# Введение в ОС Linux

Linux — свободно распространяемая многозадачная, многопользовательская операционная система.

Ядро Linux разработано Линусом Торвальдсом в 1991 г. Файлы первая версия Linux (версия 0.01) были опубликованы в Интернете 17 сентября 1991 года. Распространяется

Если быть более точными, то Linux — это только ядро, когда же речь заходит об операционной системе, то более правильно говорить «Операционная система на основе ядра Linux». Ядро ОС Linux разрабатывается под общим руководством Линуса Торвальдса и распространяется свободно (на основе лицензии GPL)

К основным характеристикам Linux можно отнести многозадачность, многопользовательский доступ и разграничение прав доступа к файлам , поддержка различных форматов файловых систем

В мире существует уже более сотни различных дистрибутивов Linux. Наиболее широко распространенные: Ubuntu, OpenSuse, Fedora Core, Debian, Mandriva, Gentoo. Подавляющее большинство дистрибутивов распространяется по лицензии GPL.

### Основные характеристики ОС Linux

В силу того, что исходные коды Linux распространяются свободно и общедоступны, к развитию системы с самого начала подключилось большое число независимых разработчиков. Благодаря этому на сегодняшний момент Linux — самая современная, устойчивая и быстроразвивающаяся система, почти мгновенно вбирающая в себя самые последние технологические новшества. Она обладает всеми возможностями, которые присущи современным полнофункциональным операционным системам типа UNIX. Приведем краткий список этих возможностей.

### Реальная многозадачность

Все процессы независимы; ни один из них не должен мешать выполнению других задач. Для этого ядро осуществляет режим разделения времени центрального процессора, поочередно выделяя каждому процессу интервалы времени для выполнения. Это существенно отличается от режима "вытесняющей многозадачности", реализованной в Windows 95, когда процесс должен сам "уступить" процессор другим процессам (и может сильно задержать их выполнение).

### Многопользовательский доступ

Linux — не только многозадачная ОС, она поддерживает возможность одновременной работы многих пользователей. При этом Linux может предоставлять все системные ресурсы пользователям, работающим с хостом через различные удаленные терминалы.

### Свопирование оперативной памяти на диск

Свопирование оперативной памяти на диск позволяет работать при ограниченном объеме физической оперативной памяти; для этого содержимое некоторых частей (страниц) оперативной памяти записываются в выделенную область на жестком диске, которая трактуется как дополнительная оперативная память. Это несколько снижает скорость работы, но позволяет организовать работу программ, требующих большего объема ОЗУ, чем фактически имеется в компьютере.

### Страничная организация памяти

Системная память Linux организована в виде страниц объемом 4K. Если оперативная память полностью исчерпана, ОС будет искать давно не использованные страницы памяти для их перемещения из памяти на жесткий диск. Если какие-либо из этих страниц становятся нужны, Linux восстанавливает их с диска. Некоторые старые Unix-системы и некоторые современные платформы (включая Microsoft Windows) переносят на диск все содержимое ОП, относящееся к неработающему в данный момент приложению, (т. е. ВСЕ страницы

памяти, относящиеся к приложению, сохраняются на диске при нехватке памяти) что менее эффективно.

**Загрузка выполняемых модулей "по требованию"**

Ядро Linux поддерживает выделение страниц памяти по требованию, при котором только необходимая часть кода исполняемой программы находится в оперативной памяти, а не используемые в данный момент части остаются на диске.

**Совместное использование исполняемых программ**

Если необходимо запустить одновременно несколько копий какого-то приложения (либо один пользователь запускает несколько идентичных задач, либо разные пользователи запускают одну и ту же задачу), то в память загружается только одна копия исполняемого кода этого приложения, которая используется всеми одновременно исполняющимися идентичными задачами.

**Общие библиотеки**

Библиотеки — наборы процедур, используемых программами для обработки данных. Существует некоторое количество стандартных библиотек, используемых одновременно более чем одним процессом. В старых системах такие библиотеки включались в каждый исполняемый файл, одновременное выполнение которых приводило к непродуктивному использованию памяти. В новых системах (в частности, в Linux), обеспечивается работа с динамически и статически разделяемыми библиотеками, что позволяет сократить размер отдельных приложений.

**Динамическое кеширование диска**

Кеширование диска — это использование части оперативной памяти для хранения часто используемых данных с диска, что существенно ускоряет доступ к часто используемым программам и задачам. Пользователи MS-DOS работают со SmartDrive, который резервирует фиксированные области системной памяти для кеширования диска. Linux использует более динамичную систему кеширования: память, зарезервированная под кеш, увеличивается, когда память не используется, и уменьшается, если системе или процессу пользователя требуется больше памяти.

**100%-ное соответствие стандарту POSIX 1003.1. Частичная поддержка возможностей System V и BSD**

POSIX 1003.1 (Portable Operating System Interface — интерфейс мобильной операционной системы) задает стандартный интерфейс Unix-систем, который описывается набором процедур языка Си. Сейчас он поддерживается всеми новыми ОС. Microsoft Windows NT также поддерживает POSIX 1003.1. Linux 100%-но соответствует POSIX. Дополнительно поддерживаются некоторые возможности System V и BSD для увеличения совместимости.

**System V IPC**

Linux использует технологию IPC (InterProcess Communication) для обмена сообщениями между процессами, использования семафоров и общей памяти.

**Возможность запуска исполняемых файлов других ОС**

Linux не является первой в истории операционной системой. Для ранее разработанных ОС, включая DOS, Windows 95, FreeBSD или OS/2, разработана масса различного, в том числе очень полезного и очень неплохого программного обеспечения. Для запуска таких программ под Linux разработаны эмуляторы DOS, Windows 3.1 и Windows 95. Более того, фирмой Vmware разработана система "виртуальных машин", представляющая собой эмулятор компьютера, в котором можно запустить любую операционную систему. Имеются аналогичные разработки и у других фирм. ОС Linux способна также выполнять бинарные файлы других Intel-ориентированных Unix-платформ, соответствующих стандарту iBCS2 (intel Binary Compatibility).

**Поддержка различных форматов файловых систем**

Linux поддерживает большое число форматов файловых систем, включая файловые системы DOS и OS/2, а также современные журналируемые файловые системы. При этом и собственная файловая система Linux, которая называется Second Extended File System (ext2fs), позволяет эффективно использовать дисковое пространство.

**Сетевые возможности**

Linux можно интегрировать в любую локальную сеть. Поддерживаются все службы Unix, включая Networked File System (NFS), удаленный доступ (telnet, rlogin), работа в TCP/IP сетях, dial-up-доступ по протоколам SLIP и PPP, и т. д.. Также поддерживается включение Linux-машины как сервера или клиента для другой сети, в частности, работает общее использование (sharing) файлов и удаленная печать в Macintosh, NetWare и Windows.

**Работа на разных аппаратных платформах**

Хотя ОС Linux первоначально была разработана для ПК на базе Intel 386/486, сейчас она может работать на всех версиях Intel-овских микропроцессоров, начиная с 386 и кончая многопроцессорными системами на Pentium III (с Pentium IV возникли определенные трудности, но, судя по сообщениям в Интернете, они были вызваны ошибками в реализации процессора).Так же успешно Linux работает на различных клонах Intel от других производителей; в Интернете встречаются сообщения о том, что на процессорах Athlon и Duron от AMD Linux работает даже лучше, чем на Intel. Кроме того, разработаны версии для других типов процессоров — ARM, DEC Alpha, SUN Sparc, M68000 (Atari и Amiga), MIPS, PowerPC и других (отметим, что в настоящей книге рассматривается только вариант для IBM-совместимых компьютеров).

