
end;

(*сравнение двух строк без учёта регистра по аналогии с stricmp, но с проверкой на русские буквы*)

function rcompare(f,s:string):boolean;
var
  i:byte;
begin
  if(Length(f)<>Length(s))then
  begin
  rcompare:=true;  exit;
  end;
  for i:=1 to Length(f) do
    if (rtolower(f[i])<>rtolower(s[i])) then
    begin
      rcompare:=true;   exit;
    end;
  rcompare:=boolean(0);
end;

(*переводит символ в нижний регистр, возвращая результат*)
function rtolower(c:char):char;
var
  d:byte;
begin
  d:=byte(c);
  if(d>=byte('A'))and(d<=byte('Z')) then inc(d,$20);
  if(d>=byte('A'))and(d<=byte('П')) then inc(d,$20);
  if(d>=byte('P'))and(d<=byte('Я')) then inc(d,$50);
  if(d=byte('Ё')) then d:=byte('ё');
  rtolower:=char(d);
end;

(*процедура удаления слова*)
procedure del;
var
  i:longint;  fp:frec;
```

```
begin
  {$I-}
  Assign(fp,ParamStr(1)); FileMode:=2;  Reset(fp);
  FillChar(temp,sizeof(rec),0);(*очищаем временную запись*)
  writeln('Введите слово : ');
  readln(temp.memrec.key);(*вводим удаляемое слово*)
  if(findinmemory(temp.memrec.key))(*если таковое имеется*)then
  begin
    (*получаем номер записи с данным словом*)
    i:=getnumber(temp.memrec.key);
    dictionary^[i].IsDel:=#0;(*устанавливаем признак удаления*)
    Seek(fp,i);   (*позиционируемся на эту запись в файле*)
    read(fp,temp);    temp.memrec.IsDel:=#0;
    Seek(fp,i);   (*позиционируемся на эту запись в файле*)
    write(fp,temp);   (*записываем с новым признаком*)
  end
  else
    writeln('Такого слова в словаре нет');
  readkey;
  close(fp);
  {$I+}
end;
```

(*функция, ищущая слово в памяти и возвращающая номер записи с ним*)

```
function getnumber(key:string):longint;
var
  i:word;
begin
  getnumber:=-1;   (*а может и не найдём...*)
  for i:=0 to MAXWORDS-1 do(*дрейфуем по словам до тех пор,*)
    if( not rcompare(key,dictionary^[i].key))then
    begin         (*пока не найдём; тогда*)
    getnumber:=i;  (*возвращаем номер*)   exit;
    end;
end;

(*если удалённая запись ещё не затёрта, признак  уда-
ления с неё можно снять*)

procedure undel;
var
  fp:frec;
  i:integer;
begin
  {$I-}
  Assign(fp,ParamStr(1));  FileMode:=2; Reset(fp);
  FillChar(temp,sizeof(rec),0);(*очищаем рабочую запись*)
  writeln('Введите слово : ');
  (*вводим предположительно удалённое слово*)
  readln(temp.memrec.key);
```

```
  for i:=0 to MAXWORDS-1 do   (*ищем только среди удалённых*)
    if( not rcompare(dictionary^[i].key,temp.memrec.key))
       and( not boolean(dictionary^[i].IsDel)) then
    begin       (*убираем признак удалённости*)
      dictionary^[i].IsDel:='R';
      Seek(fp,i);       (*позиционируемся на удалённую запись*)
      read(fp,temp);
      temp.memrec.IsDel:='R';
      Seek(fp,i);      (*позиционируемся на эту запись в файле*)
      write(fp,temp);  (*и восстанавливаем её*)
      writeln('Восстановлено!');
      i:=-1;        (*признак восстановления*)
      break;
    end;
  if wordbool(i+1) then(*если не удалось восстановить*)
    writeln('Такого слова в словаре не было');
  readkey;
  close(fp);
  {$I+}
end;

(*поиск всех слов, содержащих заданное как подстроку*)
procedure find;
var
  key:string[KEYLENGTH];
  j,i,count,RecordCount:longint;
  fp:frec;
begin
  {$I-}
  Assign(fp,ParamStr(1));  FileMode:=0;  Reset(fp);
  FillChar(temp,sizeof(rec),0);(*обнуляем рабочую запись*)
  writeln('Введите слово : ');
  readln(key);(*вводим подстроку для поиска*)
  count:=0;
  for i:=1 to byte(key[0]) do
    key[i]:=rtolower(key[i]);(*преобразуем к нижнему регистру*)
  RecordCount:=FileSize(fp); (*определяем число записей*)
  for i:=0 to RecordCount-1 do
  begin
    read(fp,temp);
    for j:=1 to byte(temp.memrec.key[0]) do
      temp.memrec.key[j]:=rtolower(temp.memrec.key[j]);
    if bytebool(pos(key,temp.memrec.key)) and
    bytebool(temp.memrec.IsDel) then
    begin
      writeln('Слово - ',temp.memrec.key,',
              значение - ',temp.value);
      inc(count);
    end;
  end;
  if not longbool(count) then
```

```
    writeln('Такого слова в словаре нет');
  readkey;
  close(fp);
  {$I+}
end;

(*вывод на экран всего словаря*)
procedure view;
var
  i,count:longint;  fp:frec;
begin
  {$I-}
  Assign(fp,ParamStr(1)); FileMode:=0; Reset(fp);
  writeln('Содержимое словаря');
  FillChar(temp,sizeof(rec),0);
  count:=0;  i:=0;
  while (i<MAXWORDS) and not eof(fp) do
  begin
  if bytebool(dictionary^[i].IsDel)(*если слово не удалено*)then
  begin
  Seek(fp,i); read(fp,temp);(*считываем его значение*)
inc(count);
writeln(count,' Слово - ',temp.memrec.key,',
значение - ',temp.value);
readkey;(*печатаем и даём полюбоваться до нажатия клавиши*)
   end;
    inc(i);
  end;
  writeln('Всего записей : ',count);(*вывод статистики*)
  readkey;
  close(fp);
  {$I+}
end;

  (*вывести список удалённых словарных единиц*)
procedure dellist;
var
  i,count:longint;
begin
(*пара слово-значение считается удалённой, если слово
есть, а признак удаления установлен*)
  writeln('Список удалённых слов:');
  count:=0;
  for i:=0 to MAXWORDS-1 do
    if not bytebool(dictionary^[i].IsDel)and
      bytebool(Length(dictionary^[i].key)) then
    begin
     inc(count);
     writeln(count,' слово - ',dictionary^[i].key);
    end;
  writeln('Всего удалённых записей : ',count);(*статистика*)
  readkey;
```

```
end;

(*Записи, помеченные к удалению, продолжают занимать место,
увеличивая размер файла, а потому их имело бы смысл физически
удалить*)

procedure pack;
var
  RecordCount,i:longint;
  fp,ff:frec;
begin
  {$I-}
  write('Упаковываю...');
  Assign(fp,ParamStr(1));  FileMode:=0;  Reset(fp);
  Assign(ff,'vocab.tmp');  FileMode:=1;  ReWrite(ff);
  RecordCount:=FileSize(fp);
  (*переписываем только существующие записи*)
  for i:=0 to RecordCount-1 do
  begin
    read(fp,temp);
    if bytebool(temp.memrec.IsDel) then
     write(ff,temp);
  end;
  close(fp); close(ff);
  {$I+}
  Erase(fp);  Rename(ff,ParamStr(1));
  writeln(#8#8#8#8#8#8#8'ано!   ');
  readkey;
end;

(*Процедур выхода с освобождением памяти*)
procedure quit;
begin
  (*освобождаем выделенную память*)
  FreeMem(dictionary,MAXWORDS*sizeof(shortrec)); halt(0);
end;

var
  i:integer; what:byte;
begin
  (*в качества параметра передадим имя словаря*)
  if ParamCount<1 then
  begin
    writeln('Используйте: ',ParamStr(0),' vocab.dat'); exit;
  end;
  init;(*начальная инициализация*)
  directvideo:=boolean(0);
  (*в бесконечном цикле обработки сообщений от клавиатуры*)
  while true do
  begin
    writeln;(*выводим пустую строку перед списком*)
    for i:=0 to -1+NUMPUNKTS do(*распечатываем все пункты меню*)
```

```
      writeln(message[i].msg);
    write('Ваш выбор : ');(*приглашение к диалогу*)
    (*ввод с отсечением недопустимых номеров пунктов меню*)
    repeat
      what:=byte(readkey);
    until (what-ord('0')>=0)and(what-ord('0')<NUMPUNKTS);
    dec(what,ord('0'));(*преобразуем цифровой символ в номер*)
    (*печатаем, что выбрали*)
   writeln(Copy(message[what].msg,3,Length(message[what].msg)-2));
    message[what].func;(*и вызываем соответствующую функцию*)
  end;
end.
```

9. Введение в объектно-ориентированное программирование на Borland C++.

9.1. Предварительные сведения.

В предыдущих материалах мы уже использовали некоторые возможности, присущие не языку Си, а C++ - однострочные комментарии, объявление переменных в любом месте подпрограмм, модификатор `const` для создания переменной в памяти, доступной только для чтения. При этом предполагалось, что вы компилируете ваши программы с помощью компилятора как минимум Турбо C++ или лучше Borland C++, в противном случае следует ожидать "непонимания" со стороны компилятора с языка С.

Язык C++ разработан в основном для "прививки" к основе популярного языка Си возможностей ***объектно-ориентированного подхода к программированию*** **(ООП)** и является языком с "гибридными" возможностями работы как в процедурной, так и в объектной методологиях. Что же собой представляет объектно-ориентированное программирование? Во время создания многофункциональных программ обработки векторов и матриц у вас мог возникнуть вопрос: а нельзя ли было создать типы данных, например, vector и matrix такими, чтобы для них были определены автоматическое размещение в памяти (конструирование) при указании в момент объявления например имени файла с компонентами и размерностей, чтобы после этого были определены для этих типов операции сложения, умножения, транспонирования, ортогонального преобразования и многие другие, записываемые так же легко, как и для скалярных типов `int`, `double`?

Ответ прост: можно, и создание абстрактных пользовательских типов данных с переопределением для них стандартных операций - одна из основных концепций ООП - она так и называется "**АБСТРАКЦИЯ**". Под абстракцией при этом подразумевается переход от конкретного числового представления информационных объектов в компьютере к конкретике некоторой прикладной области, которую обслуживает программист, создавая свои типы. Три другие основы ООП носят название **ИНКАПСУЛЯЦИЯ, НАСЛЕДОВАНИЕ, ПОЛИМОРФИЗМ** и мы последовательно познакомимся с ними, но прежде всего рассмотрим некоторые дополнения к языку Си, прямо не связанные с ООП, но

существенно расширяющие предоставляемый компилятором сервис.

9.2. Дополнения к Си, не связанные с ООП.

ТИПИРОВАННЫЕ КОНСТАНТЫ - это переменные только для чтения, которые обязательно должны получить значение в месте объявления:

```
const unsigned u=3456;
```

и могут быть только статическими, доступными только в пределах своего файла. Чтобы сделать их доступными из других единиц компиляции, придется объявлять их с ключевым словом `extern`

```
extern const unsigned u=3456;
```

Чтобы использовать ее в другом файле, нужно тоже использовать `extern` - модификатор и не присваивать значение:

```
extern const unsigned u;
```

ФУНКЦИИ.

ПЕРЕГРУЗКА ФУНКЦИЙ - под этим понимается возможность устранения конфликтов имен функций, т.е. возможность использовать в программе функции с одинаковыми именами, если они имеют отличающиеся списки аргументов. С этой возможностью связано также требование предварительного объявления прототипа функции до ее вызова на выполнение - это объявление используется компилятором для определения правильности передачи аргументов при вызове, особенно для функций с одинаковыми именами.

Если вы не хотите, чтобы функция могла изменять значения переменных по передаваемым ей указателям, то можете объявить ее параметры константами:

```
void func(const char * ch_ptr) {...}
```

Допускается также задание в функциях **АРГУМЕНТОВ ПО УМОЛЧАНИЮ**:

```
double dfunc(double d1, double d2=3.14, char* cp="");
```

Если при вызове теперь задать только значение d1, значения 2-х других параметров компилятор подставит сам по умолчанию. Но если при вызове будет задано значение cp, то все предыдущие тоже должны задаваться.

Спецификатор inline для функций

Помещается перед определением функции для того, чтобы код функции был подставлен компилятором непосредственно в

место ее вызова для исключения накладных расходов по вызову функции (помещение параметров в стек, оформление возврата и прочее).

inline void func(void) {тело функции}

При использовании inline - подстановки мы избавляемся от необходимости повторять многократно выкопировку одного и того же повторяющегося в программе фрагмента кода (как правило, небольшого по размеру) и сокращаем время выполнения программы за счет исключения вызовов подпрограмм и за счет увеличения размера занимаемой программой памяти.

ССЫЛКИ - новый тип данных в C++. Они тоже содержат адрес объекта, но при обращении по ссылке не надо применять операцию разадресации, как мы это делаем при доступе по указателям. Переменная типа "ссылка" объявляется со значком & и в месте объявления должна получить значение - ссылаться можно только на ранее "прописанный" в памяти объект. Именно поэтому ссылки еще называют псевдонимами:

```
int y=16;
int& x=y;
```

Теперь любое изменение x приведет к изменению y и наоборот:

```
x=10; //y стало равным 10 и x тоже
y=15; //x стал равным 15 и y тоже
```

УПРАВЛЕНИЕ ПАМЯТЬЮ. К таким функциям, как malloc() и free(), в C++ добавлены 3 встроенных в язык оператора new, delete, delete[]. Оператор new используется для создания объекта в куче:

```
int nf=18;
float * fp= new float[nf];
```

Оператор new позволяет попутно заполнить выделяемую память заданным значением: float * fp= new float[nf] (0); (массив заполняется нулями). Но надо быть внимательным при использовании new - он, в отличие от malloc, вернет ненулевой указатель, если вы по ошибке запросили 0 байтов памяти и запись в эту память имеет непредсказуемые последствия. Освобождение памяти осуществляют операторы delete, или delete[] (для массивов).

9.3. Классы.

Классы - это и есть механизм для создания абстрактных объектов прикладного программиста, наделенных естественно определенными реальными свойствами. Все, с чем мы работаем в нашей программе, можно рассматривать как объект определенного класса. Формально класс - это структура, содержащая набор взаимосвязанных по смыслу и ххарактеру обработки "полей-данных" и "полей-функций" для манипулирования данными; эти функции называют "функциями-членами" (member FUNCTION) или методами класса; они и определяют поведение объектов этого класса.

Перед использованием класса для создания (конструирования) объектов в памяти необходимо описать свойства, состав класса, так же, как мы это делали со структурами. Для объявления класса используют ключевое слово class, за ним следует произвольное имя класса и далее тело класса в фигурных скобках, содержащее типы и имена полей-данных, прототипы функций-членов и при необходимости определения функций-членов (для упрощения записи определения функций-членов могут быть вынесены за пределы тела класса). После закрытой скобки тела ставится точка с запятой.

Так, например, для работы с векторами нам достаточно данных о размерности и адресе вектора в памяти и объявление векторного класса с для создания массива элементов типа, например double, с этими членами (другие мы добавим позже) может иметь вид:

```
class vector
{
 long m;              //размерность (длина) вектора
 double *vec;         //адрес вектора
public:
 //функция - член, возвращающая размерность вектора
 long getm() { return m; }
 ...                  //другие члены класса
};
```

9.4. Инкапсуляция.

Область видимости переменной-члена простирается от точки объявления переменной до конца объявления класса и за пределами этого объявления доступ к ней невозможен.

Для доступа к данным класса обычно используются функции-члены, в нашем объявлении это пока одна функция, возвращающая размерность вектора. Перед ней стоит слово `public` (общедоступный), обозначающее свободный доступ. Всего уровней доступности членов класса 3: `private`, `public`, `protected`. По умолчанию члены класса - приватные, доступные только функциям-членам класса или привилегированным пользователям - например, дружественным классам. Уровень `protected` (защищенный) предназначен для использования функций-членов класса производными классами, объявившими свой базовый класс как `public` - но об этом подробнее в разделе о наследовании.

Таким образом, класс скрывает внутреннюю информацию - это и подразумевается под инкапсуляцией, но он создается для использования в программах, поэтому не может не иметь общедоступных членов. Члены-данные, как и члены-функции, могут быть объявлены с любым из трех уровней доступа.

9.5. Конструкторы.

Объявленный нами класс vector, очевидно, не очень удобен - он не содержит методов создания объектов класса. Откуда возьмется значение возвращаемой размерности, указатель на вектор? Для выполнения этой работы класс может иметь специальные функции-члены для построения объектов этого класса и эти функции называются конструкторами.

Конструктор в теле класса не обязателен, но можно определить и несколько конструкторов. C++ гарантирует, что, если класс имеет конструктор, он будет вызван при создании объекта класса до выполнения любой другой функции класса.

Какими способами может быть сконструирован вектор в памяти? В простейшем случае этим может озаботиться программа до создания объекта класса - в этом случае конструктор должен будет получить при создании векторного объекта в виде аргументов указатель на начало вектора в памяти и размерность вектора. Другая возможность - передать конструктору имя файла, в котором лежит вектор, и его размерность - тогда размещением вектора в оперативной памяти должен будет заняться соответствующий конструктор. Таких вариантов конструирования векторов (как обычно и объектов других классов) может быть достаточно много и класс, предоставляющий программисту сервис по работе с векторами должен иметь достаточно представительный комплект конструкторов.

Синтаксически конструкторы различаются компилятором от других функций членов по именам, совпадающим с именем класса, а друг от друга - по спискам параметров. Дополненный объявлениями прототипов конструкторов класс vector может выглядеть так:

```
class vector
{ //приватные данные
 long m;                 //размерность (длина) вектора
 double *vec;//указатель на элементы вектора
public:                  //общедоступные функции
 vector(char *);  //конструктор загрузки вектора из файла
 vector();        //создание пустого вектора размерности 1
 vector(long N);        //создание пустого вектора размерности N
 vector(vector &);      //конструктор копирования
 long getm() { return m; }
 ...                     //другие члены класса
};
```

9.6. Определение функций-членов.

Функция getm() определена нами в теле класса. Часто это неудобно просто для чтения, если текст функции занимает значительное место или количество функций-членов велико. В этих случаях определение функции выносится за тело класса; при этом перед именем функции через :: указывается имя класса, к которому она относится - в различных классах могут быть функции - члены с одинаковыми именами (но не конструкторы - имена классов должны быть уникальными) и такой квалификатор необходим. Определение функций-членов за пределами класса покажем на примере одного из конструкторов для случая, когда нам известна лишь размерность вектора, но неизвестны его составляющие и мы предполагаем, что они нулевые:

```
vector::vector(long a):m(a)
{
 long i;
 if(m<=0)                        //проверка размерности
 { cerr<<"Некорректный размер вектора\n"; exit(0);}
 vec=new YourOwnFloatType[m]; //попытка выделения памяти
 if(vec==NULL)//если не удалось выделить память
 { cerr<<"Not enough memory\n"; exit(0);}
 for(i=0;i<m;vec[i++]=0);   //обнуление компонентов
```

}

ЗАМЕЧАНИЯ:

Нельзя объявить конструктор, возвращающий какое -либо значение. Задача конструктора , построить объект , что не связано с возвращением значения.

Конструктор объявляемый без аргументов, называется конструктором по умолчанию (default constructor).

Если конструктор по умолчанию не определен в описании класса, то Borland C++ объявляет его автоматически. Созданный компилятором конструктор по умолчанию просто выделяет память при создании объекта своего класса.

9.7. Деструкторы.

Для выполнения действий, обратных конструированию, в частности, удалению объектов из памяти, служит другая специальная функция-член под названием деструктор. Для нее резервировано специальное имя, которое состоит из имени класса, перед которым стоит знак ~.

Для нашего класса деструктор может выглядеть так:

```
vector::~vector()
{delete []vec;} //уничтожение динамического массива
```

ЗАМЕЧАНИЯ:

- Деструкторам нельзя передавать аргументы
- В каждом классе может быть объявлен только один деструктор.
- Деструкторы не имеют типов возвращаемых значений.

9.8. Шаблоны классов и функций.

Пока мы определили наш класс vector способным обрабатывать только векторы с составляющими типа double, жестко зафиксировав тип указателя на элементы вектора; это может оказаться как избыточным, так и недостаточным для конкретных применений. Было бы неплохо придать классу большую общность, сделав его совместимым и с другими типами данных. C++ предоставляет такую возможность создания шаблона класса, способного работать с некоторым обобщенным типом данных - этот шаблон используется компилятором для создания реального класса с конкретно указанным типом данных на этапе компиляции.

Объявление такого класса начинается с ключевого слова template с последующим <class имя обобщенного типа>.

Использование шаблона класса отличается от работы с другими классами только синтаксисом объявления. При реализации шаблона класса с передачей ему определенного типа данных создается шаблонный класс и компилятор автоматически построит все функции-члены, работающие с указанным типом данных.

Возвращаемые типы функций-членов шаблона класса записываются после угловых скобок имени шаблона:

template <class T >int MyClass<T>::func (T&) {/* . . .*/ } ;

Чтобы создавать функции для работы с обобщенными типами аргументов, можно использовать предоставляемый C++ механизм создания шаблонов функций. Для этого объявление начинают со слова template с последующим <class имя типа >, указывающим, что в функции будет использоваться какая-то переменная с обобщенным типом.

При использовании шаблона функции с конкретным типом данных создается шаблонная функция. При вызове шаблонной функции компилятор генерирует функцию для обработки типа данных, который указан в операторе вызова функции. Шаблонные функции освобождают программиста от необходимости составлять их для каждого набора типов аргументов, который может понадобится в программе.

Применим сказанное о шаблонах для реконструкции нашего класса vector к работе с обобщенными типами:

```
template <class YourOwnFloatType> //подставьте свой тип
class vector
{ //приватные данные
   long m;              //размерность (длина) вектора
  YourOwnFloatType *vec;//указатель на элементы вектора
 public:               //общедоступные данные и функции
  vector(char *);       //загрузка вектора из файла
  vector();             //создание пустого вектора размерности 1
  vector(long ); //создание пустого вектора размерности m
  vector(long,YourOwnFloatType *);
  vector(vector &);
  ~vector();
  long getm() { return m; }
};
```

Приведенный ранее конструктор преобразуется к виду:

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long a):m(a)
{
 long i;
```

```
 if(m<=0)//проверка размерности
 { cerr<<"Некорректный размер вектора\n"; exit(0);}
 vec=new YourOwnFloatType[m];//попытка выделения памяти
 if(vec==NULL)                    //если не удалось
 { cerr<<"Не хватает памяти\n"; exit(0); }
 for(i=0;i<m;vec[i++]=0); //обнуление элементов
}
```

9.9. Друзья класса.

Если два класса тесно связаны друг с другом и необходимо обеспечить одному из них неограниченный доступ к членам второго, то это достигается объявлением дружественного класса, имеющего доступ ко всем членам второго класса. Дружественность объявляется ключевым словом `friend` и другом может быть объявлен как целый класс, так и отдельная функция.

СВОЙСТВА ДРУЗЕЙ

Свойство дружественности не транзитивно. Если класс A дружественен классу B, а класс B - классу C, то A не становится автоматически дружественным классу C. Конечно, A можно объявить дружественным классу C.

Дружественность в C++ необязательно взаимна. Если A дружественен B, то это вовсе не означает, что B дружественен A. Чтобы сделать два класса взаимно дружественными, нужно объявить каждый из них дружественный другому.

Функция, объявленная дружественной классу, имеет доступ ко всем членам данных класса, независимо от того, в какой секции она объявлена, в `private`, `protected` или `public`.

Прежде чем продемонстрировать использование дружественных функций, рассмотрим еще одну важную возможность C++.

9.10. Перегрузка операторов

Мы уже упоминали о нашем желании приспособить обычные операторы сложения, вычитания, умножения и пр. к записи соответствующих операций с векторами и матрицами. Но встроенные операторы C++ не могут выполнять их над создаваемыми нами типами.

Отметим следующее: хотя операторы представляются нам как некоторый прямой механизм выполнения математических или логических операций, на самом деле они представляют

собой всего лишь способ записи вызова необходимых подпрограмм. Отсюда вытекает возможность создания перегруженных операторов для работы с различными типами объектов.

ОПЕРАТОРЫ - ЭТО ОБРАЩЕНИЯ К ФУНКЦИЯМ.

Операторы могут быть представлены либо как операторные функции-члены класса, либо как их друзья. Синтаксис объявления операторных функций требует указания ключевого слова `operator` со следующим за ним знаком перегружаемого оператора. Прототип дружественной нашему классу операторной функции суммирования 2-х векторов, получающей в виде аргументов ссылки на слагаемые, может выглядеть так

```
friend vector operator+(vector &, vector &);
```

и операция сложения 2-х векторов всегда эквивалентна вызову на выполнение этой функции (разумеется, она должна быть определена).

ОСОБЕННОСТИ ПЕРЕГРУЖЕННЫХ ОПЕРАТОРОВ КАК ФУНКЦИЙ-ЧЛЕНОВ

Имена функций-членов должны начинаться со слова `operator` с последующим символом программируемой операции. Например, функция-член при реализации оператора / должна называться `operator/()`. Количество аргументов для унарных операций 0, для бинарных - только один. Это ограничение очевидно - унарная операция осуществляется над "самим собой", а указатель `this` на свой собственный тип все функции-члены получают неявно в виде первого аргумента. Бинарная операция принадлежит одному из слагаемых и для ее реализации необходима только ссылка на второе слагаемое.

Когда компилятор в выражении сталкивается с перегруженным оператором для класса X, он ищет подходящую функцию-оператор класса X, используя обычный для перегруженных функций правила сопоставления аргументов.

ПЕРЕГРУЖЕННЫЕ ОПЕРАТОРЫ КАК ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

При перегрузке операторов вне тела классов перегружаются сами встроенные операторы в C++ , и новые операторы становятся похожими на глобальные.

ОПЕРАТОР ПРИСВОЕНИЯ

Может быть объявлен только как функция-член, но не как друг. Если оператор присвоения для класса явно не определен,

но необходим компилятору для разрешения какого-то оператора, то автоматически генерируется оператор по умолчанию, выполняющий побайтовую операцию копирования.

ОПЕРАТОР ИНДЕКСИРОВАНИЯ `[]`.

Рассматривается как бинарный и может принять только один аргумент:

a = B[20];

Дружественный оператор индексирования также недопустим. Оператор индексирования можно заставить вернуть ссылку на значение и, следовательно, использовать оператор в левой части выражения присвоения: a[20]= B[30];

ОГРАНИЧЕНИЯ НА ПЕРЕГРУЗКУ ОПЕРАТОРОВ

Нельзя перегружать следующие операторы:

.	селектор члена структуры
*	оператор доступа к члену по указателю
::	оператор разрешения видимости
?:	условный тернарный оператор

Теперь можем дополнить наш класс некоторыми операторными функциями:

```
template <class YourOwnFloatType>//подставьте свой тип
class vector
{ //приватные данные
  long m;                 //размерность (длина) вектора
  YourOwnFloatType *vec;//указатель на элементы вектора
public:                   //общедоступные данные и функции
  vector(char *);         //загрузка вектора из файла
  vector();         //создание пустого вектора размерности 1
  vector(long);     //создание пустого вектора размерности m
  vector(long,YourOwnFloatType *);
  vector(vector &);
  ~vector();//destructor
  friend vector operator+(vector &,vector &);//сложение
  friend vector operator-(vector &,vector &);//вычитание
  friend YourOwnFloatType operator*(vector &,vector &);
  friend vector operator*(YourOwnFloatType ,vector &);
  friend vector operator*(vector &,YourOwnFloatType );
  vector operator=(vector &);//присвоение
  vector operator-();         //unary minus
  vector operator~(); //определение направляющих косинусов
 YourOwnFloatType operator!();//модуль вектора
  friend long operator==(vector &,vector &);//сравнение
```

```
 friend long operator!=(vector &,vector &);
 YourOwnFloatType &operator[](long a);//индексирование
 long getm() { return m; } //размерность вектора
};
```

ОПРЕДЕЛЕНИЕ ОПЕРАТОРНЫХ ФУНКЦИЙ.

Рассмотрим определения некоторых из объявленных в классе vector операторных функций перегрузки:

ИНДЕКСАЦИЯ, которая необходима:

- при получении составляющей вектора по её номеру и
- при изменении одного из компонентов.

```
template <class YourOwnFloatType>
YourOwnFloatType
&vector<YourOwnFloatType>::operator[](long a)
{
 static YourOwnFloatType error=MAX_LONGDOUBLE;
 if(a>=0&&a<m) return vec[a];              //если всё ОК
 else                                      //при выходе за пределы
{cerr<<"Индекс "<<a<<" вне диапазона вектора\n"; return error;}
}
```

СЛОЖЕНИЕ векторов одинаковой размерности. Результатом сложения является вектор той же размерности, что и исходные, компонентами которого является сумма соответствующих компонент исходных векторов.

```
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator+
(vector<YourOwnFloatType> &f,vector<YourOwnFloatType> &s)
{if(f.m!=s.m)//проверка на равенство размерностей
{cout<<"Размерности векторов не совпадают \n";exit(0);}
vector<YourOwnFloatType> temp(f.m);//создаём временный вектор
//здесь работают операции индексирования для всех трёх векторов
for(long i=0;i<f.m;i++) temp[i]=f[i]+s[i];
return temp;
}
```

УНАРНЫЙ МИНУС :

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>
vector<YourOwnFloatType>::operator-()
{
 vector<YourOwnFloatType> temp(m);//создаём временный вектор
```

/*Если `this` - это указатель на текущий объект векторного класса, то *`this` - это сам текущий объект класса vector, то есть тот, с которым мы сейчас работаем. А к любому векторному объекту мы можем применить операцию индексирования */

```
 for(long i=0;i<m;i++) temp[i]=-(*this)[i];
 return temp;
}
```

ПРИСВОЕНИЕ.

При переписывании одного вектора в другой возможны два случая:

a) если размерность обоих векторов совпадает, то просто заменяем составляющие первого вектора компонентами второго;

b) в противном случае безжалостно уничтожаем первый вектор и создаём снова, используя второй как строительный материал.

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>
vector<YourOwnFloatType>::operator=
      (vector<YourOwnFloatType> &x)
{
 if(m!=x.m)
 {
 delete []vec; m=x.m; vec=new YourOwnFloatType[m];
 if(vec==NULL)
 { cout<<"Не хватает памяти\n"; exit(0); }
 }
 for(long i=0;i<m;i++) vec[i]=x[i];
 return *this;
}
```

КОММЕНТАРИЙ: присвоение - это бинарная операция, первым параметром которой является объект, которому присваивают, вторым - объект, который присваивают. При этом первый объект, в отличие от всех остальных бинарных операций, меняется, и он же возвращается в качестве результата (это бывает необходимым для операций вида a=B=c;)

9.11. Наследование свойств классов.

При создании класса трудно сделать его состав таким, чтобы он удовлетворял запросам всех пользователей - необходимость дополнений или изменений некоторых свойств возможна. При этом крайне желательно сохранить без переписывания (унаследовать) те методы и данные, которые подходят для соответствующей прикладной области. Механизм создания производных классов от уже существующих в ООП и C++ обслуживает эти потребности очень эффективно,

предоставляя возможность строить и последовательно строить и расширять классы по возрастающей сложности от базовых простых классов до сколь угодно сложных производных. Этот процесс основан на наследовании.

Язык C++, в отличие от других объектно-ориентированных языков, поддерживает не только простое , но множественное наследование, когда данный класс порождается сразу от нескольких родительских классов, наследуя поведение всех своих предков.

ОГРАНИЧЕНИЯ В НАСЛЕДОВАНИИ.

Невозможно унаследовать:

- конструкторы
- деструкторы
- операторы `new` , определенные пользователем
- операторы присвоения , определенные пользователем
- отношение дружественности

При создании объекта производного класса автоматически вызывается конструктор базового класса. После завершения конструирования объекта конструктор базового класса становится недоступным. Конструктор базового класса вызывается только компилятором в момент создания объекта производного класса. Конструктор, в отличии от других унаследованных функций, вызвать явно нельзя.

Создание производного класса записывается в простейшем случае так:

`class` A {. . . }; `class` B:A

Все наследуемые (т.е. защищенные и общедоступные) данные и функции класса A становятся по умолчанию приватными в классе B. Но можно указать явно спецификатор доступа базового класса:

`class` B:`public` A;

Все общедоступные имена базового класса будут общедоступными в производном классе и все защищенные имена будут защищенными в производном классе.

Для спецификаторов доступа базового класса нет ключевого слова `protected`.