## Дистрибутивы Linux

В любой операционной системе можно выделить 4 основных части: ядро, файловую структуру, интерпретатор команд пользователя и утилиты. *Ядро* — это основная, определяющая часть ОС, которая управляет аппаратными средствами и выполнением программ. *Файловая структура* — это система хранения файлов на запоминающих устройствах. *Интерпретатор команд* или *оболочка* — это программа, организующая взаимодействие пользователя с компьютером. И, наконец, *утилиты* — это просто отдельные программы, которые, вообще говоря, ничем принципиально не отличаются от других программ, запускаемых пользователем, разве только своим основным назначением — они выполняют служебные функции.

Как уже говорилось выше, если быть точным, то слово "Linux" обозначает только ядро. Поэтому, когда речь идет об операционной системе, правильнее было бы говорить "операционная система, основанная на ядре Linux". Ядро ОС Linux разрабатывается под общим руководством Линуса Торвальдса и распространяется свободно (на основе лицензии GPL), как и огромное количество другого программного обеспечения, утилит и прикладных программ. Одним из следствий свободного распространения ПО для Linux явилось то, что большое число разных фирм и компаний, а также просто независимых групп разработчиков стали выпускать так называемые дистрибутивы Linux.

Дистрибутив — это набор программного обеспечения, включающий все 4 основные составные части ОС, т. е. ядро, файловую систему, оболочку и совокупность утилит, а также некоторую совокупность прикладных программ. Обычно все программы, включаемые в дистрибутив Linux, распространяются на условиях GPL, так что может сложиться впечатление, что дистрибутив может выпустить кто угодно, точнее любой, кто не поленится собрать коллекцию свободного ПО. И какая-то степень правдоподобия в таком утверждении есть. Однако разработчик дистрибутива должен по крайней мере создать программу инсталляции, которая будет устанавливать ОС на компьютер, на котором никакой ОС еще нет. Кроме того, необходимо обеспечить разрешение взаимозависимостей и противоречий

между разными пакетами (и версиями пакетов), что, как мы увидим позже, тоже является нетривиальной задачей.

Тем не менее, в мире существует уже более сотни различных дистрибутивов Linux, и все время появляются новые. Более-менее полный список их можно найти на сервере **http://www.linuxhq.com**, где даны краткие характеристики каждому дистрибутиву (упоминаются и некоторые локализованные версии). Кроме того, там же есть ссылки на другие списки дистрибутивов, так что при желании можно найти все, что вообще существует в мире (правда, все это на английском языке, и русских локализаций там маловато упомянуто).

**Файловая система**

Файловая система — это структура, с помощью которой ядро операционной системы предоставляет пользователям (и процессам) ресурсы долговременной памяти системы, т. е. памяти на различного вида долговременных носителях информации — жестких дисках, магнитных лентах. CD-ROM и т. п.

Информация в любой ОС хранится на носителях в виде файлов. Файлы группируются в каталоги, которые, в свою очередь, могут быть включены в другие каталоги. В результате получается иерархическая структура каталогов, начинающаяся с корневого каталога. Каждый (под)каталог может содержать как отдельные файлы, так и подкаталоги.

Иерархическую структуру каталогов обычно иллюстрируют рисунком "дерева каталогов", в котором каждый каталог изображается узлом "дерева", а файлы — "листьями". В MS Windows или DOS каталоговая структура строится отдельно для каждого физического носителя (т. е., имеем не отдельное "дерево", а целый "лес") и корневой каталог каждой каталоговой структуры обозначается какой-нибудь буквой латинского алфавита (отсюда уже возникает некоторое ограничение). В Linux (и UNIX вообще) строится единая каталоговая структура для всех носителей, и единственный корневой каталог этой структуры обозначается символом "/". В эту единую каталоговую структуру можно подключить любое число каталогов, физически расположенных на разных носителях (как говорят, "смонтировать файловую систему" или "смонтировать носитель").

Имена каталогов строятся по тем же правилам, что и имена файлов. И, вообще, каталоги в принципе ничем, кроме своей внутренней структуры (до которой ОС уже есть дело) не отличаются от "обычных" файлов, например, текстовых.

Полным именем файла (или путем к файлу) называется список имен вложенных друг в друга подкаталогов, начинающийся с корневого каталога и оканчивающийся собственно именем файла. При этом имена подкаталогов в этом списке разделяются тем же символом "/", который служит для обозначения корневого каталога. Например, на моем компьютере /home/kos/ve/book/filesystem1.htm является полным именем того файла, в котором я сохранил первый вариант данного текста.

В каждый момент времени пользователь работает с одним экземпляром оболочки shell и эта оболочка хранит значение так называемого "текущего" каталога, т. е. того каталога, в котором пользователь сейчас работает. Имеется специальная команда, которая сообщает вам значение текущего каталога — pwd.

В Linux типовая структура каталогов выдерживается, пожалуй, даже более строго, чем в Windows. Более того, существует даже стандарт на структуру каталогов для UNIX-подобных ОС, так называемый Filesystem Hierarchy Standart (FHS).

Стандарт FHS предлагает создать в корневом каталоге следующие подкаталоги:

- **bin** - Этот каталог содержит в основном готовые к исполнению программы,

большинство из которых необходимы во время старта системы (или в однопользовательском системном режиме, используемом для отладки). Здесь хранится значительное количество общеупотребительных команд Linux

- **boot** - неизменяемые файлы, необходимые для загрузки системы;
- **dev** - файлы устройств;
- **etc** -этот каталог и его подкаталоги содержат большинство данных, необходимых для начальной загрузки системы и основные конфигурационные файлы. В /etc находятся, например, файл inittab, определяющий загружаемую конфигурацию, и файл паролей пользователей passwd. Часть конфигурационных файлов может находится и в /usr/etc. Каталог /etc не должен содержать двоичных файлов (их следует перенести в /bin или /sbin). Ниже приводится назначение основных (но далеко не всех!) подкаталогов каталога /etc;
- **home** - домашние каталоги пользователей;
- **lib** - основные разделяемые библиотеки и модули ядра; Этот каталог содержит разделяемые библиотеки функций, необходимых компилятору языка С и модули (драйверы устройств). Даже если в системе не установлен компилятор языка С, разделяемые библиотеки необходимы, поскольку они используются многими прикладными программами. Они загружаются в память по мере необходимости выполнения каких-то функций, что позволяет уменьшить объем кода программ — в противном случае один и тот же код многократно повторялся бы в различных программах
- **mnt** - это точка монтирования для временно монтируемых файловых систем. Если на компьютере запускается поочередно Linux и MS DOS, то этот каталог обычно используется, чтобы монтировать файловую систему MS DOS. Если вы имеете привычку монтировать несколько дополнительных носителей, например, дискеты, CD-ROM, дополнительный жесткий диск и т. д., то можно создать в нем соответственно дополнительные подкаталоги для каждого носителя;
- **root** - домашний каталог пользователя суперпользователя root
- **opt** - дополнительные пакеты программного обеспечения;
- **sbin** - основные системные исполняемые файлы;
- **tmp** - временные файлы;
- **usr** - Этот каталог огромен и его структура в основном повторяет структуру корневого каталога. В его подкаталогах находятся все основные приложения. В соответствии со стандартом FHS рекомендуется выделять для этого каталога отдельный раздел диска или вообще располагать его на сетевом диске, общем для всех компьютеров в сети. Такой раздел или диск монтируют только для чтения и располагают в нем общие конфигурационные и исполняемые файлы, документацию, системные утилиты и библиотеки, а также включаемые файлы (файлы типа include);
- **var** - переменные данные.