ПЕРЕДАЧА АРГУМЕНТОВ В БАЗОВЫЙ КЛАСС

Если при создании объекта производного класса конструктору базового необходимы параметры, их можно передать через : после конструктора производного класса:

```
class A
{
```

```
 int a,B,c;
 public:
 A(int x,int y,int z){a=x,B=y,c=z;}
}

class B: public A
{
 int value;
 public:
 Second(int d): First(d,d+1,d+5){ value=d; }
 Second(int d,int e);
};

Second::Second(int d,int e) : First(d,e,13)
{ value=d+e;}
```

ПОРЯДОК ВЫЗОВА КОНСТРУКТОРОВ

Порядок вызова конструкторов при создании объектов производных классов в C++ фиксирован - он идет от корневого базового класса к производным по дереву наследования. Прежде всего строится базовый класс, затем производный. Если базовый класс, в свою очередь, является производным, то процесс рекурсивно повторяется до тех пор, пока не будет достигнут корневой класс.

ПОРЯДОК ВЫЗОВА ДЕСТРУКТОРОВ обратный к порядку вызова конструкторов.

КОРРЕКЦИЯ ХАРАКТЕРИСТИК БАЗОВОГО КЛАССА.

Если вас не совсем устраивает поведение одной или нескольких функций базового класса, то проще всего переопределить их в производном классе.

9.12. Полиморфизм и виртуальные функции.

Полиморфизм - это свойство C++ определять, какую функцию выполнять, в момент передачи вызываемой функции конкретного объекта. Такой процесс известен как позднее связывание.

Если же компилятор, базируясь на тексте программы, вызывает фиксированные имена функций, а потом компоновщик заменяет эти идентификаторы физическими адресами, такой процесс называется ранним связыванием, поскольку

идентификаторы функций связываются с физическими адресами до выполнения.

В C++ позднее связывание для функции определяется при ее объявлении с помощью ключевого слова `virtual` и это имеет смысл только для объектов, являющихся частью иерархии классов в системе наследования.

Пример использования виртуальных функций:

```
#INClude <stdio.h>
class A {
     public:
      virtual void Display(){ puts("\nКласс A");}
     };
class B: public A
{
     public:
      virtual void Display(){ puts("\nКласс B");}
     };
void Show(A* a)
{ a->Display();
//определить во время выполнения, какую использовать функцию
}

void main()
{ A *a=new A; B *B=new B;
 a->Display(); // использование A::Display();
 B->Display(); // использование B::Display();
 Show (a);  // использование A::Display();
 Show (B);  // использование B::Display();
}
```

Полиморфное поведение функции-члена `Display()` проявляется в функции `Show()`, в которой, просто изучив ее код, невозможно предсказать, какая именно функция, `A::Display()` или `B::Display()`, будет вызываться.

Виртуальная функция не обязательно должна переопределяться в производном классе. Чтобы распространить возможность переопределения функции вниз по дереву наследования, каждый производный класс должен объявлять одну и ту же функцию виртуальной.

ПЕРЕОПРЕДЕЛЕНИЕ ВИРТУАЛЬНОЙ ФУНКЦИИ

Функция, объявленная в производном классе, переопределяет виртуальную функцию в базовом классе только тогда, когда имеет то же имя и работает с тем же количеством и типом аргументов, что и виртуальная функция базового класса.

Если они отличаются хоть одним аргументом, то функция в производном классе считается совершенно новой и переопределения не происходит.

Чтобы переопределить виртуальную функцию базового класса, не обязательно объявлять функцию в производном классе виртуальной - это нужно, чтобы дать возможность переопределять ее в последующих производных классах.

АБСТРАКТНЫЕ КЛАССЫ

Классы, расположенные ближе к корню дерева наследования, часто имеют одну или несколько пустых (с пустым телом {}) виртуальных функций для согласованности действий в цепочке наследования.

C++ позволяет создавать абстрактные классы, которые являются только своеобразным шаблоном для создания производных классов. Создать объект абстрактного класса нельзя, это вызовет сообщение об ошибке.

Класс становится абстрактным, если в нем есть хотя бы одна чисто виртуальная функция, объявленная с присвоением нуля и не имеющая тела (определения) вообще:

```
virtual void printOn() = 0;
```

Все чисто виртуальные функции должны быть обязательно переопределены в производном классе. Целесообразно объявлять виртуальными функции ввода - вывода класса.

ОГРАНИЧЕНИЯ : виртуальными не могут быть конструкторы и статические функции-члены.

9.13. Стандартные библиотеки классов Borland C++.

9.13.1. Потоки ввода - вывода.

Стандартный ввод - вывод в C++ в отличие C осуществляется не через функции, а через перегруженные операторы.

Потоки C++ реализованы через целую иерархию классов iostream (iostream `class` library). Они дают возможность осуществлять ввод - вывод через унифицированную запись для разных типов вводимых и выводимых данных, переложив все детали на компилятор. Потоковые операторы записываются так :

поток_ввода >> типизированная_переменная;

поток_вывода << типизированная_переменная;

Объект потока всегда расположен в левой части выражения. Операторы << и >> указывают на направление потока данных от одного объекта к другому. С потоком ввода-вывода можно использовать любой встроенный тип данных. Пользовательские классы, если они были разработаны для поддержки ввода-вывода, также совместимы с потоками.

Пример стандартного ввода-вывода различных типов:

```
#INClude <iostream.h>
void main()
{
int a; char c; float f; double d;
cin >> a; cin >> c; cin >> f; cin >> d;
cout << a; cout << c; cout << f; cout << d;
cout << "\nПервая демонстрационная строка";
cout << "Вторая демонстрационная строка"
      << "\nСтрока сообщения"
      << "\nПродолжение";
}
```

C++ вместо файлов использует потоки.

Поток cin - стандартный символьный поток ввода, обычно связанный с клавиатурой. Он заменяет stdin, который использовался для тех же целей в C.

Поток cout подобен stdout и обычно связан с таким символьным выводным устройством, как дисплей персонального компьютера.

cerr заменяет stderr как стандартный поток вывода.

clog сходен с cerr, но обеспечивает буферизацию , тогда как cin, cout и cerr ее не обеспечивают.

Четыре стандартных потока cin, cout, crr и clog автоматически открываются до начала выполнения функции `main()` и закрываются после ее завершения.

Для работы с любым стандартным потоком в файл программы следует включить (`#INClude`) стандартный заголовочный файл iostream.h. Объект потока должен появляться в левой части операторов << или >>. Несколько потоковых операций, даже тогда, когда они относятся к объектам разных типов, могут быть объединены в одну строку. Комбинация символов >> называется оператором извлечения, поскольку они используются для получения символов из потока. Символы << называются оператором вставки, поскольку они используются для помещения символов в поток.

Предыдущий пример можно переписать так:

```
void main()
{ int a; char c; float f; double d;
 cin >> a >> c >> f >> d; //ввод всех данных
 cout << a << c << f << d;//вывод всех данных
}
```

ВВОД - ВЫВОД ПОЛЬЗОВАТЕЛЬСКИХ КЛАССОВ

Для работы с пользовательскими классами надо перегрузить операторы вставки в поток и извлечения из потока.

МАНИПУЛЯТОРЫ ДЛЯ ФОРМАТИРОВАНИЯ ВВОДА-ВЫВОДА.

Манипуляторы - это специальные функции, разработанные для модификации работы потока. Библиотека iostream имеет несколько готовых манипуляторов. Манипуляторы указывают , например , ширину поля, точность при вычислении с плавающей точкой и т. п.

Поскольку библиотека iostream поддерживает форматирование ввода-вывода через класс ios, встроенные манипуляторы должны использоваться только в классах , производных от ios.

Пример:

```
#INClude<iostream.h>
#INClude<iomanip.h>
void main()
{ cout<< hex << 20;}
```

На экране будет отображено число 14.

Для работы с манипулятором используется стандартный заголовочный файл iomanip.h. Манипуляторы действуют на ввод-вывод в поток до внесения новых изменений.

Список встроенных манипуляторов.	
DEC	Устанавливает 10-тичную систему счисления. Воздействует на int и long. Потоки используют основание 10 по умолчанию.
hex	Устанавливает 16-ричную систему счисления
oct	Устанавливает 8-ричную систему счисления
ws	Выбирает из потока ввода символы пропуска. Поток будет читаться до появления символа , отличного от пропуска , или до возникновения ошибки потока
endl	Вставляет в поток вывода символ новой строки и затем сбрасывает поток
ends	Вставляет '\0' в поток вывода
flus	Сбрасывает поток вывода

h	

setbase() устанавливает основание счисления к любому из четырех значений:

0	Основание по умолчанию. При выводе 10-тичное, при вводе - числа, начинающиеся с '0' , считаются 8-ричными, начинающиеся с '0x' , - 16-ричными. Во всех остальных случаях основание считается 10-тичным
8	Для ввода-вывода используется основание 8
10	Для ввода-вывода - 10
16	Для ввода-вывода - 16

Другие значения игнорируются. Библиотека iostream не поддерживает произвольных оснований , подобных 3, 12 и т. д. Если нужно представить значения по основанию, отличному от 8, 10 или 16, то соответствующее преобразование нужно выполнить явно.

resetiosflags(long) очищает один или более флагов форматирования в ios::x_flags

setiosflags(long) устанавливает один или более флагов форматирования в ios::x_flags

setfill(int) устанавливает символ - заполнитель.

Символ - заполнитель используется для заполнения поля тогда, когда ширина поля больше ширины выведенного значения. Заполнение не будет происходить , если пользователь не указал минимальной ширины поля с помощью манипулятора setw(int) или функции ios::width(int). По умолчанию символом-заполнителем является пробел. Заполнение будет происходить справа , слева, или как-то еще, в зависимости от значения битов ios::adjustfield, установленных обращением к ios::setf(long)

setprecision() устанавливает число цифр после 10-тичной точки в числах с плавающей точкой. Этот манипулятор действует только на потоке вывода setw(int width) Устанавливает ширину следующей вставляемой в поток вывода переменной. Если значение следующей переменной требует для записи меньше места , чем указано , то будет осуществляться заполнение символом-заполнителем , установленным манипулятором setfill(int). Ширина автоматически сбрасывается в 0 после каждой вставки в поток.

УСТАНОВКА И СБРОС ФЛАГОВ ФОРМАТИРОВАНИЯ

Встроенные флаги форматирования содержатся в базовом классе ios библиотеки iostream. Каждый флаг устанавливается

или сбрасывается с помощью встроенных манипуляторов. Манипуляторы resetioflags(long) и setioflags(long) позволяют сбросить или установить один или больше флагов.

//отображение результатов при разных системах счисления

```
v=100ж
cout << setiosflags(ios::showBase)
<<"\n" << v << " "
 << oct << v << " "
 << hex << v << endl;
```

В итоге получим следующий результат:

```
100 0144 0x64
```

ИСПОЛЬЗОВАНИЕ ФОРМАТИРУЮЩИХ МАНИПУЛЯТОРОВ

Список параметров манипуляторов setiosflags(long) и resetiosflags(long) : skipws left right internal DEC oct hex showBase showpoint uppercase showpos scientific fixed UNITbuf

ПОТОКИ И ФАЙЛЫ.

Описанный механизм ввода-вывода можно использовать и для файлов как в текстовом, так и в бинарных режимах. В библиотеке с файлами работает несколько классов, но чаще всего используется ifstream и ofstream. Класс ifsream поддерживает файлы, открытые для чтения, а ofstream - открытые для записи.

ТЕКСТОВЫЕ ФАЙЛЫ ДЛЯ ВВОДА.

Текстовый файл - это последовательность ASCII-символов, разделенная на строки. Каждая предыдущая строка заканчивается символом перехода к новой строке. Классы файловых потоков снабжены функциями для открытия и закрытия файлов, чтения и записи строк текста и выполнения множества других действий.

В качестве примера рассмотрим конструктор нашего класса vector для случая размещения его составляющих в текстовом файле в формате m d1 d2... dm, где m - размерность вектора, а di - его компоненты:

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(char *f)
{
 long i;
 ifstream fp=f;//пытаемся открыть файл
 if(!fp)//если не удалось
```

```
 { cerr<<"Не могу открыть файл "<<f<<"\n"; exit(0); }
 fp>>m;//вводим размерность
 if(m<=0)//проверка на корректность
 { cerr<<"Некорректный размер вектора\n"; exit(0);}
 vec=new YourOwnFloatType[m];//попытка выделения памяти
 if(vec==NULL)
 { cerr<<"Не хватает памяти\n"; exit(0); }
 for(i=0;i<m&&fp>>vec[i];i++);//считывание из файла
}
```

При использование файловых потоков в программу следует включать стандартный файл fstream.h. Чтобы отрыть файл на чтение, нужно определить объект ifstream и указать имя файла.

Извлечение символа новой строки осуществляется также как и любого другого символа. Посимвольное извлечение менее эффективно, чем обработка целой строки с помощью getline(. . .). Однако это может быть удобно, если перед использованием символы необходимо обработать. Извлечение из строки несимвольных типов прекращается при обнаружении символов пропуска или символов, недопустимых при обработки соответствующего типа данных.

Если файл открыт с помощью локальной переменной ifstream , то он автоматически будет закрыт по окончании работы функции, поэтому нет необходимости закрывать его явно. Любой файл закроется, когда связанный с ним объект выйдет из области видимости.

Большинство встроенных механизмов извлечения из потока и вставки в поток работает в текстовом режиме и не поддерживает бинарных операций. В текстовом режиме потоке ввода рассматриваются как последовательность символьных строк, где каждая строка заканчивается символом новой строки. Считывание данных из текстового потока в переменную приводит к автоматическому преобразованию данных в надлежащий формат. В текстовом режиме определенные ASCII-символы (в диапазоне от 0 до 0x1F) имеет специальное значение. При чтении из потока последовательность '\r ' '\n' преобразуется в '\n', а при записи в поток символ '\n' преобразуется в последовательность '\r' '\n'.

ПРОВЕРКА ПОТОКА НА ОШИБКИ

Проще всего так:-

```
if(!stream) //были ошибки
if(stream) //ошибок не было
```

Информацию об ошибке можно получить с помощью функции ios::rdstate().

```
if(stream.eof()) //конец файла
if(stream.good()) //ошибок не было
```

Чаще всего ios::clear() используется для сброса бита конца файла.

Если этот бит установлен, то для продолжения работы с файлом следует переместить указатель get в начало файла и сбросить бит ios::eofbit, например:

```
file.clear(); //сброс всех флагов ошибок
file.seekg(0); //перемещение указателя файла в начало
```

ВЫВОД ТЕКСТОВЫХ ФАЙЛОВ

При открытии файла в режиме вывода нужно точно знать, как будет записываться файл. По умолчанию существующий файл (в момент открытия) усекается до нуля, но такое поведение можно изменить.

Чтобы открыть файл для записи, не утратив при этом существующего в нем текста, нужно использовать режим добавления в конец:

```
ofstream copy_file("COPY", ios::app);
```

При открытии файла вывода можно потребовать, чтобы выполнялись и другие условия, например, открываемый файл еще не должен существовать, или, наоборот, он уже обязательно должен существовать. Эти условия устанавливаются с помощью флагов при создании объекта потока:

```
//убедиться, что файл еще не существует
ofstream copy_file("COPY", ios::noreplace);
//убедиться, что файл уже существует
ofstream copy_file("COPY", ios::nocreate);
```

В качестве примера рассмотрим операторную функцию записи вектора в текстовый файл с разделением чисел пробелами:

```
template <class YourOwnFloatType>
ostream &operator<<(ostream &os,vector<YourOwnFloatType> &x)
{
    for(long i=0;i<x.m;i++)
        os<<x[i]<<" ";//компоненты разделяем пробелами
    return os;
}
```

ЧТЕНИЕ БИНАРНЫХ ФАЙЛОВ

Открытие и закрытие их осуществляется точно так же , как и текстовых файлов, за исключением того, что в функцию

открытия передается флаг ios::binary. Возможно открытие текстового файла в бинарном режиме.

ЗАПИСЬ В БИНАРНЫЕ ФАЙЛЫ

Запись бинарных данных в файл можно осуществлять посимвольно и блоками. Для первого метода используется функция put(`char`), для второго - функция write(`char` *, `int`).

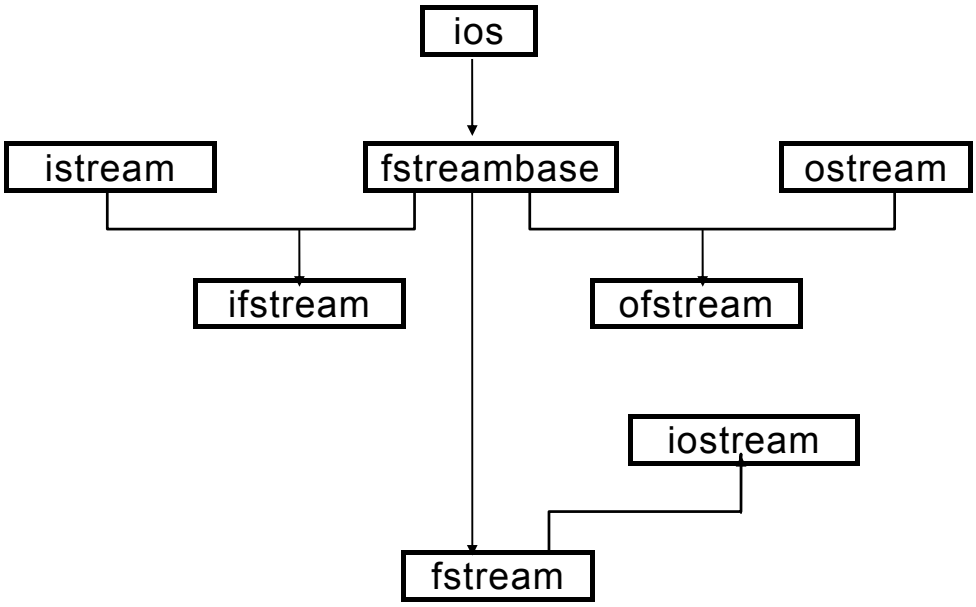
КОПИРОВАНИЕ ФАЙЛОВ

Используя потоковый ввод-вывод, легко осуществить операцию копирования содержимого одного файла в другой. Для этого вначале откроем эти файлы в двоичном режиме, один для чтения, другой - для записи, получим указатель на внутренний файловый буфер копируемого файла и используем переопределённый оператор вывода в поток указателя на объект типа "файловый буфер":

```
ifstream fin("shtcho",ios::in|ios::binary);
ofstream fout("nushtcho",ios::out|ios::binary);
fout<<fin.rdbuf();//возвращает указатель на filebuf
```

ИЕРАРХИЯ ПОТОКОВЫХ КЛАССОВ C++.

Приводим ее в заключение краткого обзора для ориентировки при использовании:



СОЗДАНИЕ СОБСТВЕННЫХ КЛАССОВ ДЛЯ РАБОТЫ С ВЕКТОРАМИ И МАТРИЦАМИ.

Для иллюстрации использования C++ и ООП приведем 2 класса с обширным сервисом в части обслуживания работы с векторами и матрицами.