В соответствии с требованиями стандарта приложения не должны создавать файлы и каталоги или требовать наличия каких-то специальных файлов и каталогов (помимо перечисленных) в корневом каталоге. Во-первых, размер корневой файловой системы желательно сохранять по возможности малым, а во-вторых, стандарт FHS обеспечивает достаточную гибкость и удобство размещения файлов, не попавших в корневую систему, в других файловых системах и подкаталогах. Некоторые подкаталоги корневого каталога факультативны. Но уж если они существуют, то должны размещаться в корневом каталоге, но не обязательно в корневой файловой системе.

**Права доступа к файлам и каталогам**

Поскольку Linux — система многопользовательская, вопрос об организации разграничения доступа к файлам и каталогам является одним из существенных вопросов, которые должна решать операционная система. Механизмы разграничения доступа, разработанные для системы UNIX в 70-х годах (возможно, впрочем, они предлагались кем-то и раньше), очень просты, но они оказались настолько эффективными, что просуществовали уже более 30 лет и по сей день успешно выполняют стоящие перед ними задачи.

В основе механизмов разграничения доступа лежат имена пользователей и имена групп пользователей. Вы уже знаете, что в Linux каждый пользователь имеет уникальное имя, под которым он входит в систему (логируется). Кроме того, в системе создается некоторое число групп пользователей, причем каждый пользователь может быть включен в одну или несколько групп. Создает и удаляет группы суперпользователь, он же может изменять состав участников той или иной группы. Члены разных групп могут иметь разные права по доступу к файлам, например, группа администраторов может иметь больше прав, чем группа программистов.

В индексном дескрипторе каждого файла записаны имя так называемого владельца файла и группы, которая имеет права на этот файл. Первоначально, при создании файла его владельцем объявляется тот пользователь, который этот файл создал. Точнее — тот пользователь, от чьего имени запущен процесс, создающий файл. Группа тоже назначается при создании файла — по идентификатору группы процесса, создающего файл. Владельца и группу файла можно поменять в ходе дальнейшей работы с помощью команд chown и chgrp (подробнее о них будет сказано чуть позже).

Вообще говоря, права доступа и информация о типе файла в UNIX-системах хранятся в индексных дескрипторах в отдельной структуре, состоящей из двух байтов, т. е. из 16 бит (это естественно, ведь компьютер оперирует битами, а не символами r, w, x). Четыре бита из этих 16-ти отведены для кодированной записи о типе файла. Следующие три бита задают особые свойства исполняемых файлов, о которых мы скажем чуть позже. И, наконец, оставшиеся 9 бит определяют права доступа к файлу. Эти 9 бит разделяются на 3 группы по три бита. Первые три бита задают права пользователя, следующие три бита — права группы, последние 3 бита определяют права всех остальных пользователей (т. е. всех пользователей, за исключением владельца файла и группы файла).

При этом, если соответствующий бит имеет значение 1, то право предоставляется, а если он равен 0, то право не предоставляется. В символьной форме записи прав единица заменяется соответствующим символом (r, w или x), а 0 представляется прочерком.

Право на чтение (r) файла означает, что пользователь может просматривать содержимое файла с помощью различных команд просмотра, например, командой more или с помощью любого текстового редактора. Но, подредактировав содержимое файла в текстовом редакторе, вы не сможете сохранить изменения в файле на диске, если не имеете права на запись (w) в этот файл. Право на выполнение (x) означает, что вы можете загрузить файл в память и попытаться запустить его на выполнение как исполняемую программу. Конечно, если в действительности файл не является программой (или скриптом shell), то запустить этот файл на выполнение не удастся, но, с другой стороны, даже если файл действительно является программой, но право на выполнение для него не установлено, то он тоже не запустится.

**Оболочка и графический интерфейс**

Хотя мы часто говорим, что "пользователь работает с операционной системой", фактически это не верно, поскольку на деле взаимодействие с пользователем организует специальная программа. Существует два вида таких программ — оболочка, или shell**,** для

работы в текстовом режиме (интерфейс командной строки) и графический интерфейс пользователя GUI (Graphical User Interface), организующий взаимодействие с пользователем в графическом режиме.

### Графический интерфейс

Хотя Linux представляет собой очень мощную и развитую операционную систему, но, если работать с ней только через интерфейс командной строки, она довольно трудна в обращении и "недружелюбна" к пользователю. Все необходимые операции выполняются путем запуска отдельных команд, перечень которых огромен, и которые надо помнить наизусть.

Широко известной альтернативой интерфейсу командной строки является так называемый графический интерфейс, который обеспечивает дополнительные удобства для пользователя, в частности, возможность запуска программ в отдельных окнах, обозначения программ (или других обьектов) в виде маленьких картинок (пиктограмм, значков, иконок), возможность оперировать с обьектами с помощью мыши, а также гораздо большую плотность информации на том же пространстве экрана.

Естественно, что для ОС Linux существуют средства, обеспечивающие дружественный к пользователю графический интерфейс. На первый взгляд он очень похож на широко известный графический интерфейс Microsoft Windows, но его внутреннее устройство принципиально отличается. В этой главе мы рассмотрим, как этот интерфейс работает и как его настроить.

### XFree86 и его составные части

Графический интерфейс в Linux строится на основе стандарта X Window System (заметьте, что Window, а не Windows) или просто "X" (в просторечии — "иксы"), первоначальный вариант которого был разработан в 1987 году в Массачусетском технологическом институте. Начиная со второй версии этот стандарт поддерживался консорциумом X, созданным в январе 1988 г. с целью унификации графического интерфейса для ОС UNIX. С 1997 года консорциум X преобразован в X Open Group (**http://www.x.org**). В настоящее время действует версия 11 выпуск 6 стандарта на графическую подсистему для UNIX-систем, которая кратко обозначается как X11R6.

Свободно распространяемая реализация стандарта X11R6 для UNIX-систем с процессорами 80386/80486/Pentium (в том числе для ОС Linux) была создана группой программистов, которую вначале возглавлял Дэвид Вексельблат (David Wexelblat). Эта реализация известна как XFree86 (**http://www.xfree86.org**), и может использоваться не только в Linux, но и в System V/386, 386BSD, FreeBSD и других версиях UNIX для систем на базе процессоров Intel x86. В настоящее время выпущена уже 4-ая версия XFree86, однако, и 3-я версия еще широко используется и входит в состав основных дистрибутивов Linux.

Система X Window построена на основе модели "клиент/сервер". Правда, модель эта в данном случае используется как бы в "перевернутом" виде. Дело в том, что X сервер работает на компьютере пользователя (а не на каком-то удаленном "сервере") и обеспечивает вывод изображения на экран монитора. X-сервер работает непосредственно с "железом": видеосистемой, устройствами ввода и динамиком. Эта программа захватывает оборудование и предоставляет его возможности другим программам как ресурсы (собственно, именно поэтому она и считается сервером) по особому протоколу, который называется X-протокол, или протокол сетевой связи (X Network Protocol). Кстати, специализированный компьютер, на котором исполняется исключительно X-сервер, называется (аппаратным) X-терминалом.

Но сам X-сервер изображение не формирует, он только "доставляет" графику видео-драйверу. Если запустить только X-сервер, вы увидите просто серый экран с характерным крестиком курсора посредине. С помощью мыши этот крестик можно перемещать по экрану. И все! На нажатие кнопок мыши и клавиш никакой видимой реакции не следует. И невидимой тоже — сервер готов передавать эти сигналы своим клиентам, а клиенты пока не

запущены. Хотя на самом деле некоторые комбинации клавиш X перехватывает и обрабатывает. Это <Ctrl>+<Alt>+<Backspace> — завершение работы сервера (если эта возможность не запрещена при конфигурации), <Ctrl>+<Alt>+<+> и <Ctrl>+<Alt>+<-> — "горячее" переключение доступных видеорежимов, и <Control>+<Alt>+<F#> — переключение в другую виртуальную консоль.

Чтобы получить на экране какие-то более содержательные изображения, одного X-сервера недостаточно, надо запустить менеджер окон и хотя бы одну программу-клиент, которая будет формировать изображение. В роли "клиентов" X-сервера выступают приложения, работающие с X Window, например графический редактор GIMP, текстовый редактор Corel WordPerfect, эмулятор терминала xterm и другие.

Между клиентами и сервером стоят еще два очень важных компонента графического интерфейса: библиотека графических функций X-lib и менеджер окон (рис.1). X-Lib содержит графические функции, которые обеспечивают выполнение низкоуровневых операций с графическими образами. Менеджер окон вызывает функции из X-Lib для управления дисплеем и выполнения любых преобразований изображений в окнах.

Когда X-приложение запускается, оно передает управление менеджеру окон. Менеджер окон обеспечивает выполнение всех операций с окнами: прорисовку рамок, меню, иконок, полос прокрутки и других элементов окна, а также предоставляет возможность изменять вид и положение окна в процессе работы в соответствии с потребностями пользователя.

Менеджер окон также вызывает соответствующие функции для программ-клиентов в тех случаях, когда пользователь взаимодействует с приложением с помощью клавиатуры и мыши. Именно поэтому при настройке XFree86 необходимо задать не только видеокарту, но и мышь и клавиатуру. Оконному менеджеру нужно знать характеристики этих устройств, чтобы выполнять свои задачи.

Расширенные графические среды типа Motif, CDE, KDE, GNOME, GNUStep и т. д. не замещают перечисленные выше компоненты системы X Window, а расширяют и дополняют их. KDE, например, добавляет библиотеку графических функций Qt в дополнение к X-Lib. Motif имеет собственный набор графических функций. GNOME использует библиотеку GTk +, которая составляет основу GIMP. Кроме того, в GNOME используется также CORBA (The Common Object Request Broker Architecture — универсальная архитектура посредничества при запросе объектов) и библиотека Imlib для дальнейшего расширения возможностей графической подсистемы.

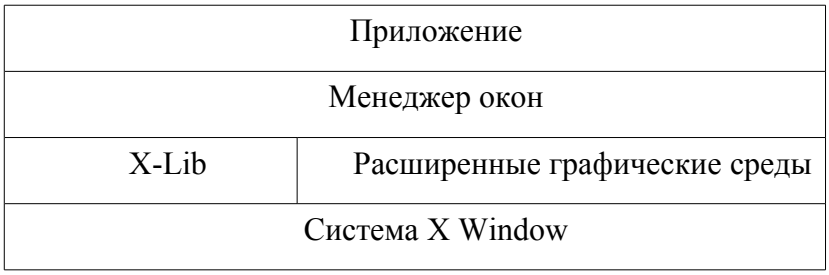


Рис 1. Архитектура графической системы в Linux

Поскольку клиент и сервер являются отдельными процессами, они могут работать на разных компьютерах, а взаимодействовать по сети.

# Команды Linux

**man** (от англ. manual — руководство) — команда Unix, предназначенная для форматирования и вывода справочных страниц.

**top** — консольная команда UNIX-совместимых операционных систем, список работающих в данный момент процессов и информацию о них. Команда top показывает список работающих в данный момент процессов и информацию о них, включая использование ими памяти и процессора. Список интерактивно формируется в реальном времени.

Чтобы выйти из программы top, нажмите клавишу [q].

**free** - Показывает количество свободной и используемой памяти в системе.

**ps** (от англ. process status) — консольная команда UNIX-совместимых операционных систем, выдающая отчёт о работающих процессах.

**ls** - выдача информации о файлах или каталогах

Синтаксис команды:

ls [флаги] [имя ...]

Команда ls для каждого имени каталога распечатывает список входящих в этот каталог файлов; для файлов - повторяется имя файла и выводится дополнительная информация в соответствии с указанными флагами. По умолчанию имена файлов выводятся в алфавитном порядке. Если имена не заданы, выдается содержимое текущего каталога. Если заданы несколько аргументов, то они сортируются по алфавиту, однако сначала всегда идут файлы, а потом каталоги с их содержимым.

**passwd** - смена входного пароля

Синтаксис команды:

passwd [входное_имя]

Команда passwd меняет (или устанавливает) пароль, связанный с входным_именем пользователя.

Обычный пользователь может менять только пароль, связанный с его собственным входным_именем.

Команда запрашивает у обычных пользователей старый пароль (если он был), а затем дважды запрашивает новый. После первого запроса проверяется, достаточен ли "возраст" старого пароля. Возраст - это промежуток времени (обычно несколько дней), который должен пройти между сменами пароля. Если возраст недостаточен, новый пароль отвергается и passwd завершается.

Если возраст достаточен, делается проверка на соответствие нового пароля техническим требованиям. Когда новый пароль вводится во второй раз, две копии нового пароля сравниваются. Если они не совпали, цикл запроса нового пароля повторяется, но не более двух раз.

**Технические требования к паролям:**

1. Каждый пароль должен содержать не менее 6 символов. Значащими являются

только первые 8.

2. Каждый пароль должен содержать как минимум две буквы (большие или малые) и хотя бы одну цифру или знак.

3. Каждый пароль должен отличаться от входного_имени, прочитанного слева направо или задом наперед, и от его циклических сдвигов. При сравнении не делается различий между большими и малыми буквами.

4. Новый пароль должен отличаться от старого хотя бы тремя символами. При сравнении не делается различий между большими и малыми буквами.

Суперпользователь (root) имеет право изменять любые пароли, поэтому у него старый пароль не запрашивается. Суперпользователь не связан ограничениями на возраст пароля и соответствие техническим требованиям. Суперпользователь может создать пустой пароль, нажимая возврат каретки в ответ на запрос нового пароля.

**su** (сокр. от англ. Substitute User) — команда Unix-подобных операционных систем, позволяющая пользователю войти в систему под другим именем, не завершая текущий сеанс. Обычно используется для временного входа суперпользователем для выполнения административных работ.

Синтаксис команды:

su [-] [имя_пользователя [аргумент ... ]]

Команда su позволяет пользователю выполнять команды от имени другого пользователя, не завершая текущий сеанс, или получить роль. По умолчанию предполагается работа от имени пользователя root (суперпользователя).

Для использования su необходимо ввести соответствующий пароль (если только команду не вызывает пользователь root). Если введен правильный пароль, su создает новый процесс командного интерпретатора, с такими же реальными и эффективными идентификаторами пользователя и группы, а также списком дополнительных групп, что и у указанного пользователя.

**sudo** - выполнить команду от имени другого пользователя

Синтаксис команды:

sudo -V | -h | -l | -L | -v | -k | -K | -s | [ -H ] [ -P ] [ -S ] [ -b ] | [ -p запрос ] [ -c класс|- ] [ -a тип_аутентификации ] [ -u имя_пользователя/#uid ] команда

sudo позволяет разрешенному пользователю выполнять команду как суперпользователь или другой пользователь, как определено в файле sudoers. Реальный и эффективный uid и gid при этом устанавливаются так, чтобы соответствовать таковым целевого пользователя, как определено в файле passwd (также инициализируется вектор группы, если целевой пользователь - не root). По умолчанию sudo требует, что бы пользователи аутентифицировали себя при помощи пароля (ЗАПОМНИТЕ: это пароль пользователя, не пароль root). Как только пользователь аутентифицировал себя происходит обновление временной метки и пользователь может использовать sudo некоторый период времени без пароля (по умолчанию пять минут, если в sudoers не указано другое).

## Работа с файлами и каталогами

**pwd** - выдача имени текущего каталога. Бывает, что при ее изучении, вы попадаете в какой-то каталог, про который уже не помните, как он называется и как вы в него попали.

Узнать его полное имя позволяет команда pwd.