```
#ifndef __MATRIX_H
#define __MATRIX_H
#ifndef __IOSTREAM_H
#INClude <iostream.h>
#endif
```

/*Файл включения для параметризованных классов векторов и матриц. */

```
template <class YourOwnFloatType>//подставьте свой тип
class vector
{ //приватные данные
  long m;//размерность (длина) вектора
  YourOwnFloatType *vec;//указатель на элементы вектора
 public://общедоступные данные и функции
  vector(char *);//загрузка вектора из файла
  vector();//создание пустого вектора единичной размерности
  vector(long);//создание пустого вектора заданной размерности
//создание вектора заданной размерности с данными из массива
  vector(long,YourOwnFloatType *);
  vector(vector &);//конструктор копирования
  ~vector();//деструктор
  friend vector operator+(vector &,vector &);//сложение
  friend vector operator-(vector &,vector &);//вычитание
  friend YourOwnFloatType operator*(vector &,vector &);
  friend vector operator*(YourOwnFloatType ,vector &);
  friend vector operator*(vector &,YourOwnFloatType );
  //вывод вектора в поток
  friend ostream &operator<<(ostream &,vector &);
  //ввод вектора из потока
  friend istream &operator>>(istream &,vector &);
  vector operator=(vector &);//присвоение
  vector operator-();//унарный минус
  //нормализация (определение направляющих косинусов)
  vector operator~();
  YourOwnFloatType operator!();//модуль вектора
  //проверка на равенство
  friend long operator==(vector &,vector &);
  //проверка на неравенство
  friend long operator!=(vector &,vector &);
  //индексирование элементов вектора
  YourOwnFloatType &operator[](long a);
  long getm() { return m; }//размерность вектора
};

template <class YourOwnFloatType>
```

```
class matrix
{ //приватные данные
  long m,n;//row,columns - размерность матрицы
  vector<YourOwnFloatType> *mtr;//указатель на вектор данных
  public://общедоступные члены
  //загрузка матрицы из файла в формате m n d11 d12 ... dmn
  matrix(char *);
  matrix();//пустая матрица 1x1
  matrix(long,long);//пустая матрица size1xsize2
  matrix(long,long,YourOwnFloatType *);//из массива
  matrix(matrix &);//конструктор копирования
  ~matrix();//деструктор
  friend matrix operator+(matrix &,matrix &);//сложение
  friend matrix operator-(matrix &,matrix &);//вычитание
  //умножение матриц
  friend matrix operator*(matrix &,matrix &);
  //умножение матрицы на число
  friend matrix operator*(YourOwnFloatType ,matrix &);
  friend matrix operator*(matrix &,YourOwnFloatType );
  //вывод матрицы в поток
  friend ostream &operator<<(ostream &,matrix &);
  //ввод матрицы из потока
  friend istream &operator>>(istream &,matrix &);
  //метод ортогонализации
  friend matrix SLAE_Orto(matrix &,matrix &);
  //метод Гаусса с выбором главного элемента
  friend matrix SLAE_Gauss(matrix &,matrix &);
  friend YourOwnFloatType det2(matrix &);//медленный Δ
  friend YourOwnFloatType det(matrix &);  //быстрый Δ
  //создание матрицы из имеющейся без заданных строки и столбца
  matrix minor(long,long);
  matrix operator=(matrix &);//присвоение
  //набор сокращённых операций
  matrix operator*=(matrix &x);//умножения,
  matrix operator+=(matrix &x);//сложения и
  matrix operator-=(matrix &x);//вычитания
  matrix operator^(long);//степень матрицы как операция
  //степень матрицы как дружественная функция
  friend matrix pow(matrix &,long);
  matrix operator~();   //транспонирование
  matrix operator!(); //"классическое" обращение матрицы
  matrix operator*();   //численное обращение матрицы
  //быстрый детерминант унарная операция
  YourOwnFloatType operator&();
  //быстрый детерминант-оператор преобразования к типу-параметру
  operator YourOwnFloatType();
  matrix operator-();//унарный минус
```

```
  //проверка на равенство
  friend long operator==(matrix &,matrix &);
  //проверка на неравенство
  friend long operator!=(matrix &,matrix &);
  //индексирование матрицы
  vector<YourOwnFloatType> &operator[](long a);
  friend long sign(YourOwnFloatType x);//получение знака
  //Вычисление собственных векторов и значений методом Якоби
  friend void Reigen(matrix&, vector<YourOwnFloatType>&, matrix&);
  long getm() { return m; }//get rows
  long getn() { return n; }//get columns
};

#endif
```

Теперь файл определений всех функций:

```
#INClude <fstream.h>
#INClude <stdlib.h>
#INClude <math.h>
#INClude <values.h>
#INClude «matrix.h»
```

/*Основными объектами линейной алгебры являются вектора и матрицы, которые средствами языка C++ достаточно легко превратить в соответствующие классы, сохранив при этом естественное представление матрицы как упорядоченного кортежа арифметических векторов, а вектора - как упорядоченного кортежа объектов любой природы. */

```
//индикатор ошибки – максимально возможное машинное число
const long double MAX_LONGDOUBLE=1.7976931348e308;
```

/* Создавать вектор можно по-разному. Например, если он находится на внешнем устройстве в формате m d_1 d_2 . . . d_m, где m - размерность вектора, а d_i - его компоненты, то имеем следующий конструктор:* /

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(char *f)
{
  long i;

  ifstream fp=f;//пытаемся открыть файл
  if(!fp)//если не удалось
  {cerr<<"Не могу открыть файл "<<f<<"\n"; exit(0);}
  fp>>m;//вводим размерность
  if(m<=0)//проверка на корректность
```

```
  { cerr<<"Некорректный размер вектора\n"; exit(0);}
  vec=new YourOwnFloatType[m];//попытка выделения памяти
  if(vec==NULL)
  {cerr<<"Не хватает памяти\n"; exit(0); }
  for(i=0;i<m&&fp>>vec[i];i++);//считывание из файла
}
```

/* В случае, когда нам известна лишь размерность вектора, но неизвестны его составляющие, предполагаем, что они нулевые: */

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long a):m(a)
{
  long i;
  if(m<=0)//проверка размерности
  { cerr<<"Некорректный размер вектора\n"; exit(0);}
  vec=new YourOwnFloatType[m];//попытка выделения памяти
  if(vec==NULL)//если не удалось выделить память
  { cerr<<"Не хватает памяти\n"; exit(0);}
  for(i=0;i<m;vec[i++]=0);//обнуление
}
```

/* Наконец, нам могут быть известны как размерность, так и компоненты вектора :*/

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long                a,
YourOwnFloatType *v) : m(a)
{
  long i;
  if(m<=0)//проверка размерности
  { cerr<<"Некорректный размер вектора\n"; exit(0);}
  vec=new YourOwnFloatType[m];//выделение памяти
  if(vec==NULL)//если не выделили
  { cerr<<"Не хватает памяти\n"; exit(0);}
  for(i=0;i<m;i++)
    vec[i]=v[i];//копирование из внешнего массива в
вектор
}
```

/* Есть ещё один случай, когда мы ничего не можем сказать о размерности и компонентах вектора - при создании массива векторов, то есть матрицы, когда для оператора new требуется конструктор без параметров или когда размер вектора заранее неизвестен. */

```
template <class YourOwnFloatType>
```

```
vector<YourOwnFloatType>::vector():m(1)
{
  vec=new YourOwnFloatType[m];
  if(vec==NULL)
  { cout<<"Не хватает памяти\n"; exit(0);}
  *vec=0;//обнуляем единственный имеющийся элемент
}
```

/* В реальных расчетах могут использоваться вектора больших размерностей, поэтому размещаются они в свободной памяти компьютера, а когда необходимость в них отпадает - уничтожаются. */

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::~vector()
{
  delete []vec;//уничтожение динамического массива
}
```

/*Необходимость в индексации вектора есть в двух случаях:

- при получении составляющей вектора по её номеру и
- при изменении не всего вектора, а только 1-й его составляющей.

При этом, конечно, следует учитывать возможность ошибочного задания номера составляющей : допустимый диапазон значений [0,m). */

```
template <class YourOwnFloatType>
YourOwnFloatType
&vector<YourOwnFloatType>::operator[](long a)
{
  static YourOwnFloatType error=MAX_LONGDOUBLE;
  if(a>=0&&a<m)  return vec[a];  //если всё ОК
  else {//при выходе за пределы
    cerr<<"Индекс "<<a<<" вне диапазона вектора\n";
  return error; }
}
```

/* Создавая вектор, можно попутно инициализировать его данными из уже существующего: */

```
template <class YourOwnFloatType> vector<YourOwnFloatType>
    ::vector(vector<YourOwnFloatType> &ex) : m(ex.m)
{ vec=new YourOwnFloatType[m];
  if(vec==NULL)
  { cout<<"Не хватает памяти\n";  exit(0); }
  for(long i=0;i<m;i++)
  //при копировании ex используется уже индексация
```

```
    vec[i]=ex[i];
}
```

/*Сложение векторов является алгебраической операцией только тогда, когда вектора одинаковой размерности. Результатом сложения является вектор той же размерности, что и исходные, компонентами которого является сумма соответствующих компонент исходных векторов. */

```
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator+
(vector<YourOwnFloatType> &f,vector<YourOwnFloatType> &s)
{
  if(f.m!=s.m)//проверка на равенство размерностей
  { cout<<"Длины векторов не совпадают \n"; exit(0);}
  vector<YourOwnFloatType> temp(f.m);//создаём временный вектор
  //здесь работают операции индексирования для всех трёх векторов
  for(long i=0;i<f.m;i++)
    temp[i]=f[i]+s[i];
  return temp;
}
```

/* Введём несколько вспомогательных унарных операций:
- "минус": */

```
template <class YourOwnFloatType> vector<YourOwnFloatType>
vector<YourOwnFloatType>::operator-()
{
  vector<YourOwnFloatType> temp(m);//создаём временный вектор
```

/*Если this - это указатель на текущий объект векторного класса, то *this - это сам текущий объект класса vector, то есть тот, с которым мы сейчас работаем. А к любому векторному объекту мы можем применить операцию индексирования */

```
  for(long i=0;i<m;i++)  temp[i]=-(*this)[i];
  return temp;
}
```

/* нормирование вектора */

```
template <class YourOwnFloatType> vector<YourOwnFloatType>
vector<YourOwnFloatType>::operator~()
{
  vector<YourOwnFloatType> temp(m);
  //скалярное произведение текущего объекта на самого себя
  YourOwnFloatType scalp=(*this)*(*this);
  for(long i=0;i<m;i++)
    temp[i]=(*this)[i]/sqrt(scalp);//направляющие косинусы
  return temp;
}
```

/* Модуль вектора - бинарное отношение, которое ми определим не совсем стандартно, а именно как квадратный корень скалярного произведения вектора на самого себя: */

```
template <class YourOwnFloatType> inline
YourOwnFloatType vector<YourOwnFloatType>::operator!()
{  return sqrt((*this)*(*this));}
```

/* Операция, которую алгебраической назвать нельзя - это, скорее, пример очень распространённого тернарного отношения "скалярное произведение двух векторов": */

```
template <class YourOwnFloatType> YourOwnFloatType
operator*(vector<YourOwnFloatType> &f, vector<YourOwnFloatType>
&s)
{
  if(f.m!=s.m)
  {
     cout<<"Умножение векторов с несовпадающими размерами
невозможно!\n";
    exit(0);
  }
  YourOwnFloatType temp=0;
  for(long i=0;i<f.m;i++)
    temp+=f[i]*s[i];//суммируем произведения составляющих векторов
  return temp;
}
```

/*Важным при подсчётах является тернарное отношение "умножение числа на вектор": */

```
template <class YourOwnFloatType> vector<YourOwnFloatType>
operator*(YourOwnFloatType ld,vector<YourOwnFloatType> &v)
{
  vector<YourOwnFloatType> temp=v;
  for(long i=0;i<v.getm();i++)
    temp[i]=temp[i]*ld;//скорее, это даже удлинение вектора
  return temp;
}
```

/* Тернарное отношение "умножение числа на вектор" является коммутативным, поэтому через него можно определить и умножение вектора на число: */

```
template <class YourOwnFloatType>
inline vector<YourOwnFloatType> operator*
    (vector<YourOwnFloatType> &v,YourOwnFloatType ld)
{
```

```
  return ld*v;//очень просто - вызвали другую функцию
}
```

/*Имея определённые бинарную операцию сложения векторов и унарную получения вектора, противоположного к данному, можно на векторном языке, не обращаясь к компонентам векторов, определить операцию вычитания: */

```
template <class YourOwnFloatType> inline vector<YourOwnFloatType>
operator-(vector<YourOwnFloatType>  &f,vector<YourOwnFloatType>
&s)
{
  return f+(-s);
}
```

/* При переписывании одного вектора в другой возможны два случая:

1. если размерность обоих векторов совпадает, то просто заменяем составляющие первого вектора компонентами второго;
2. в противном случае безжалостно уничтожаем первый вектор и создаём снова, используя второй как строительный материал. */

```
template <class YourOwnFloatType> vector<YourOwnFloatType>
vector<YourOwnFloatType>::operator=(vector<YourOwnFloatType> &x)
{
  if(m!=x.m)
  {
  delete []vec; m=x.m; vec=new YourOwnFloatType[m];
    if(vec==NULL)
    { cout<<"Не хватает памяти\n";  exit(0); }
  }
  for(long i=0;i<m;i++)   vec[i]=x[i];
```

/*присваивание - это бинарная операция, первым параметром которой является объект, которому присваивают, вторым - объект, который присваивают. При этом первый объект, в отличие от всех остальных бинарных операций, меняется, и он же возвращается в качестве результата (это бывает необходимым для операций вида a=b=c;)*/

```
  return *this;
}
```

/* Сравнение векторов является тернарным отношением, результатом которого является число нуль, если вектора не равны и единица в противном случае. Два вектора будем считать равными, если они имеют одинаковые длины и их соответствующие составляющие совпадают: */

```
template <class YourOwnFloatType>
long operator==(vector<YourOwnFloatType> &f,
vector<YourOwnFloatType> &s)
{
  if(f.m!=s.m) return 0;//при несовпадении размерностей
  for(long i=0;i<f.m;i++)
    if(f[i]!=s[i]) return 0;//если хоть один элемент не совпал
  return 1;
}
```

/*Неравенство векторов определим через равенство и операцию отрицания: */

```
template <class YourOwnFloatType> inline long operator!=
(vector<YourOwnFloatType> &f,vector<YourOwnFloatType> &s)
{
  return !(f==s);//логично
}
```

/*Мощный I/O-механизм C++ позволяет в естественной форме выводить (вводить) вектора на любое устройство отображения информации: вывод */