**cd** - смена текущего каталога.

Синтаксис команды:

cd [каталог]

Команда cd применяется для того, чтобы сделать заданный каталог текущим. Если каталог не указан, используется значение переменной окружения $HOME (обычно это каталог, в который Вы попадаете сразу после входа в систему). Если каталог задан полным маршрутным именем, он становится текущим. По отношению к новому каталогу нужно иметь право на выполнение, которое в данном случае трактуется как разрешение на поиск.

**chmod** - изменение режима доступа к файлам

Синтаксис команды:

chmod режим файл

Права доступа к указанным файлам (среди которых могут быть каталоги) изменяются в соответствии с указанным режимом. Режим может быть задан в абсолютном или символьном виде.

Использование символьного вида основано на однобуквенных обозначениях, которые определяют класс доступа и права доступа для членов данного класса. Права доступа к файлу зависят от идентификатора пользователя и идентификатора группы, в которую он входит. Режим в целом описывается в терминах трех последовательностей, по три буквы в каждой:

| Владелец | Группа | Прочие |
|---|---|---|
| (u) | (g) | (o) |
| rwx | rwx | rwx |

Здесь владелец, члены группы и все прочие пользователи обладают правами чтения файла, записи в него и его выполнения. В примере показаны обозначения как для класса доступа, так и для прав доступа внутри класса.

Для задания режима доступа в символьном виде используется следующий синтаксис:

[кому] операция права

Часть [кому] есть комбинация букв u, g и o (владелец, члены группы и прочие пользователи соответственно). Если часть кому опущена или указано a, то это эквивалентно ugo.

Операция может быть: + (добавить право), - (лишить права), = (в пределах данного класса присвоить права абсолютно, то есть добавить указанные права и отнять неуказанные).

Права - любая осмысленная комбинация следующих букв:

r Право на чтение.

w Право на запись.

x Право на выполнение (поиск в каталоге).

s При выполнении переустанавливать действующий идентификатор пользователя или группы.

t После выполнения программы сохранять сегмент команд (бит навязчивости).

l Учет блокировки доступа.

Опустить часть права можно только если операция есть = (для лишения всех прав).

Если надо сделать более одного указания об изменении прав, то при использовании символьного вида в правах не должно быть пробелов, а указания должны разделяться запятыми. Например, команда chmod u+w,go+x f1 добавит для владельца право писать в файл f1, а для членов группы и прочих пользователей - право выполнять файл. Права устанавливаются в указанном порядке. Право s можно добавлять только для пользователя и группы, право t - только для пользователя.

Чтобы установить права, позволяющие владельцу читать и писать в файл, а членам группы и прочим пользователям только читать, надо использовать следующую запись:

chmod u=rw,go=r f1

Позволить всем выполнять файл f2

chmod +x f2

**chown** —изменить владельца файла

Только суперпользователь может изменять владельца файла. Владелец файла может изменять группу файла на любую группу, к которой он принадлежит. Суперпользователь может произвольно изменять группу.

**cp** - копирование файлов

cp файл1 [файл2 ...] целевой_файл

Команда cp копирует файл1 в целевой_файл. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a). Если целевой_файл является каталогом, то файл1, файл2, ..., копируются в него под своими именами. Только в этом случае можно указывать несколько исходных файлов.

Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Режим, владелец и группа целевого_файла при этом не меняются.

Если целевой_файл не существует или является каталогом, новые файлы создаются с теми же режимами, что и исходные (кроме бита навязчивости, если Вы не суперпользователь). Время последней модификации целевого_файла (и последнего доступа, если он не существовал), а также время последнего доступа к исходным файлам устанавливается равным времени, когда выполняется копирование. Если целевой_файл был ссылкой на другой файл, все ссылки сохраняются, а содержимое файла изменяется.

**mv** - перемещение (переименование) файлов

Синтаксис команды:

mv [-f] файл1 [файл2 ...] целевой_файл

Команда mv перемещает (переименовывает) файл1 в целевой_файл. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a). Если целевой_файл является каталогом, то файл1, файл2, ..., перемещаются в него под своими именами. Только в этом случае можно указывать несколько исходных файлов.

Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Если при этом обнаруживается, что в целевой_файл не разрешена запись, то выводится режим этого файла [см. chmod] и запрашивается строка со стандартного ввода. Если эта строка начинается с символа y, то требуемые действия все же выполняются, при условии,

что у пользователя достаточно прав для удаления целевого_файла. Если была указана опция -f или стандартный ввод назначен не на терминал, то требуемые действия выполняются без всяких запросов. Вместе с содержимым целевой_файл наследует режим файла1.

Если файл1 является каталогом, то он переименовывается в целевой_файл, только если у этих двух каталогов общий надкаталог; при этом все файлы, находившиеся в файле1, перемещаются под своими именами в целевой_файл. Если файл1 является файлом, а целевой_файл - ссылкой, причем не единственной, на другой файл, то все остальные ссылки сохраняются, а целевой_файл становится новым независимым файлом.

**rm** - удаление файлов

Синтаксис команды:

rm [-f] [-i] файл ...

rm -r [-f] [-i] каталог ... [файл ...]

Команда rm служит для удаления указанных имен файлов из каталога. Если заданное имя было последней ссылкой на файл, то файл уничтожается. Для удаления пользователь должен обладать правом записи в каталог; иметь право на чтение или запись файла не обязательно. Следует заметить, что при удалении файла в Linux, он удаляется навсегда. Здесь нет возможностей вроде "мусорной корзины" в windows 95/98/NT или команды undelete в DOS. Так что, если файл удален, то он удален!

Если нет права на запись в файл и стандартный ввод назначен на терминал, то выдается (в восьмеричном виде) режим доступа к файлу и запрашивается подтверждение; если оно начинается с буквы y, то файл удаляется, иначе - нет. Если стандартный ввод назначен не на терминал, команда rm ведет себя так же, как при наличии опции -f.

Допускаются следующие три опции:

-f Команда не выдает сообщений, когда удаляемый файл не существует, не запрашивает подтверждения при удалении файлов, на запись в которые нет прав. Если нет права и на запись в каталог, файлы не удаляются. Сообщение об ошибке выдается лишь при попытке удалить каталог, на запись в который нет прав (см. опцию -r).

-r Происходит рекурсивное удаление всех каталогов и подкаталогов, перечисленных в списке аргументов. Сначала каталоги опустошаются, затем удаляются. Подтверждение при удалении файлов, на запись в которые нет прав, не запрашивается, если задана опция -f или стандартный ввод не назначен на терминал и не задана опция -i. При удалении непустых каталогов команда rm -r предпочтительнее команды rmdir, так как последняя способна удалить только пустой каталог. Но команда rm -r может доставить немало острых впечатлений при ошибочном указании каталога!

-i Перед удалением каждого файла запрашивается подтверждение. Опция -i устраняет действие опции -f; она действует даже тогда, когда стандартный ввод не назначен на терминал.

ПРИМЕРЫ Опция -i часто используется совместно с -r. По команде:

rm -ir dirname

запрашивается подтверждение:

directory dirname: ?

При положительном ответе запрашиваются подтверждения на удаление всех содержащихся в каталоге файлов (для подкаталогов выполняются те же действия), а затем подтверждение на удаление самого каталога.

**rmdir** - удаление каталогов

Синтаксис команды:

rmdir [-p] [-s] каталог ...

Команда rmdir удаляет указанные каталоги, которые должны быть пустыми. Для удаления каталога вместе с содержимым следует воспользоваться командой rm с опцией -r. Текущий каталог [см. pwd] не должен принадлежать поддереву иерархии файлов с корнем - удаляемым каталогом.

Для удаления каталогов нужно иметь те же права доступа, что и в случае удаления обычных файлов [см. rm].