```
template <class YourOwnFloatType> ostream
&operator<<(ostream &os,vector<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)
    os<<x[i]<<" ";//разделяем пробелами
  return os;
}

    /* ввод из потока */

template <class YourOwnFloatType> istream
&operator>>(istream &is,vector<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)  is>>x[i];
  return is;//принимаем и возвращаем ссылку на поток ввода
}
```

/* Напомним, что матрица - это тоже вектор, элементами которого являются не числовые объекты, а арифметические вектора. В связи с этим между векторным и матричным классами есть много общего, однако иногда можно наблюдать и существенные отличия.

Возьмём, скажем, классический файловый метод хранения данных. Если для вектора достаточно было указать одно служебное число - его длину (размерность), то для матрицы их

требуется уже два, причём первое из этих чисел будет определять количество векторов в матрице, а второе - размерность каждого из этих векторов. В файловых операциях с векторами нам поможет вышеопределённая операция введения вектора с некоторого устройства: */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>::matrix(char *f)
{
  long i;
  ifstream fp=f;
  if(!fp)
  { cerr<<"Не могу открыть файл "<<f<<"\n"; exit(0);}
  fp>>m>>n;
  if(m<=0||n<=0)
  { cerr<<"Bad matrix size\n"; exit(0);}
  //здесь работает конструктор без параметров
  mtr=new vector<YourOwnFloatType>[m];
  if(mtr==NULL)
  { cerr<<"Не хватает памяти\n"; exit(0); }
  //и только здесь вектор расширяется до нужной размерности
  for(i=0;i<m;i++)
    mtr[i]=vector<YourOwnFloatType>(n);
  //при вводе используем перегруженную операцию класса vector
  for(i=0;i<m&&fp>>mtr[i];i++);
}
```

/* зная только размеры матрицы, мы можем считать её нулевым элементом данного размера и сконструировать соответствующим образом: */

```
template <class YourOwnFloatType> matrix<YourOwnFloatType>::
matrix(long a,long b):m(a),n(b)
{
  long i;
  if(m<=0||n<=0)
  { cerr<<"Bad matrix size\n"; exit(0); }
  mtr=new vector<YourOwnFloatType>[m];
  if(mtr==NULL)
  { cerr<<"Не хватает памяти\n"; exit(0); }
  for(i=0;i<m;i++)
    mtr[i]=vector<YourOwnFloatType>(n);
}
```

/*Получив о матрице все возможные сведения, имеет смысл, сконструировав её, инициализировать соответствующими элементами: */

```
template <class YourOwnFloatType>
```

```
matrix<YourOwnFloatType>::
matrix(long a,long b,YourOwnFloatType *mt):m(a),n(b)
{
  if(m<=0||n<=0)
  { cerr<<"Bad matrix size\n"; exit(0); }
  mtr=new vector<YourOwnFloatType>[m];
  if(mtr==NULL)
  { cerr<<"Не хватает памяти\n";  exit(0); }
  for(long i=0,l=0;i<m;i++) mtr[i]=vector<YourOwnFloatType>(n);
  for(i=0;i<m;i++)
  //сконструировав, копируем данные из массива в матрицу
    for(long j=0;j<n;j++)    mtr[i][j]=mt[l++];
}
```

/* Обратный случай (когда мы не знаем ни размеров, ни элементов матричных векторов) решается достаточно просто - созданием матрицы из одного единственного вектора длиною в один элемент */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>::matrix():m(1),n(1)
{ mtr=new vector<YourOwnFloatType>[1];//а обнулится он вектором
  if(mtr==NULL)
  {cout<<"Не хватает памяти\n"; exit(0); }
}
```

/* уничтожение матрицы автоматически уничтожает все её вектора: */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>::~matrix()
{
  delete []mtr;//вызов деструкторов для всех векторов
}
```

/*Как и вектора, матрицы тоже бывает необходимым проиндексировать; результатом этого действия, естественно, будет соответствующий вектор: */

```
template <class YourOwnFloatType>
vector<YourOwnFloatType> &matrix<YourOwnFloatType>::
                                    operator[](long a)
{
  static vector<YourOwnFloatType> error;
  error[0]=MAX_LONGDOUBLE;
  if(a>=0&&a<m)
    return mtr[a];//а дальше можно индексировать вектор
  else
  {
    cerr<<"Индекс "<<a<<" вне матрицы \n";
```

```
    return error;
  }
}
```

/*Если в памяти ЭВМ уже есть какая-то матрица, с неё можно снять слепок: */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>::matrix
    (matrix<YourOwnFloatType> &ex) : m(ex.m), n(ex.n)
{
  mtr=new vector<YourOwnFloatType>[m];
  if(mtr==NULL)
  { cout<<"Не хватает памяти\n"; exit(0); }
  //индексирование и присваивание векторов
  for(long i=0;i<m;i++) mtr[i]=ex[i];
}
```

/* Сложение матриц одинаковой размерности сводится к сложению соответствующих векторов: */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType> operator+
(matrix<YourOwnFloatType> &f,matrix<YourOwnFloatType> &s)
{
  if(f.m!=s.m||f.n!=s.n)
  { cout<<"Размеры этих матриц должны совпадать\n"; exit(0);}
  matrix<YourOwnFloatType> temp(f.m,f.n);
  for(long i=0;i<f.m;i++)  temp[i]=f[i]+s[i];
  return temp;
}
```

```
 /*  Как и для векторов, определим унарный "минус" :*/
```

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator-()
{
  matrix<YourOwnFloatType> temp(m,n);
  for(long i=0;i<m;i++)    temp[i]=-(*this)[i];
  return temp;
}
```

/* В отличие от скалярного произведения векторов, умножение матриц является бинарной алгебраической операцией, однако только для отдельных видов матриц, к тому же эта операция не является коммутативной! */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType> operator*
```

```
 (matrix<YourOwnFloatType> &f,matrix<YourOwnFloatType> &s)
{
  if(f.n!=s.m)
  {
    cout<<"Умножение матриц с заданными размерами невозможно!\n";
    exit(0);
  }
  matrix<YourOwnFloatType> temp(f.m,s.n);
  for(long j=0;j<s.n;j++)
    for(long i=0;i<f.m;i++)
      for(long k=0;k<f.n;k++)
        temp[i][j]+=f[i][k]*s[k][j];
  return temp;
}
```

/*Полезной является унарная операция увеличения матрицы в некоторое число раз: */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType> operator*
   (matrix<YourOwnFloatType> &f,YourOwnFloatType s)
{
  matrix<YourOwnFloatType> temp=f;
  for(long i=0;i<f.m;i++)  temp[i]=temp[i]*s;
  return temp;
}
```

/*Оператор возведения матрицы в целую степень можно определить так:

- любая матрица в нулевой степени является единичной,
- положительная степень определяется через произведение,
- отрицательная - через обращение матрицы и произведение.

Последний случай можно красиво реализовать с использованием рекурсии, как, впрочем, и предыдущий. */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType> matrix<YourOwnFloatType>::
                                    operator^(long l)
{
  matrix<YourOwnFloatType> temp(getm(),getn());
  if(getm()!=getn())
  {
    cerr<<"Для неквадратных матриц степень не определена \n";
    exit(0);
  }
  if(l==0)
  {
```

```
    for(long i=0;i<getm();i++)    temp[i][i]=1;
    return temp;
  }
  if(l>0)
  { temp=(*this); for(long i=1;i<l;i++) temp*=(*this);
    return temp;
  }
  else  return (*(*this))^(-l);
}
```

/* Степень матрицы в виде функции */

```
template <class YourOwnFloatType>
inline matrix<YourOwnFloatType>
pow(matrix<YourOwnFloatType> &x,long l)
{  return x^l;}
```

/*Умножение числа на матрицу реализуем через уже имеющуюся операцию умножения матрицы на число для сохранения коммутативности */

```
template <class YourOwnFloatType>
inline matrix<YourOwnFloatType> operator*
     (YourOwnFloatType f,matrix<YourOwnFloatType> &s)
{  return s*f;}
```

/*Вычитание традиционно реализуем через сложение и унарный минус: */

```
template <class YourOwnFloatType>
inline matrix<YourOwnFloatType> operator-
 (matrix<YourOwnFloatType> &f,matrix<YourOwnFloatType> &s)
{  return f+(-s);}
```

/*При присвоении содержимое матрицы x переписывается в текущую сразу только тогда, когда их размеры совпадают. В противном случае приходится уничтожать текущую матрицу, заново её создавать с новыми размерами и лишь тогда производить повекторное копирование */

```
template <class YourOwnFloatType> matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator=(matrix<YourOwnFloatType> &x)
{
  if(m!=x.m||n!=x.n)
  {
    delete []mtr;m=x.m,n=x.n; mtr=new
vector<YourOwnFloatType>[m];
    if(mtr==NULL)
    { cout<<"Не хватает памяти\n";  exit(0); }
  }
```

```
  for(long i=0;i<m;i++)  mtr[i]=x[i];
  return *this;//возвращение себя
}

     /*  Набор сокращённых операций: умножение */

template <class YourOwnFloatType> matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator*=(matrix<YourOwnFloatType>
&x)
{  return (*this)=(*this)*x;}

     /* сложение */
template <class YourOwnFloatType> inline matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator+=(matrix<YourOwnFloatType>
&x)
{ return (*this)=(*this)+x;}

     /* вычитание */
template <class YourOwnFloatType> inline matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator-=(matrix<YourOwnFloatType>
&x)
{//используем только что сделанное сокращённое сложение
  return (*this)+=-x;
}

/*Очень часто бывает нужна унарная операция транспонирования: */

template              <class              YourOwnFloatType>
matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator~()
{
  matrix<YourOwnFloatType> temp(n,m);
  for(long j=0;j<n;j++)
    for(long i=0;i<m;i++)    temp[j][i]=mtr[i][j];
  return temp;//переставлены столбцы и строки
}

/*Сравнивая матрицы, мы сначала учитываем, одинаковой ли
они размерности, а потом в случае необходимости сравниваем
вектора: */

template <class YourOwnFloatType> long
operator==(matrix<YourOwnFloatType> &f, matrix<YourOwnFloatType>
&s)
{
  if(f.m!=s.m||f.n!=s.n)    return 0;
  for(long i=0;i<f.m;i++)
    if(f[i]!=s[i])          return 0;
```

```
  return 1;
}
```

/* Неравенство - оно и есть НЕ равенство */

```
template <class YourOwnFloatType> inline long operator!=
(matrix<YourOwnFloatType> &f,matrix<YourOwnFloatType> &s)
{
  return !(f==s);//!operator==(f,s);
}
```

/*Результат иногда хочется посмотреть. Например, так: */

```
template <class YourOwnFloatType> ostream
&operator<<(ostream &os,matrix<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)   os<<x[i]<<"\n";
  return os;
}
```

/*Ввод матрицы из потока целиком перекладываем на векторный класс, используя его метод для ввода */

```
template <class YourOwnFloatType> istream
&operator>>(istream &is,matrix<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)   is>>x[i];
  return is;
}
```

/*В матричной форме систему линейных алгебраических уравнений (СЛАУ) можно представить в виде

A*X=B,

где A - матрица коэффициентов при неизвестных,
X - вектор-столбец этих самих неизвестных, а
B - вектор-столбец свободных членов, взятых с обратными знаками.

Если система имеет решение, то оно будет таким:

A^(-1)*A*X=A^(-1)*B
(A^(-1)*A)*X=A^(-1)*B
E*X=A^(-1)*B
X=A^(-1)*B,

Существенным моментом является выбор метода решения. Для небольших систем (<200) можно использовать прямые методы (например, метод Гаусса); для реальных расчётов ми рекомендуем метод ортогонализации, в основе

которого лежат ортогональные преобразования матриц, которые вы осуществляли ранее */

```
template <class YourOwnFloatType> matrix<YourOwnFloatType> SLAE_Orto (matrix<YourOwnFloatType> &f,matrix<YourOwnFloatType> &s)
{
  matrix<YourOwnFloatType> mtr2(f.m+1,f.m+1),res(f.m,1);
  //формируем матрицу из двух для решения СЛАУ
  for(long i=0;i<f.m;i++)
    for(long j=0;j<f.n;j++)   mtr2[i][j]=f[i][j];
    for(i=0;i<f.m;i++)//заносим в последнюю строку 0 0 ... 0 1
    mtr2[i][f.m]=-s[i][0];
    mtr2[f.m][f.m]=1;

  mtr2[0]=~mtr2[0];//нормируем нулевую строку
  //для улучшения повторяем процесс ортонормирования 3 раза
  for(long k=0;k<3;k++)
  {
    for(long l=1;l<f.m+1;l++)
    {
      for(i=0;i<l;i++)
      { //скалярное произведение
        YourOwnFloatType p=mtr2[l]*mtr2[i];
        for(long j=0;j<(f.m+1);j++)
          mtr2[l][j]-=p*mtr2[i][j];
      }
      mtr2[l]=~mtr2[l];//norm(l);
    }
  }
  for(i=0;i<f.m;i++)//переписываем результат
    res[i][0]=mtr2[f.m][i]/mtr2[f.m][f.m];
  return res;
}
```

/*Уже знакомый вам метод Гаусса с выбором главного элемента */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType> SLAE_Gauss
  (matrix<YourOwnFloatType> &f,matrix<YourOwnFloatType> &s)
{
  long i,j,k,num;

  matrix<YourOwnFloatType> mtr2(f.m,f.m+1),res(f.m,1);
  //формируем матрицу из двух для решения СЛАУ
  for(i=0;i<f.m;i++)
    for(j=0;j<f.n;j++)    mtr2[i][j]=f[i][j];
```

```
  for(i=0;i<f.m;i++)     mtr2[i][f.m]=s[i][0];
  //выбор главного элемента
  for(i=0;i<f.m;i++)//col
  {
    YourOwnFloatType max=mtr2[0][i];
    for(j=i,num=i;j<f.m;j++)    //row
      if(fabs(mtr2[j][i])>max)
        max=fabs(mtr2[j][i]),num=j;
    if(num!=i)
      for(k=0;k<f.m+1;k++)//num & i
      {
        YourOwnFloatType temp=mtr2[num][k];
        mtr2[num][k]=mtr2[i][k];
        mtr2[i][k]=temp;
      }
  }
  for(i=0;i<f.m;i++)
    if(!mtr2[i][i])
    {
       cerr<<"Решение системы при возможно вырожденной
    матрице прямыми методами невозможно\n"; exit(0);
    }
  //Прямой ход Гаусса
  for(i=0;i<f.m;i++)//
  {
    double sw=mtr2[i][i];
    for(j=0;j<f.m+1;j++)
      mtr2[i][j]/=sw;
    for(k=i+1;k<f.m;k++)
    {
      double c=mtr2[k][i];
      for(j=0;j<f.m+1;j++)
        mtr2[k][j]-=mtr2[i][j]*c;
    }
  }
  //Обратный ход Гаусса
  for(i=f.m-2;i>=0;i--)//row
  {
    double s=0;
    for(j=i+1;j<f.m;j++) //col
      s+=mtr2[i][j]*mtr2[j][f.m];
    mtr2[i][f.m]-=s;
  }
  //переписываем результат
  for(i=0;i<f.m;i++)
    res[i][0]=mtr2[i][f.m];
  return res;
}
```

/*Для обращения матрицы и нахождения детерминанта классическими методами нужна функция получения минора матрицы для данного её элемента */

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::minor(long a, long b)
{
  matrix<YourOwnFloatType> temp(getm()-1,getn()-1);
  for(long i1=0,i2=0;i1<getm();i1++)
    if(i1!=a)
    {
      for(long j1=0,j2=0;j1<getn();j1++)
        if(j1!=b)
          temp[i2][j2]=(*this)[i1][j1],
          j2++; i2++;
    }
  return temp;
}
```

/* Классический детерминант - вычисление разложением по одной из строк */

```
template <class YourOwnFloatType>
YourOwnFloatType det2(matrix<YourOwnFloatType> &x)
{
  if(x.getm()!=x.getn())
  {
    cerr<<"Для неквадратных матриц детерминант не определён"<<endl;
    exit(0);
  }
  if(x.getm()==1)   return x[0][0];
  if(x.getm()==2) return x[0][0]*x[1][1]-x[0][1]*x[1][0];
  YourOwnFloatType result=0;
  for(long i=0;i<x.getm();i++)
    result+=pow(-1,1+i)*det(x.minor(1,i))*x[1][i];
    return result;
}
```

/* Медленная операция обращения квадратной матрицы */

```
template <class YourOwnFloatType> matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator!()
{
  if(getm()!=getn())
  { cerr<<"Попытка обращения неквадратной матрицы "<<endl;  exit(0); }
  matrix<YourOwnFloatType> temp=*this;
  YourOwnFloatType _det=det(*this);
  if(_det==0)
```

```
  { cerr<<"Особенная матрица не имеет обратной\n";  exit(0);}
  for(long i=0;i<getm();i++)
    for(long j=0;j<getn();j++)
      temp[i][j]=pow(-1,2+i+j)*det(minor(j,i))/_det;
  return temp;
}

/* Численное нахождение детерминанта методом Гаусса */

template <class YourOwnFloatType> YourOwnFloatType
det(matrix<YourOwnFloatType> &y) //быстрый Δ
{
  YourOwnFloatType sw,c,det,max;
  long i,j,k,how,num;

  matrix<YourOwnFloatType> x=y;
  if(x.getm()!=x.getn())
  {
    cerr<<"Детерминант определён только для квадратных
матриц "<<endl;  exit(0);
  }
  if(x.getm()==1)  return x[0][0];
  if(x.getm()==2)  return x[0][0]*x[1][1]-x[0][1]*x[1][0];
  for(how=i=0;i<x.getm();i++)//col
  {
    max=x[0][i];
    for(j=i,num=i;j<x.getm();j++)    //row
      if(fabs(x[j][i])>max)
      { max=fabs(x[j][i]);  num=j; }
    if(num!=i)
    {
      for(k=0;k<x.getm();k++)//num & i{
     YourOwnFloatType temp=x[num][k];x[num][k]=x[i][k];x[i][k]=temp;}
      how++;
    }
  }
  for(i=0;i<x.getm();i++)//
  {
    for(sw=x[i][i],j=i+1;j<x.getm();j++)
      x[i][j]/=sw;
    for(k=i+1;k<x.getm();k++)
      for(c=x[k][i],j=0;j<x.getm();j++)
        x[k][j]-=x[i][j]*c;
  }
  for(det=1,i=0;i<x.getm();i++) det*=x[i][i];
  det*=pow(-1,how);
  return det;
}

    /*  Быстрое обращение матрицы */
```

```
template <class YourOwnFloatType>
matrix<YourOwnFloatType>
matrix<YourOwnFloatType>::operator*()//быстрое обращение
{
  long i,j,k,l;
  YourOwnFloatType v,maxabs,s,tsr;
  matrix<YourOwnFloatType> temp(getm(),2*getn());

  if(getm()!=getn())
  { cerr<<"Попытка обращения неквадратной матрицы "<<endl;
exit(0);}
  if(det(*this)==0)
  {cerr<<"Особенная матрица не имеет обратной\n";exit(0);}
  for(i=0;i<getm();i++)
  {
    for(j=0;j<getn();j++)  temp[i][j]=(*this)[i][j];
    for(j=getm();j<2*getm();j++)
      temp[i][j]=(j==i+getm()) ? 1 : 0;
  }
  for(i=0;i<getm();i++)
  {
    for(maxabs=fabs(temp[i][i]),k=i,l=i+1;l<getm();l++)
      if(fabs(temp[l][i])>maxabs)
      maxabs=fabs(temp[l][i]), k=l;
       if(k!=i)
        for(j=i;j<2*getm();j++)
        v=temp[i][j], temp[i][j]=temp[k][j], temp[k][j]=v;
  }
  for(i=0;i<getm();i++)
  {
    for(s=temp[i][i],j=i+1;j<2*getm();j++)   temp[i][j]/=s;
    for(j=i+1;j<getm();j++)
    {
      for(tsr=temp[j][i],k=i+1;k<2*getm();k++)
        temp[j][k]-=temp[i][k]*tsr;
    }
  }
  for(k=getm();k<2*getm();k++)
    for(i=getm()-1;i>=0;i--)
    {
      for(tsr=temp[i][k],j=i+1;j<getm();j++)
        tsr-=temp[j][k]*temp[i][j];
      temp[i][k]=tsr;
    }
  matrix<YourOwnFloatType> result=(*this);
  for(i=0;i<getm();i++)
    for(j=getm();j<2*getm();j++)
      result[i][j-getm()]=temp[i][j];
  return result;
```

```
}
```

/* Быстрый детерминант как операция */

```
template <class YourOwnFloatType> inline YourOwnFloatType
matrix<YourOwnFloatType>::operator&()
{  return det(*this);}
```

/* Детерминант рассматривается как оператор преобразования матрицы в число */

```
template <class YourOwnFloatType> inline
matrix<YourOwnFloatType>::operator YourOwnFloatType()
{return det(*this);}
```

/* Параметризованная функция, определяющая знак числа */

```
template <class YourOwnFloatType> inline long
sign(YourOwnFloatType x){  return (x<0) ? -1 : 1;}
```

/*Собственные векторы и собственные значения симметричной матрицы */

```
template <class YourOwnFloatType> void Reigen (matrix
<YourOwnFloatType> &xxx,
 vector<YourOwnFloatType> &resultv, matrix<YourOwnFloatType>
&resultm)
{
  YourOwnFloatType c,n1,n2,t,m,f,g,u,v1,v2,v3,e=1e-8;
  long j,q,i,p,n=xxx.getm(),d;

  resultm=xxx;
  matrix<YourOwnFloatType> a=xxx;
  u=0;
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
    {
      resultm[i][j]=resultm[j][i]=0;
      if(i==j)     resultm[i][i]=1;
      else         u+=a[i][j]*a[i][j];
    }
  if(u!=0)
 { n1=t=sqrt(u);  n2=e/n*n1;  d=0;
    do
    {  t/=n;
      do
      {
        for(q=1;q<n;q++)
          for(p=0;p<q;p++)
```

```
            if(fabs(a[p][q])>t)
            {
            d=1,v1=a[p][p],v2=a[p][q],v3=a[q][q],m=(v1-v3)*(v1-
v3)/4;
              g=m==0 ? -1 : sign(v3-v1)*v2/sqrt(v2*v2+m);
              f=g/sqrt(2*(1+sqrt(1-g*g))); c=sqrt(1-f*f);
              for(i=0;i<n;i++)
              {
                if(i!=p&&i!=q)
                {
                  u=a[i][p],m=a[i][q],a[q][i]=u*f+m*c,a[i][q]=a[q][i];
                  a[p][i]=u*c-m*f,a[i][p]=a[p][i];
                }
                u=resultm[i][p]; m=resultm[i][q];
                resultm[i][q]=u*f+m*c;resultm[i][p]=u*c-m*f;
              }
             m=f*f; g=c*c; u=f*c;
             a[p][p]=v1*g+v3*m-2*v2*u; a[q][q]=v1*m+v3*g+2*v2*u;
             a[p][q]=(v1-v3)*u+v2*(g-m); a[q][p]=a[p][q];
            }
            if(!d)
              d=1;
        }while(!d);
      }while(t>=n2);
    }
    else
      resultm=a;
    resultv=vector<YourOwnFloatType>(n);
    for(i=0;i<n;i++)
      resultv[i]=a[i][i];
}
```

9.13.2. Библиотеки контейнерных классов C++.

9.13.2.1. Общие сведения.

Одной из наиболее трудоемких работ в прикладном программировании, помимо создания пользовательских интерфейсов, является обработка разнообразных совокупностей элементов или групп - массивов, списков и других, более сложных структур, называемых обобщенно контейнерами. Borland C++, начиная с версии 2, содержит библиотеки контейнерных классов, использование которых существенно облегчает работу с групповыми объектами.

Таких библиотек две - первая базируется на абстрактном классе Object и все ее классы должны быть его производными. Основное ее неудобство именно в этом, так как даже попытка

создать контейнер со стандартными типами `int` или `double` вынуждает порождать промежуточные производные от Object классы вроде `class int`:Object и `class Double`:Object с введением в них новых типов данных и переопределением всех чисто виртуальных функций. Поэтому с 3-й версии Borland C++ дополнен второй библиотекой, которая представляет собой библиотеку многократно используемых шаблонов контейнеров, предусматривающих возможность хранения как скалярных, так и объектных типов. В 5-й версии Borland C++ объектная библиотека уже отсутствует, а возможности библиотеки шаблонов контейнеров значительно расширены.

9.13.2.2. Объектные контейнеры.

Классы этой библиотеки объединены в 2 группы:

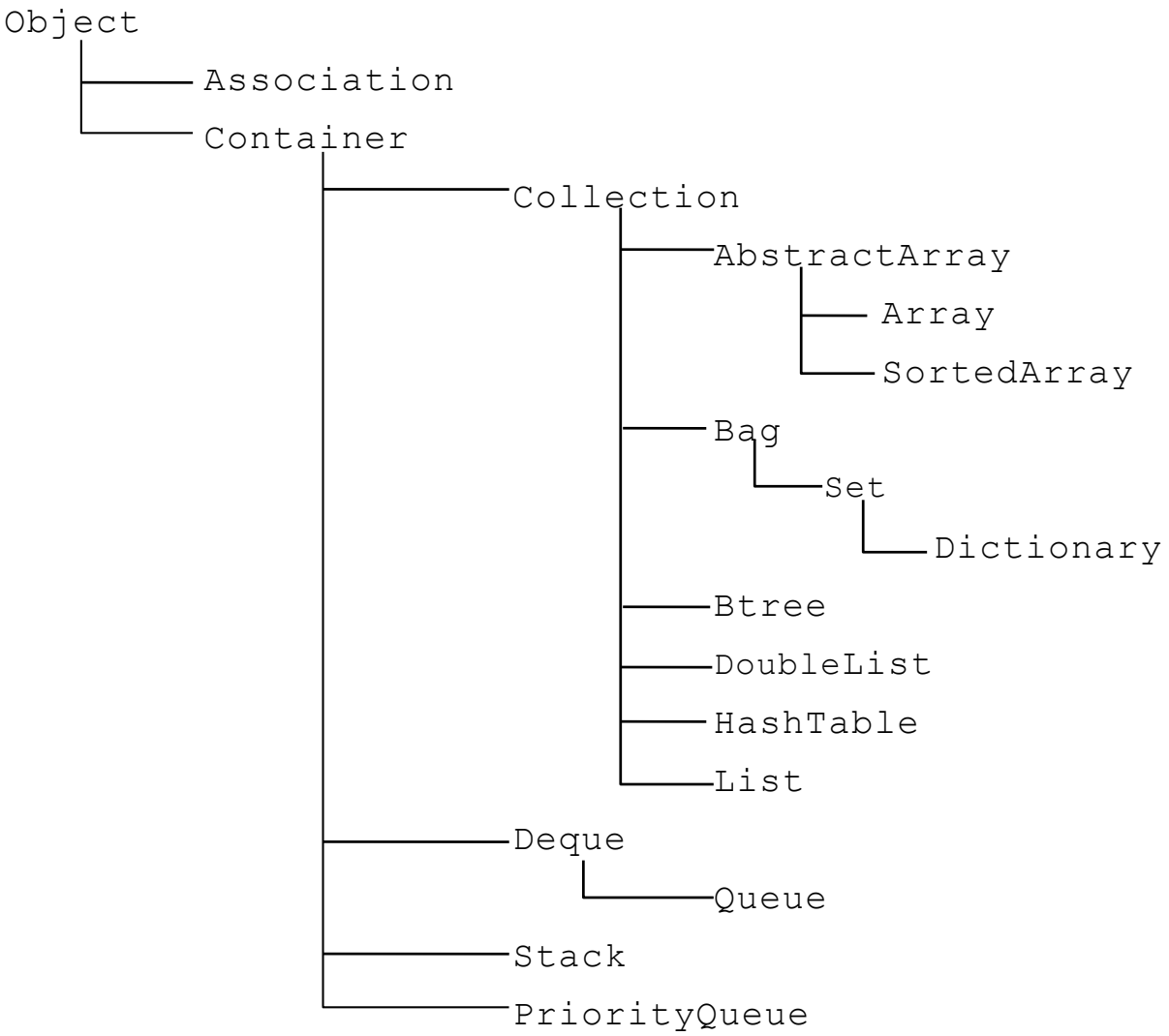
- классы собственно контейнеров;
- классы итераторов для доступа к элементам контейнеров.

Доступ к элементам контейнеров возможен либо с помощью внутренних функций-членов, либо с помощью внешних объектов-итераторов. Библиотека содержит следующий набор контейнерных классов:

массивы (Arrays)
ассоциации (Associations)
мультимножества (Bags)
бинарные деревья (Binary trees)
деки [очереди] (Deques)
словари (Dictionaries)
двусвязные списки (`Double` lists)
хэш-таблицы (Hash tables)
списки (Lists)
очереди (Queues)
множества (Sets)
стеки (Stacks)
TShould`Delete`
векторы (Vectors)

Каждый контейнерный класс предоставляет созданному на его основе объекту обширный набор услуг, реализуемый через функции-члены класса. Для начала работы с тем или иным контейнерным классом необходимо вначале ознакомиться с содержимым соответствующего включаемого файла с объявлением класса и прототипами всех функций членов.

Эти классы объединены в следующую иерархию наследования:



Итераторы специализируются по доступу к элементам определенных контейнеров и являются друзьями тех контейнеров, которые они умеют итерировать. В библиотеке есть следующий набор итераторов:

ArrayIterator
BtreeIterator
DoubleListIterator
HashTableIterator
ListIterator

Мы приведем состав только некоторых наиболее употребительных контейнерных классов, который поможет составить общее представление о возможностях объектных контейнеров.

МАССИВЫ.

В библиотеке есть два порожденных от класса AbstractArray класса массивов - Array и SortedArray.

Массивы создаются пустыми. При добавлении они вставляются в объектах класса Array на явно указанное место по индексу, а объекты класса SortedArray хранятся в отсортированном виде. При удалении принадлежащего контейнеру объекта надо удалить и элемент контейнера.

Массивы размещаются в куче и в момент создания для массива можно задать небольшой размер, который будет расти по мере добавления элементов.

Возможности массивов характеризуются следующим материалом из заголовочного файла AbstractArray :

```
class AbstractArray: public Collection {
    public:
AbstractArray( int upper, int lower = 0, sizeType s = 0);
```

Конструктор создает массив элементов с индексами от lower до upper, где lower может быть больше нуля. Если параметр s не равен нулю, то массив при переполнении будет расти, увеличиваясь каждый раз на s элементов. В противном случае создается массив фиксированного размера.

`virtual ~AbstractArray();`

Этот деструктор проверяет, владеет ли массив своими элементами. Если да, то деструктор итерируется по элементам массива и удаляет их.

`Object operator [] (int i) const;`

Функция возвращает ссылку на элемент с индексом i. Функция работает во всех классах, порожденных от AbstractArray, как на чтение, так и на запись значений с использованием выражений, подобных следующим:

```
Array array(50): Object& oBj1 = array [12]; array [13] = oBj1;
```

`int lowerBound() const;`

Функция возвращает нижнюю границу индексов массива.

`int upperBound() const;`

Это верхняя граница индексов на текущий момент.

`sizeType arraySize() const;`

Функция возвращает размер массива.

`virtual void detach( Object _FAR &, DeleteType = NoDelete );`

Если объект найден, то функция удаляет его из массива и разрушает его.

`virtual void detach( int i, DeleteType = NoDelete);`

Эта функция удаляет объект в позиции i и замещает его объектом theErrorObject.

`void destroy(int i);`

Данная функция с помощью функции-члена AbstractArray:: detach (int, DeleteType) извлекает из массива объект в позиции i и удаляет его.

virtual void flush(DeleteType = DefDelete);

Функция извлекает все объекты из AbstractArray и при необходимости удаляет их.

virtual int isEqual(const Object&) const;

Функция возвращает ненулевое значение, если два массива равны.

virtual void printContentsOn(ostream&) const;

Данная функция вызывает функцию printOn(ostream&), которая должна быть определена в производном классе, для вывода в поток печатного представления содержимого объекта класса AbstractArray. При выводе в поток пустые позиции массива пропускаются.

virtual ContainerIterator& initIterator() const;

Эта функция возвращает ссылку на объект ArrayIterator, который после этого можно использовать для итерации по объектам массива. Возвращаемый объект-итератор выделяется в куче, поэтому после работы с ним его необходимо удалить. Например:

```
Array array(100);
ContainerIterators& next = array.initIterator ();
while (next)
      cout << "\n" << (Object&) next++;
delete &next;
```

Если вы не удалите объект-итератор, то это приведет к ошибкам распределения памяти во время выполнения программы и даже к разрушению системы. Очень просто забыть об итераторе и не удалить его. Если это произойдет, компилятор будет "нем как рыба", поскольку не будет иметь точного представления о том, что вы делаете.

protected:

Object& ObjectAt(int i) const

Эта функция возвращает ссылку на объект массива, который находится в заданной позиции. Если объекта в заданной позиции нет, то возвращается NOOBJECT. Если нижняя граница массива не равна 0, то позиции массива и индексы будут разными. Например, в массиве с границами 5-10 элемент с индексом 5 будет занимать позицию 0.