Командой rmdir обрабатываются следующие опции:

-p Позволяет удалить каталог и вышележащие каталоги, оказавшиеся пустыми. На стандартный вывод выдается сообщение об удалении всех указанных в маршруте каталогов или о сохранении части из них по каким-либо причинам.

-s Подавление сообщения, выдаваемого при действии опции -p.

**ln** - создание ссылки на файл

Синтаксис команды:

ln [-f] файл1 [файл2 ...] целевой_файл

Команда ln делает целевой_файл ссылкой на файл1. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a). Если целевой_файл является каталогом, то в нем создаются ссылки на файл1, файл2, ... с теми же именами. Только в этом случае можно указывать несколько исходных файлов.

Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Если при этом обнаруживается, что в целевой_файл не разрешена запись, то выводится режим доступа к этому файлу [см. chmod] и запрашивается строка со стандартного ввода. Если эта строка начинается с символа y, то требуемые действия все же выполняются, при условии что у пользователя достаточно прав для удаления целевого_файла. Если была указана опция -f или стандартный ввод назначен не на терминал, то требуемые действия выполняются без всяких запросов. Целевой_файл наследует режим доступа к файлу1.

ОГРАНИЧЕНИЯ

Команда ln не создает ссылок между разными файловыми системами, поскольку они (файловые системы) могут добавляться и удаляться.

**mkdir –** создание каталога

mkdir [-m режим_доступа] [-p] каталог ...

По команде mkdir создается один или несколько каталогов с режимом доступа 0777 [возможно измененном с учетом umask и опции -m]. Стандартные файлы (. - для самого каталога и .. - для вышележащего) создаются автоматически; их нельзя создать по имени. Для создания каталога необходимо располагать правом записи в вышележащий каталог.

Идентификаторы владельца и группы новых каталогов устанавливаются соответственно равными реальным идентификаторам владельца и группы процесса.

Командой mkdir обрабатываются две опции:

-m режим_доступа - (явное задание режима_доступа для создаваемых каталогов [см. chmod]).

-p (при указании этой опции перед созданием нового каталога предварительно создаются все несуществующие вышележащие каталоги).

ПРИМЕРЫ Чтобы создать поддерево каталогов tmpdir/temp/dir, надо выполнить команду

```
mkdir -p tmpdir/temp/dir
```

**grep** - поиск образца в файле

Выполняет поиск образца в текстовых файлах и выдает все строки, содержащие этот образец. Она использует компактный недетерминированный алгоритм сопоставления.

**find** - поиск файлов

Синтаксис команды:

```
find список_поиска выражение
```

Рекурсивно просматривает каждый из каталогов, перечисленных в списке_поиска, отыскивая файлы, удовлетворяющие логическому выражению,

# Файловая система Linux

## Файлы и их имена

Компьютер есть не что иное, как инструмент для обработки информации. А информация в любой ОС хранится на носителях в виде файлов. С точки зрения ОС файл представляет собой непрерывный поток (или последовательность) байтов определенной длины. Внутренний формат файла операционную систему не интересует. Но ОС должна дать файлу какое-то имя, с помощью которого пользователь, а точнее, программы-приложения, будут обращаться к файлу. Как организовать это обращение— дело файловой системы, пользователя это чаще всего не интересует. Поэтому с точки зрения пользователя файловая система выглядит как логическая структура каталогов и файлов.

Имена файлов в Linux могут иметь длину до 255 символов и состоять из любых символов, кроме символа с ADCII кодом 0 и символа / (слэша). Однако имеется еще ряд символов, которые имеют в оболочке shell специальное значение и которые поэтому не рекомендуется включать в имена. Это следующие символы:

! @ # $ & ~ % * ( ) [ ] { } ' " \ : ; > < ` пробел.

Если имя файла содержит один из этих символов (это не рекомендуется, но возможно), то вы должны перед этим символом поставить символ обратного слэша "\" (в том числе и перед самим этим слэшем, т.е. повторить его дважды).

```
[user]$ mkdir \\my\&his
```

Можно также заключить имя файла или каталога с такими символами в двойные кавычки. Например, для создания каталога с именем "My old files" следует использовать команду:

```
[user]$ mkdir "My old files"
```

так как команда

```
[user]$ mkdir My old files
```

создаст каталог с именем "My".

Аналогичным образом можно поступать и с другими символами, перечисленными выше,

т.е. их можно включать в имена файлов, если имя файла взять в двойные кавычки или отменить специальное значение символа с помощью обратного слэша. Но все же предпочтительнее не использовать эти символы, включая пробел, в именах файлов и каталогов, потому что могут возникнуть проблемы при обращении к таким файлам из некоторых приложений, а также при переносе таких файлов в другие файловые системы.

Но к точке сказанное не относится, и в Linux часто ставят более одной точки в именах файлов, например, This_is.a.forth-chapter_of_my_book.about.Linux. При этом теряет смысл такое понятие (принятое в DOS), как расширение имени файла, хотя все же часто последние части имени, отделенные точками, используют для обозначения файлов каких-то особых типов (например, .tar.gz используется для обозначения сжатых архивов). Но исполняемые и неисполняемые файлы в Linux распознаются не по расширениям имен файлов. Для этого существуют другие признаки, о которых мы скажем чуть позже. Точка имеет особое значение в именах файлов. Если она является первым символом имени, то данный файл считается скрытым для некоторых команд, например, он не показывается при выполнении команды ls.

В Linux различаются символы верхнего и нижнего регистра в именах файлов. Поэтому FILENAME.tar.gz и filename.tar.gz вполне могут существовать одновременно и являться именами разных файлов.

Мы привыкли считать, что файл полностью определяется его именем. Однако с точки зрения ОС и файловой системы это немного не так (точнее, совсем не так).

Каждому файлу в Linux соответствует так называемый "индексный дескриптор" файла, или "inode", (однозначного перевода этого термина на русский язык не существует, в разных книгах эту структуру называют по-разному). Именно индексный дескриптор содержит всю необходимую файловой системе информацию о файле, включая информацию о расположении частей файла на носителе, типе файла и многое другое. Индексные дескрипторы файлов содержатся в специальной таблице (inode table), которая создается при создании файловой системы на носителе. Каждый логический и физический диск имеет собственную таблицу индексных дескрипторов. Дескрипторы в этой таблицы пронумерованы последовательно, и именно номер дескриптора файла является его истинным именем в системе (этот номер мы будем называть индексом файла). Однако для человека такая система имен неудобна. Сможете ли вы вспомнить, что сохранили в файле с номером 56734? Поэтому файлам даются еще "человеческие" имена, и помимо этого файлы группируются в каталоги.

Приведенная выше информация нужна здесь только для того, чтобы сказать, что имя любого файла в Linux является ни чем иным, как ссылкой на индексный дескриптор файла. Поэтому каждый файл может иметь сколько угодно разных имен. Эти имена называют еще "жесткими" ссылками. Когда вы удаляете файл, имеющий несколько разных имен - жестких ссылок, то фактически удаляется только одна ссылка— та, которую вы указали в команде удаления файла. Даже когда вы удаляете последнюю ссылку, это еще может не означать удаления содержимого файла— если файл еще используется системой или каким-то приложением, то он сохраняется до тех пор, пока он не "освободится".

Для того, чтобы дать файлу (или каталогу) дополнительное имя (создать жесткую ссылку), используется команда ln в следующем формате:

```
ln имя_существующего_файла новое_имя
```

Пример:

```
[user]$ln /home/howto/font-HOWTO-ru/Font-HOWTO.html ~/fonts.html
```

(специальный символ ~ здесь и вообще в системе означает домашний каталог пользователя, о котором будет сказано чуть дальше). Теперь можно вместо длинного имени /home/howto/font-HOWTO-ru/Font-HOWTO.html использовать просто ~/fonts.html. Подробнее о команде ln вы можете прочитать на странице интерактивного руководства man.