Object *ptrAt(int i) const

Эта функция возвращает указатель на объект массива, находящийся в заданной позиции. Если объекта в заданной позиции нет, то возвращается указатель theErrorObject. Комментарии, сделанные при описании предыдущей функции-

члена по поводу разницы между позицией и индексом, справедливы и для данной функции.

`int` find(`const` Object&);

Эта функция просматривает AbstractArray в поисках заданного объекта. Если объект найден, то возвращается номер его позиции. В противном случае возвращается значение INT_MIN

`void Real`locate(size`Type`);

Данная функция вызывается в классах SortedArray и Array для расширения массива. Функция выделяет массив требуемого размера, затем копирует в него все элементы старого массива, сохраняя индексы объектов. Неиспользованные элементы в новом массиве заполняются указателями theErrorObject. Попытки получить доступ к таким элементам с помощью функции AbstractArray :: ObjectAt(`int`) приведут к возврату значения NOOBJECT.

`void` setData(`int`, Object *);

Благодаря этой функции позиция s (где первая позиция всегда = 0, независимо от значения нижней границы массива) указывает на заданный Object.Предыдущий указатель в позиции s затираются .

`void` insertEntry(`int` s);

Данная функция создает место для нового элемента в позиции s. Все элементы , начиная с s , поднимаются вверх на одну позицию. Значение в новой позиции s не изменится .

`void` removeEntry(`int` s);

Эта функция удаляет позицию s , сжимая массив и переписывая все элементы выше s ниже на одну позицию .

`void` squeezeEntry(`int`);

Эта функция удаляет позицию s , сжимая массив и переписывая все элементы выше s ниже на одну позицию.

size`Type` delta;

Данное значение используется для расширения массива, когда к массиву добавляется delta новых позиций. Значение delta для каждого AbstractArray фиксировано и устанавливается с помощью параметра, передаваемого конструктору. Значение по умолчанию =0. При нулевом приращении массив имеет фиксированный размер.

`int` lowerBound; `int` upperBound;

Эти величины определяют нижнюю и верхнюю границы индексов массива.

`int` lastElementIndex;

Это значение индекса, который будет использоваться при очередной вставке объекта без явного указания индекса массива. Начальное значение равно lowerBound и

увеличивается при добавлении любого нового объекта, что позволяет пользователю последовательно вставлять объекты в массив, не заботясь при этом об индексах.

АССОЦИАЦИИ (класс Association).

Под ассоциацией понимают пару взаимосвязанных информационных объектов, один из которых трактуется как ключ (key), а второй как значение (value). На практике это могут быть, например, понятие или слово как ключ и его толкование как значение.

В контейнерной библиотеке Borland C++ оба объекта, включаемые в ассоциацию, должны быть производными от класса Object. Объекты класса Association часто сами являются элементами другого контейнера Dictionary, объекты которого в этом случае представляют собой так называемый ассоциативный (по существу - толковый) словарь.

Ниже приводятся справочные данные по классу Association:

class Association: public Object, public virtual Object::TShouldDelete {

public:

Association(Object &key, Object &value);

Этот конструктор принимает ссылки на два объекта Object и использует их для инициализации двух приватных членов данных aKey и aValue.

Association(const Association &);

Этот конструктор копирования принимает ссылку на другой объект Association и инициализирует себя с тем же самым ключом и значением.

virtual ~Association();

Данный деструктор объявлен виртуальным с целью предоставить производным классам способ очистки памяти, когда они разрушаются через ссылку на Object.

Object & key() const;

Object & value() const;

Эти функции возвращают значения приватных объектов, хранящихся в Association.

virtual classType isA() const;

Функция возвращает уникальное беззнаковое целое для идентификации класса Association во время выполнения.

virtual char *nameOf() const;

Функция возвращает строку "Association".

virtual hashValueType hashValue() const;

Данная функция возвращает хэш-значение, определенное для ключа объекта ассоциации.

`virtual int` isEqual(`const` Object &) `const`;

Функция возвращает ненулевое значение (истина), когда две ассоциации имеют одинаковое значение члена aKey. Значения aValue не сравнивается.

`virtual int` isAssociation`() const`;

Данная функция просто возвращает ненулевое значение, чтобы пользователь мог убедится, что объекты, указываемые ссылкой Object&, являются ассоциациями.

`virtual void` pr`int`On(ostream &) `const`;

Эта виртуальная функция вызывается в операторе вставки в поток вывода ::`operator`<<(ostream&,Object&).

<u>ВНИМАНИЕ</u>:В Association следует передавать только те объекты, память для которых была выделена из кучи.

СЛОВАРИ (класс Dictionary).

Класс является производным от класс множества `Set` и позволяет создавать объекты с однократным для одного значения вхождения в этот контейнер с неупорядоченным хранением элементов. Для получения значения по заданному ключу не надо никаких итераций - это внутреннее дело класса, время поиска значения невелико.

Справочные данные по заголовочному файлу:

```
class Dictionary : public Set{
    public:
```

Dictionary(`unsigned` sz = `DEFAULT`_HASH_TABLE_SIZE) :

Этот конструктор вызывает конструктор базового класса и создает пустой объект Dictionary.

`virtual void` add(Object&);

Данная функция помещает объект класса Association в Dictionary.

Association& lookup(`const` Object&) `const`;

Если в словаре есть Association с заданным ключом, то возвращает эта Association , в противном случае - NOOBJECT.

`virtual classType` isA`() const`

`virtual char` *nameOf`() const`

Эти две функции предназначены для идентификации объектов Dictionary во время выполнения. Первая возвращает уникальный идентификатор, вторая - указатель на строку "Dictionary".

ПРИМЕР РАБОТЫ С ОБЪЕКТНЫМИ КОНТЕЙНЕРАМИ.

Пусть перед нами стоит задача составить программу для создания, пополнения, корректировки и эксплуатации хранящегося в дисковых файлах толкового словаря, содержащего слова и ассоциированные с ними толкования. Эксплуатация словаря состоит в том, что пользователь задает с клавиатуры слово и получает на экране соответствующее ему толкование. Создание и корректировка словаря состоит в дополнении словаря словами и их толкованиями или удалении отдельных слов и их толкований из словаря с сохранением скорректированного варианта в файлах.

```
1)  #INClude <dict.h>
2)  #INClude <assoc.h>
3)  #INClude <conio.h>
4)  #INClude <strng.h>
5)  #INClude <fstream.h>


6)  int main(int argc, char *argv[])
7)  {
 8)   char key[80], value[255];
 9)   if(argc!=2)
 10)  {
  11)   cerr<<"Используйте "<<*argv<<" vocab.txt"<<endl;
  12)   return 1;
 13)  }
 14)  Dictionary vocab;
 15)  ifstream f=*(argv+1);
 16)  if(f)
 17)  {
  18)   for(;f.getline(key,79,'$')&&f.getline(value,254,'$');
19)  vocab.add(*new Association(*new String(key),*new
  String(value))));
  20)   f.close();
 21)  }
 22)  for(;;)
 23)  {
  24)   cout<<"       Ваши действия:"<<endl
      25)   <<"1. Просмотр словаря"<<endl
      26)   <<"2. Поиск слова по ключу"<<endl
      27)   <<"3. Добавление слова"<<endl
      28)   <<"4. Удаление слова"<<endl
      29)   <<"5. Изменение толкования"<<endl
      30)   <<"6. Выход"<<endl;
```

```
31)   for(char ch=0;ch<'1'||ch>'6';ch=getch());
32)   switch(ch)
33)   {
 34)    case '1':
 35)    {
  36)    for(ContainerIterator
     &iter=vocab.initIterator();int(iter);getch())
  37)    {
   38)    Association& as=(Association&)iter++;
   39)    cout<<(const char*)(String&)as.key()<<" ЭТО "<<
      40)    (const char*)(String&)as.value()<<"\n";
  41)    }
  42)    delete &iter;
  43)    break;
 44)    }
 45)    case '2':
 46)    {
  47)    cout<<"Введите слово : ";
  48)    cin>>key;
  49)    Association& df=vocab.lookup(*new String(key));
  50)    if(df==NOOBJECT )
   51)    cout<<"Слова " <<key<< " в словаре нет\n";
  52)    else
   53)    cout<<"Толкование : "<<df.value()<<endl;
  54)    break;
 55)    }
 56)    case '3':
 57)    {
  58)    cout<<"Введите слово : ";
  59)    cin>>key;
  60)    Association& df=vocab.lookup(*new String(key));
  61)    if(df==NOOBJECT )
  62)    {
   63)    cout<<"Введите толкование : ";
   64)    for(*value=0;!strlen(value);cin.getline(value,254));
   65)    String      *s1=new      String(key),      *s2=new
      String(value);
   66)    Association *newAssoc=new Association(*s1,*s2);
   67)    vocab.add(*newAssoc);
  68)    }
  69)    else
   70)    cout<<"Это слово уже занесено в словарь"<<endl;
  71)    break;
 72)    }
```

```
73)   case '4':
74)   {
 75)   cout<<"Введите слово : ";
 76)   cin>>key;
 77)   Association& df=vocab.lookup(*new String(key));
 78)   if(df!=NOOBJECT )
  79)   vocab.detach(df);
 80)   else
  81)   cout<<"Такого слова в словаре нет"<<endl;
 82)   break;
83)   }
84)   case '5':
85)   {
 86)   cout<<"Введите слово : ";
 87)   cin>>key;
 88)   Association& df=vocab.lookup(*new String(key));
 89)   if(df!=NOOBJECT )
 90)   {
  91)   cout<<"Введите новое толкование : ";
  92)   for(*value=0;!strlen(value);cin.getline(value,254,'\n'));
  93)   (String&)(df.value())=String(value);
 94)   }
 95)   else
  96)   cout<<"Такого слова в словаре нет"<<endl;
 97)   break;
98)   }
99)   case '6':
100)  {
 101)  ofstream out=*(argv+1);
 102)  for(ContainerIterator &iter=vocab.initIterator();int(iter);)
 103)  {
  104)  Association& as=(Association&)iter++;
  105)  out<<((const char*)(String&)as.key())<<"$"<<
    106)  ((const char*)(String&)as.value())<<"$";
 107)  }
 108)  delete &iter;
 109)  return 0;
110)  }
 111)  }
 112)  }
113)  }
```

Прокомментируем, по возможности, каждую строку:

1. Эта строка содержит подключение файла, содержащего объявление класса словаря - Dictionary, хранящего в себе ссылки на созданные динамически ассоциации.
2. Эта строка содержит подключение файла, содержащего объявление класса ассоциаций - Association, контейнера, содержащего в себе два объекта - наследника абстрактного класса Object. Один из этих объектов будет являться ведущим (ключом), а другой - ведомым (значением).
3. Стандартный заголовочный файл библиотеки консольного ввода-вывода, необходимый нам только для функции получения символа без эха на экране.
4. Этот заголовочный файл содержит объявление класса для работы со строками как с объектами `String` языка Паскаль. Объекты `String` для хранения массива символов поддерживают и управляют своей собственной динамически распределенной памятью. В них есть функции , проверяющие строки на равенство и сравнивающие их.

Рассмотрим объявление класса `String`:

class `String`: public Sortable {

public:

`String`(`const char` * ="");

Этот конструктор принимает указатель на строку, заканчивающуюся нулевым символом, и копирует ее в динамически распределенный буфер. Когда объект `String` разрушается, то разрушается и буфер. Конструктору можно передать любую строку (статическую, автоматическую или глобальную), т.к. ее содержимое копируется во внутренний массив и больше не используется.

`String`(`const String`&);

Этот конструктор создает новый объект `String`, копируя содержимое другого объекта в `String` во внутренний распределенный буфер.

virtual ~`String`();

Эта функция уничтожает буфер , который использовался для хранения строки.

virtual int isEqual(`const` Object&) `const`;

virtual int isLessThan(`const` Object&) `const`;

Эти функции вызываются из глобальных операторов
::operator==(`const` Object& , `const` Object&),
::operator!=(`const` Object&, `const` Object&),
::operator!<(`const` Object&, `const` Object&).

virtual class`Type` isA() `const`;

Эта виртуальная функция возвращает уникальный идентификатор объектов String. Возвращаемое значение - это stringClass

virtual char *nameOf() const;

Эта функция возвращает указатель на строку "String"

virtual hashValueType hashValue() const;

Данная функция, основываясь на содержимом объекта String, вычисляет хэш-значение. Это значение, в свою очередь, может использоваться как индекс для данного объекта String в хэш-массивах. Хэш-значение перед началом цикла устанавливается в нуль, в результате нулевые строки всегда возвращают хэш-значение, равное нулю .

virtual void printOn(ostream&) const;

Данная функция вызывается глобальным оператором вставки в поток ::operator<<(ostream&, const Object&). Функция копирует в поток символы объекта String.

String& operator =(const String&);

Этот конструктор копирует содержимое другого объекта String. Предыдущее содержимое String затирается .

operator const char *() const;

Данный оператор возвращает указатель на строку, заканчивающуюся нулевым символом. Указатель можно использовать только для чтения, но не для записи.

private:

sizeType len;

Данный приватный член данных хранит длину динамически распределенного буфера для строки, оканчивающейся нулевым символом. Это значение (с учетом служебного символа) на единицу больше количества значащих символов в строке.

char *theString;

Это указатель на динамически распределенный буфер, в котором хранится строка, оканчивающаяся нулевым символом. При разрушении объекта String буфер удаляется .

};

5. В этом файле включения, кроме всего прочего, содержатся объявления двух основных классов для работы с файловыми потоками ввода (ifstream) и вывода (ofstream). Конструируя объекты этих классов, мы связываем их с соответствующими файлами для ввода и вывода соответственно.

6. Наша программа будет принимать всего один параметр - имя файла со словарём. При этом совершенно не существенно,

есть ли этот файл на диске или нет - лишь бы имя это было корректным.

7. Начало программы.

8. Эти два массива служат для различных операций с ключом и значением, которые нельзя осуществить непосредственно с объектами класса Association.

9. Если в командной строке меньше или больше одного необходимого нам параметра, то эта ситуация относится к ошибочным.

10. Начало обработчика ошибки.

11. Рекомендация пользователю с указанием на тип ошибки и средства её исправления.

12. Окончание программы с неким кодом завершения.

13. Конец обработчика ошибок.

14. Конструируем объект словарного класса, не передаваемая ему в качестве параметров никаких данных, поэтому создастся словарь с начальным размером, по умолчанию равным обычному размеру хэш-таблицы.

15. Конструируем объект ввода из файлового потока, передаваемая ему имя файла из командной строки для открытия. По умолчанию файл открывается как текстовый для ввода.

16. Проверяем, открылся ли файл. Так как условному оператору в качестве параметра необходимо целочисленное логическое значение 0 или не 0, то в дедушке класса ifstream - ios - пришлось определить две операторные функции: преобразования к указателю на пустой тип и неравенства. Обе эти функции служат одной цели - проверить, всё ли в порядке в потоке ввода. Если так оно и есть, первая функция (собственно, её мы и используем) вернёт ненулевой указатель, преобразуемый в длинное целое и подходящий для if, а вторая - 0. Следовательно, наше условие можно сформулировать так: "если в потоке не было ошибок", а в применении к данному конкретному случаю - если удалось открыть файл.

17. При открытом файле пытаемся его читать.

18. В этом цикле в качестве условия его окончания используются та же операторная функция преобразования ссылки на поток в обобщённый указатель, что и при открытии файла, только в данном случае мы используем его косвенно. Ссылки на поток нам вернут два вызова метода getline класса istream - папы класса ifstream, который наследует этот общедоступный метод от него. В качестве параметров этой функции-члену базового для ifstream класса передаются указатель на считываемую из файла строку, максимально возможное количество считываемых символов и символ-ограничитель строки, извле-

каемый из потока, но в строку не добавляемый. Чтение из файла в строку эта функция прекращает по достижении символа-ограничителя либо по исчерпанию лимита символов для чтения.