Число жестких ссылок на файл (т. е. разных имен файла) можно узнать, выполнив команду `ls` с параметром `-l`

## Каталоги

Если бы файловая структура не позволяла использовать ничего кроме просто имен файлов, даже сколь угодно длинных (т. е. все файлы располагались бы в одном общем списке), то обращаться к ним было бы чрезвычайно трудно. Вообразите себе список из нескольких тысяч файлов! Поэтому файлы группируются в каталоги, которые, в свою очередь, могут быть включены в другие каталоги. В результате получается иерархическая структура каталогов, начинающаяся с корневого каталога. Каждый (под)каталог может содержать как отдельные файлы, так и подкаталоги.

Иерархическую структуру каталогов обычно иллюстрируют рисунком "дерева каталогов", в котором каждый каталог изображается узлом "дерева", а файлы— "листьями". В MS Windows или DOS каталоговая структура строится отдельно для каждого физического носителя (т.е., имеем не отдельное "дерево", а целый "лес") и корневой каталог каждой каталоговой структуры обозначается какой-нибудь буквой латинского алфавита (отсюда уже возникает некоторое ограничение). В Linux (и UNIX вообще) строится единая каталоговая структура для всех носителей, и единственный корневой каталог этой структуры обозначается символом "/". В эту единую каталоговую структуру можно подключить любое число каталогов, физически расположенных на разных носителях (как говорят, "смонтировать файловую систему" или "смонтировать носитель").

Имена каталогов строятся по тем же правилам, что и имена файлов. И, вообще, каталоги в принципе ничем, кроме своей внутренней структуры (до которой ОС уже есть дело) не отличаются от "обычных" файлов, например, текстовых.

Полным именем файла (или путем к файлу) называется список имен вложенных друг в друга подкаталогов, начинающийся с корневого каталога и оканчивающийся собственно именем файла. При этом имена подкаталогов в этом списке разделяются тем же символом "/", который служит для обозначения корневого каталога.

Для каждого пользователя определен его "домашний каталог"— каталог, в котором пользователь имеет все права: может создавать и удалять файлы, менять права доступа к ним и т.д. В каталоговой структуре Linux домашние каталоги пользователей обычно размещаются в каталоге /home и имеют имена, совпадающие с именем пользователя. Например, /home/jim. Каждый пользователь может обратиться к своему домашнему каталогу с помощью значка ~, т.е., например, пользователь jim может обратиться к каталогу /home/jim/doc как к ~/doc. Когда пользователь входит в систему, текущим каталогом становится домашний каталог данного пользователя.

Для изменения текущего каталога служит команда cd. В качестве параметра этой команде надо указать полный или относительный путь к тому каталогу, который вы хотите сделать текущим. Понятие полного пути уже было пояснено, а понятие относительного пути требует дополнительного пояснения. Относительным путем называется перечисление тех каталогов, которые нужно пройти в "дереве каталогов", чтобы перейти от текущего каталога к какому-то другому каталогу (назовем его целевым). Если целевой каталог, т.е. каталог, который вы хотите сделать текущим, расположен ниже текущего в структуре каталогов, то сделать это просто: вы указываете сначала подкаталог текущего каталога, затем подкаталог того каталога и т. д., вплоть до имени целевого каталога. Если же целевой каталог расположен выше в каталоговой структуре, или вообще на другой "ветви" дерева, то ситуация несколько сложнее. Конечно, можно было бы пользоваться полным путем, но тогда придется записывать очень длинные маршруты.

Эта трудность преодолевается следующим образом. Для каждого каталога (кроме корневого) в дереве каталогов однозначно определен "родительский каталог". В каждом

каталоге имеются две особых записи. Одна из них обозначается просто точкой и является указанием на этот самый каталог, а вторая запись, обозначаемая двумя точками,— указатель на родительский каталог. Эти имена из двух точек и используются для записи относительных путей. Чтобы сделать текущим родительский каталог, достаточно дать команду

```
[user]$ cd ..
```

А чтобы перейти по дереву каталогов на два "этажа" вверх, откуда спуститься в подкаталог kat1/kat2 надо дать команду

```
[user]$ cd ../../kat1/kat2
```

Команда ls служит для вывода на экран списка имен файлов и подкаталогов текущего каталога. Нужно отметить, что фактически команда ls просто выводит содержимое файла, который описывает данный каталог, и не происходит никаких обращений к самим файлам. Любой каталог, как уже говорилось,— это обычный файл, в котором перечислены все файлы и подкаталоги этого каталога.

Если дать команду ls без параметров, то выводятся только имена файлов текущего каталога. Если нужно просмотреть содержимое не текущего, а какого-то другого каталога, надо указать команде ls полный или относительный путь к этому каталогу.

**Назначение основных системных каталогов**

Если вы работали с Windows то вы знаете, что, хотя пользователь имеет полную свободу в организации структуры каталогов, некоторые "обычаи" все же сохраняются. Так системные файлы располагаются обычно в подкаталоге C:\Windows, вновь устанавливаемые программы по умолчанию размещаются в каталоге C:\Program Files и т. д.. В Linux типовая структура каталогов выдерживается, пожалуй, даже более строго. Более того, существует даже стандарт на структуру каталогов для UNIX-подобных ОС, так называемый Filesystem Hierarchy Standart (FHS), полный текст которого можно найти по адресу http://www.pathname.com/fhs/. большинство дистрибутивов Linux придерживается стандарта FHS.

Основные каталоги файловой структуры мы уже рассматривали, поэтому перечень вы можете посмотреть выше

## Типы файлов

В предыдущих разделах мы рассмотрели два типа файлов: обычные файлы и каталоги. Но в Linux существует еще несколько типов файлов. С ними мы познакомимся в этом разделе.

Как уже было сказано, с точки зрения операционной системы файл представляет собой просто поток байтов. Такой подход позволяет распространить концепцию файла на физические устройства и некоторые другие объекты. Это позволяет упростить организацию данных и обмен ими, потому что аналогичным образом осуществляется запись данных в файл, передача их на физические устройства и обмен данными между процессами. Во всех этих случаях используется один и тот же подход, основанный на идее байтового потока. Поэтому наряду с обычными файлами и каталогами, файлами с точки зрения Linux являются также:

- файлы физических устройств;
- именованные каналы (named pipes);
- гнезда (sockets);
- символические ссылки (symlinks).

**Файлы физических устройств**

Как уже говорилось, с точки зрения ОС Linux, все подключаемые к компьютеру устройства (жесткие и съемные диски, терминал, принтер, модем и т. д.), представляются

файлами. Если, например, надо вывести на экран какую-то информацию, то система как бы производит запись в файл /dev/tty01.

Физические устройства бывают двух типов: символьными (или байт-ориентированными) и блочными (или блок-ориентированными). Различие между ними состоит в том, как производится считывание и запись информации в эти устройства. Взаимодействие с символьными устройствами производится посимвольно, в режиме потока байтов. К таким устройствам относятся, например, терминалы. На блок-ориентированных устройствах информация записывается (и, соответственно, считывается) блоками. Примером устройств этого типа являются жесткие диски. На диск невозможно записать или считать с него один байт: обмен с диском производится только блоками.

Взаимодействием с физическими устройствами в Linux управляют драйверы устройств, которые либо встроены в ядро, либо подключаются к нему как отдельные модули. Для взаимодействия с остальными частями операционной системы каждый драйвер образует коммуникационный интерфейс, который выглядит как файл. Большинство таких файлов для различных устройств как бы "заготовлены заранее" и располагаются в каталоге /dev.

Если вы заглянете в каталог /dev, то увидите там огромное количество файлов физических устройств. ("Заглянуть в каталог" означает выполнить последовательно две команды cd и ls.) В табл.1 приведена небольшая справка по именам наиболее часто используемых специальных файлов.