19. Как вы, наверное, заметили, у цикла чтения словарных единиц из файла тела нет - все необходимые действия делаются в двух секциях цикла. Договоримся о том, что слова и их толкования в файле будут разделяться каким-либо редко употребляемым значком - например, знаком доллара: этим и объясняется его наличие в качестве третьего параметра метода getline. Далее мы используем метод add словарного класса, который требует в качестве параметра объект-ассоциацию. Этой ассоциацией словарь в дальнейшем ***владеет***, то есть может изменить его место в памяти и т.д., а при выходе словарного объекта из области видимости (в нашем случае - окончания программы) в деструкторе все ассоциации, которыми словарь владеет, уничтожаются. Проблема состоит в том, что словарь принимает не объект производного от Object класса - Association, а ссылку на него. Значит, чтобы словарь корректно работал с этим объектом, он должен "жить" всё время, пока словарь с ней работает, и умирать тогда, когда словарю он не нужен. Конечно, можно было бы лишить словарь права владения ассоциацией - тогда мы сами бы следили за её использованием, но легче, эффективнее и надёжнее создавать объект динамически в оперативной памяти - тогда ссылка на него существует всегда, вплоть до явного его уничтожения, которое и проделает словарь в деструкторе, так что запись вида *new Something означает, что мы вызываем конструктор класса Something с помощью оператора new, возвращающего нам безымянный указатель на объект класса Something, созданного в оперативной памяти. Затем этот безымянный указатель разадресуется и мы получаем некий безымянный, но вполне конкретный объект, живущий ***отдельно*** от создавшей его программы в памяти компьютера. Значит, наша ассоциация создаётся не в стеке, как сам словарный объект, а в памяти динамически, и ссылка на этот ассоциативный объект добавляется в словарь. Но сама ассоциация - это тоже контейнер, владеющий своими элементами, ключом и значением. Оба этих элемента обязательно должны быть наследниками от корневого класса Object иерархии контейнерных классов; нам же хочется, чтобы это были строковые литералы типа указатель на `char`, не являющиеся наследниками Object. Компромиссным решением является использование одного из наследников Object - описанного выше строкового класса, потому мы и создаём два объекта

этого типа в памяти компьютера, как делали это с самой ассоциацией, разадресуем указатели на них, получая ссылки на безымянные объекты типа `String` в памяти и инициализируем нашу ассоциацию этими ссылками. Один строковый объект мы конструируем из ключа, а второй - из значения, считанных предыдущим оператором из файла.

20. Как только операция преобразования потока в обобщённый указатель вернёт нам 0-указатель, значит, в потоке произошла какая-то ошибка. Мы будем считать, что эта ошибка связана с достижением конца файла одной из функций чтения и вызовем метод класса ifstream для закрытия файла.

21. Конец условия "если файл удалось открыть".

22. Начинаем бесконечный цикл обработки сообщений от клавиатуры.

23.

24. В следующих семи строках мы используем перегруженный оператор вставки в стандартный поток вывода, ассоциированный с экраном, строковых литералов и стандартного манипулятора перевода строки.

25.

26.

27.

28.

29.

30.

31. Принимаем выбор пользователя, отсекая все недозволенный варианты до тех пор, пока не будет выбран какой-либо из существующих пунктов меню.

32. Начинаем анализ принятого символа, ведя по нему в дальнейшем разводку программы.

33.

34. По этому выбору мы будем выводить на экран содержимое словаря, то есть все имеющиеся в нём ассоциации.

35.

36. Бабушка Словаря - Сумка дала внуку в наследство многое. Среди всего этого богатства есть метод initIterator(), создающий динамически в памяти объект-итератор для данного контейнера и возвращающий ссылку на него типа ContainerIterator. Таким образом, бабушка передаёт нам владение этим итератором и, когда он перестанет нам быть нужным, мы должны сами его изничтожить. В классе ContainerIterator есть операторная функция преобразования итератора к целому типу (в отличие от проверки на открытие и

конец файла для потоков, на этот раз мы её вызовем явно), возвращающая ненулевое значение в том случае, если конец контейнера ещё не достигнут. При создании итераторного объекта он устанавливается на первый элемент, которым владеет контейнер (в процессе выполнения программы вы заметите, что хранятся они отнюдь не в том порядке, в котором вы их туда заносили).

37.

38. Любой объект итерационного типа может быть преобразован в ссылку на Object с помощью соответствующей операторной функции - это будет ссылка на текущий итерируемый элемент. Ссылку на Object, как и указатель на него, можно преобразовать в ссылку на любой производный от него тип, в данном случае нам приходится преобразовывать его в ссылку на ассоциацию (это делается по аналогии с преобразованием указателя на void, возвращаемого функциями выделения памяти, в указатель на нужный нам тип). После того, как мы получили ссылку на текущую ассоциацию в словаре, мы перемещаемся на следующую за ней с помощью перегруженной в классе ContainerIterator операторной функции инкремента.

39. Получив ссылку на ассоциацию, мы можем воспользоваться её методами для получения ключа и значения по этому ключу, которыми владеет данная ассоциация. Так как ключом и значением может быть любой производный от Object класс, то мы явно преобразуем полученную ссылку в ссылку на тот тип, который мы записывали и в ключ, и в значение, а именно - на тип `String`, но только для того, чтобы воспользоваться перегруженной в этом классе операторной функции преобразования строкового объекта в константный указатель на `char`. Фактически мы получаем внутреннее представление этого класса - указатель на `char` для того, чтобы вставить его в стандартный поток вывода; эту процедуру мы проделываем как с ключом, так и значением.

40.

41.

42. Метод initIterator() класса Bag, базового для Dictionary, как уже отмечалось, с помощью оператора new создает в памяти безымянный объект-итератор, разадресует и возвращает ссылку на него. У нас эта ссылка уже имеет конкретное имя, но суть этого не меняется - по адресу переменной ссылочного типа лежит динамически созданный объект, который никто, кроме нас, не уничтожит; это мы и проделываем, проитерировав весь словарь.

43. Обработка пункта "Просмотр словаря" закончена.

44.

45. Здесь мы будем искать значение слова, введённого с клавиатуры.

46.

47. Вставляем в стандартный поток вывода литерал-приглашение, вызывая перегруженную функцию для вывода в поток указателя на `char`.

48. В эту переменную мы считываем слово, которое будем искать. Перегруженный оператор ввода из поток указателя на `char` заканчивает чтения при встрече любого пробельного символа. Всё, что до него, он извлекает из потока и копирует в строку, оставляя остальное в потоке для считывания при следующем вызове этого оператора. В данном случае ввод прекратится по нажатию Enter, но в строку скопируется всё до первого пробела, табуляции или перевода строки; остальное останется в буфере клавиатуры.

49. Функция поиска в словаре требует в качестве параметра динамически созданный ключ и возвращает ссылку на найденную ассоциацию. Как и во всех предыдущих случаях, изменение значения по ссылке приведёт к изменению значения тех переменных, на которые она ссылается (чьим псевдонимом является). В частности, при некорректных действиях с полученной ссылкой на ассоциацию, вы портите значение в области памяти, на которую она ссылается, а, так как этой областью памяти владеет словарь, то портите значение в словаре. С другой стороны, при понимании выполняемых действий ссылочный механизм контейнерных классов позволяет вам "достучатся" до объектов в контейнере, манипулируя ими по своему усмотрению (пользуясь их методами, меняя их и т.п.).

50. Если ассоциация с запрошенным ключом не была найдена, метод lookup возвращает ссылку на константный объект, служащий признаком ошибки "Отсутствие запрошенного объекта". Это значение по адресу статического указателя ZERO - члена класса Object. Так как этот указатель нигде не инициализируется, то значение его при каждом запуске программы будет совершенно случайным, но, так как этот указатель статический, то это значение будет одним и тем же для все наследников Object, в том числе и ассоциаций. Только по этому мы можем использовать его в качестве флага.

51. Если ассоциация с указанным ключом в словаре не была найдена, то выводим соответствующее диагностическое сообщение.

52.

53. А коли найдено - выводим его толкование, обращаясь к соответствующему методу класса ассоциаций. Заметьте, что здесь ссылка на Object вставляется в поток непосредственно; при этом происходит вызов дружественной этому классу функции вставки в поток

```
inline ostream _FAR& operator << ( ostream _FAR& out, const
Object _FAR& obj )
{
   obj.printOn( out );
   return out;
}
```

А так как функция printOn объявлена в классе Object виртуальной и затем переопределена в строковом классе как операторная функция вывода указателя на `char`, мы можем ею воспользоваться.

54. Заканчиваем обработку пункта "Поиск слова по ключу"

55.

56. Здесь мы будем пополнять словарь новыми ассоциативными парами "слово - толкование".

57.

58. Приглашение ко вводу.

59. Вводим ключевое слово.

60. Одинаковые слова не должны входить в словарь дважды для того, чтобы не возникало неоднозначности при поиске, поэтому создадим динамический строковый объект с введённым словом как параметром и попытаемся выяснить с помощью метода поиска - а нет ли у нас уже такой словарной пары, благо поиск идёт только по ключу.

61. Если это новое слово, то таких ассоциаций у нас быть и не должно (функция поиска вернёт ссылку на отсутствующий объект); можем смело заносить такое слово в словарь.

62.

63. Приглашение к вводу.

64. Толкование слова не должно быть пустым, что и достигается приведённым циклом. Для ввода строки из потока мы снова воспользуемся функцией getline, вместо третьего параметра которой подставится значение по умолчанию - перевод строки. Следовательно, мы позволяем ввести многословной толкование, оканчивающееся при нажатии на клавишу возврата каретки.

65. Сконструируем два объекта строкового типа, выделив под них память с помощью new и запомнив возвращённые ей указатели.

66. Из этих двух динамически созданных объектов таким же способом создаём ассоциацию.

67. Вызвав функцию добавления из словарного класса, передадим словарю права владения этой ассоциацией.

68. Действия в трёх предыдущих строках расписаны только для иллюстрации - это можно сделать и одной строкой (см. П. 19).

69. Если функция поиска по введённому слову нашла-таки соответствующую ему ассоциативную пару, то позволять вводить значение по этому слову не будем - для этого есть функция модификации значения.

70. Диагностика невозможности добавления слова-дубликата.

71. Конец блока "Добавление слова".

72.

73. Начало блока "Удаление слова".

74.

75. Приглашаем ко вводу.

76. Вводим слово, которое вместе со значением надо изъять из словаря.

77. Ищем ассоциацию, ключом в которой является введённое слово.

78. Если такая ассоциация есть, функция поиска вместо признака ошибки вернёт ссылку на неё.

79. Имея ссылку на ассоциацию, мы можем её удалить с помощью метода detach, отсутствующего в словарном классе, но имеющемся выше по иерархии наследования. Этой функции, как и большинству других, передаётся ссылка на Object. В обязанности этой функции входить освободить область памяти, занимаемую объектом, на который ей передана ссылка. Это делается с помощью стандартного оператора delete, который вызывает деструктор класса, на который получена ссылка - Object. Однако этот деструктор объявлен виртуальным, поэтому вместо него вызовется деструктор ассоциативного класса, который, в свою очередь, должен уничтожить принадлежащие ему элементы. Для этих элементов по той же причине вызовутся их деструкторы, то есть деструкторы строкового типа. В деструкторе класса `String`, в свою очередь, освободят память из под указателей на `char`, по адресу которых и хранятся реальные строки. После такой разрушительной работы мы снова возвращаемся в функцию detach, которая дополнительно уменьшает на единицу внутреннюю переменную, определяющую количество элементов в контейнере.

80. Если слова не нашли - и удалять нечего.

81. Об этом и сообщим.

82. Конец обработки блока "Удаление слова".

83.

84. Начало блока "Изменение толкования".

85.

86. Приглашение ко вводу.

87. Извлекаем из стандартного потока ввода слово, предположительно имеющееся в словаре, толкование которого надо изменить.

88. Пытаемся найти ассоциацию с этим словом.

89. Если она есть, то перед нами встаёт задача - не трогая объекта-ключа, которым владеет ассоциация, изменить объект-значение.

90.

91. Приглашаем ввести новое толкование.

92. Вводим его, вызывая getline с явно указанным третьим параметром (символом перевода строки).

93. С помощью метода value класса Association получаем ссылку на Object и преобразуем её в ссылку на `String`. Это ссылка на когда-то динамически распределённый нами объект типа `String`, использовавшийся в качестве параметра-значения при создании одной из ассоциаций. Именно объект по этой ссылке мы должны изменить, поэтому вначале создаём новый объект типа `String` вызовом его конструктора с новым значением в качестве параметра и вызываем перегруженный для этого типа оператор присваивания, меняя таким образом значение по ссылке.

94. Объект типа `String` в правой части оператора присваивания создавался нами так же, как и объект класса Dictionary - в стеке, поэтому время его жизни ограничено текущим блоком. В отладчике вы можете наблюдать, как он разрушается, но не стоит горевать - перегруженный оператор присваивания его скопировал в участок памяти, соответствующий значению в ассоциации.

95. Если такого слова нет, то и значение у него мы вряд ли сможем поменять.

96. О чём и расскажем.

97. Конец блока "Изменение толкования".

98.

99. При выходе из программы мы должны переписать словарь из памяти в файл.

100.

101. Открываем файл, конструируя объект класса ofstream для вывода в поток; по умолчанию, это текстовый файл с указанным именем. Никаких проверок на его открытие не делается - позаботьтесь это сделать сами, вставив условие анализа ошибки открытия обратное тому, что приведено в строке 16.

102. Выполняемые здесь действия аналогичны тем, что были в блоке вывода на экран, за исключением того, что поток вывод теперь связан не с экраном, а с файлом. При выводе слово и его толкование разделяются знаками доллара для того, чтобы в дальнейшем этот файл вновь можно было использовать с данной программой.

103-108.

109. Выходим из программы с нулевым кодом завершения.

110. Конец блока выхода.

111. Закрывающая скобка оператора выбора пункта меню.

112. Закрывающая скобка бесконечного цикла

113. Закрывающая скобка программы. Здесь объект словарного класса выходит из области видимости, что приводит к вызову его деструктора, в котором уничтожаются все объекты, которыми он владеет - в данном случае ассоциации, которые в своих деструктора уничтожают объекты строкового типа, а те в своих деструкторах освобождают память из под хранимых в них строк.

Список использованной литературы.

1. Я. Белецкий. Энциклопедия языка Си. - М.: Мир, 1992.
2. А.П. Брудно, Л.И. Каплан. Московские олимпиады по программированию. - М.: Наука, 1990.
3. Р. Вайнер, Л. Пинсон. C++ изнутри. - Киев.: ДиаСофт, 1993.
4. Р.С. Гутер, Ю.Л. Полунов. От Абака до компьютера. - М.: Знание, 1995.
5. У. Дал, Э. Дейкстра, К. Хоор. - Структурное программирование. -М.: Мир, 1975.
6. Р.Данкан. Профессиональная работа в MS-DOS.-М.:Мир,1993.
7. Р. Джордейн. Справочник программиста персональных компьютеров типа IBM PC,XT и AT.-М.:Финансы и статистика, 1992.
8. С. Дьюхарст, К. Старк. Программирование на C++. - К.: ДиаСофт, 1993.
9. А.И. Касаткин, А.Н. Вальвачев. От Turbo C к Borland C++. - Минск.: Вышейшая школа, 1992.
10. А.И. Касаткин. Управление ресурсами. - Мн.:Выш. шк., 1992.
11. А.И. Касаткин. Системное программирование. - Минск.: Вышейшая школа, 1993.
12. Пол Лукас. C++ под рукой. - Киев.: ДиаСофт,1993.
13. П. Нортон. Программно - аппаратная организация IBM PC. - М.: Радио и связь, 1991.
14. Д.Б. Поляков, И.Ю. Круглов. Программирование в среде Турбо Паскаль (версия 5.5.).-М.:МАИ, А/О "РосВузНаука",1992.
15. Бьярн Страуструп. Язык программирования C++ в 2-х частях, вторая редакция. - Киев.: ДиаСофт, 1993.
16. Б.А. Трахтенброт. Алгоритмы и вычислительные автоматы. - М.: Советское радио, 1974.
17. Р. Уинер. Язык Турбо Си. - М.: Мир, 1991.
18. М. Уэйт, С. Прата, Д. Мартин. Язык Си: Руководство для начинающих. - М.: Мир, 1988.
19. Тэд Фейсон. Объектно-ориентированное программирование на Borland C++ 4.5. - Киев.: Диалектика, 1996.
20. А.В. Фролов, Г.В. Фролов. Операционная система MS-DOS. - М.: Диалог - МИФИ, 1992.
21. А.В. Фролов, Г.В. Фролов. Аппаратное обеспечение IBM PC. Ч. 1,2 - М.: Диалог - МИФИ, 1992.
22. А.В. Фролов, Г.В. Фролов. Программирование видеоадаптеров. - М.: Диалог - МИФИ, 1992.
23. Г. Шилдт. Си для профессионалов. - М.: Мир, 1989.
24.К.Штейнбух. Автомат и человек. - М.: Советское радио, 1967.