***Таблица 1.***

*Основные специальные файлы.*

| Имя | Значение |
|---|---|
| /dev/console | Системная консоль, т.е. монитор и клавиатура, физически подключенные к компьютеру |
| /dev/hd | Жесткие диски с IDE-интерфейсом. Устройство /dev/hda1 соответствует первому разделу на первом жестком диске (/dev/hda), т.е. на диске, подключенном как Primary Master |
| /dev/sd | Жесткие диски с SCSI-интерфейсом |
| /dev/fd | Файлы дисководов для гибких дисков. Первому дисководу соответствует /dev/fd0, второму /dev/fd1 |
| /dev/tty | Файлы поддержки пользовательских консолей. Название сохранилось с тех пор, когда к системе UNIX подключались телетайпы в качестве терминалов. В Linux эти файлы устройств обеспечивают работу виртуальных консолей (переключаться между которыми можно с помощью <Alt>+<F1>— <Alt>+<F6>) |
| /dev/pty | Файлы поддержки псевдо-терминалов. Применяются для удаленных рабочих сессий с использованием telnet |
| /dev/ttS | Файлы, обеспечивающие работу с последовательными портами. /dev/ttS0 соответствует COM1 в MS-DOS, /dev/ttS1— COM2. Если ваша мышь подключается через последовательный порт, то /dev/mouse является символической ссылкой на соответствующий /dev/ttSN |
| /dev/cua | Специальные устройства для работы с модемами |
| /dev/null | Это устройство— просто черная дыра. Все, что записывается в /dev/null, навсегда потеряно. На это устройство можно перенаправить вывод ненужных сообщений. Если /dev/null используется как устройство ввода, то оно ведет себя как файл нулевой длины |

## Именованные каналы (pipes)

Еще один тип специальных файлов— именованные каналы, или буферы FIFO (First In— First Out). Файлы этого типа служат в основном для того, чтобы организовать обмен данными между разными приложениями (pipe переводится с английского как труба).

Канал— это очень удобное и широко применяемое средство обмена информацией между процессами. Все, что один процесс помещает в канал, другой может оттуда прочитать. Если два процесса, обменивающиеся информацией, порождены одним и тем же родительским процессом (а так чаще всего и происходит), канал может быть неименованным. В противном случае требуется создать именованный канал, что можно сделать с помощью программы

mkfifo. При этом собственно файл именованного канала участвует только в инициации обмена данными.

**Доменные гнезда (sockets)**

Гнезда— это соединения между процессами, которые позволяют им взаимодействовать, не подвергаясь влиянию других процессов. Вообще гнезда (и взаимодействие программ при помощи гнезд) играют очень важную роль во всех Unix-системах, включая и Linux: они являются ключевым понятием TCP/IP и соответственно на них целиком строится Интернет. Однако с точки зрения файловой системы гнезда практически неотличимы от именованных каналов: это просто метки, позволяющие связать несколько программ. После того как связь установлена, общение программ происходит без участия файла гнезда: данные передаются ядром ОС непосредственно от одной программы к другой.

Несмотря на то, что другие процессы могут видеть файлы гнезд как элементы каталога, процессы, не участвующие в данном конкретном соединении, не могут осуществлять над файлами гнезд операции чтения/записи. Среди стандартных средств, использующих гнезда - система X Window, система печати и система syslog.

**Символические ссылки**

В разделе об именах файлов уже говорилось о том, что файл в Linux может иметь несколько имен или "жестких ссылок".

Жесткая ссылка является просто еще одним именем для исходного файла. Она прописывается в индексном дескрипторе исходного файла. После создания жесткой ссылки невозможно различить, где исходное имя файла, а где ссылка. Если вы удаляете один из этих файлов (точнее одно из этих имен), то файл еще сохраняется на диске (пока у него есть хоть одно имя-ссылка).

Очень трудно различить первоначальное имя файла и позже созданные жесткие ссылки на него. Поэтому жесткие ссылки применяются там, где отслеживать различия и не требуется. Одно из применений жестких ссылок состоит в том, чтобы предотвратить возможность случайного удаления файла.

Особенностью жестких ссылок является то, что они прямо указывают на номер индексного дескриптора, а, следовательно, такие имена могут указывать только на файлы внутри той же самой файловой системы (т.е., на том же самом носителе, на котором находится каталог, содержащий это имя).

Но в Linux имеется другой тип ссылок, так называемые символические ссылки. Эти ссылки тоже могут рассматриваться как дополнительные имена файлов, но в то же время они представляются отдельными файлами— файлами типа символических ссылок. В отличие от жестких ссылок символические ссылки могут указывать на файлы, расположенные в другой файловой системе, например, на монтируемом носителе, или даже на другом компьютере. Если исходный файл удален, символическая ссылка не удаляется, но становится бесполезной. Используйте символические ссылки в тех случаях, когда хотите избежать путаницы, связанной с применением жестких ссылок.

Создание любой ссылки внешне подобно копированию файла, но фактически как исходное имя файла, так и ссылка указывают на один и тот же реальный файл на диске. Поэтому, например, если вы внесли изменения в файл, обратившись к нему под одним именем, вы обнаружите эти изменения и тогда, когда обратитесь к файлу по имени-ссылке. Для того, чтобы создать символическую ссылку, используется уже упоминавшаяся команда ln с дополнительной опцией -s:

```
ln -s имя_файла_или_каталога имя_ссылки
```

Пример:

```
[user]$ ln -s /home/kos/ve/HOWTO/font-HOWTO-ru/ ~/FONTS
```

После выполнения такой команды в моем домашнем каталоге появился подкаталог FONTS. Если теперь мы просмотрим список файлов в каталоге /home/kos с помощью команды ls -l, то среди прочих увидим такую строку:

```
lrwxrwxrwx 1 kos kos 31 Dec 13 21:13 FONTS -> /home/kos/ve/HOWTO/font-HOWTO-ru/
```

Обратите внимание на самый первый символ в этой строке: он показывает, что данная запись соответствует символической ссылке. Впрочем, это видно и в поле имени, где после нового имени и стрелки указано исходное имя файла (в данном случае— каталога).

Если вы создали в каталоге kat1 символическую ссылку, которая указывает на какой-то другой каталог, то вы можете переместить каталог kat1 куда угодно, символическая ссылка при этом будет оставаться корректной. Точно так же можно перемещать сами символические ссылки. Но остерегайтесь использовать ".." (т.е. ссылку на родительский каталог) в полных именах файлов, включающих символические ссылки, поскольку по символической ссылке нельзя проследовать в обратном направлении, а ".." всегда означает истинный родительский каталог данного каталога.

## Права доступа к файлам и каталогам

Поскольку Linux— система многопользовательская, вопрос об организации разграничения доступа к файлам и каталогам является одним из существенных вопросов, которые должна решать операционная система. Механизмы разграничения доступа, разработанные для системы UNIX в 70-х годах, очень просты, но они оказались настолько эффективными, что просуществовали уже более 30 лет и по сей день успешно выполняют стоящие перед ними задачи.

В основе механизмов разграничения доступа лежат имена пользователей и имена групп пользователей. Вы уже знаете, что в Linux каждый пользователь имеет уникальное имя, под которым он входит в систему. Кроме того, в системе создается некоторое число групп пользователей, причем каждый пользователь может быть включен в одну или несколько групп. Создает и удаляет группы суперпользователь (root), он же может изменять состав участников той или иной группы. Члены разных групп могут иметь разные права по доступу к файлам, например, группа администраторов может иметь больше прав, чем группа программистов.

В индексном дескрипторе каждого файла записаны имя так называемого владельца файла и группы, которая имеет права на этот файл. Первоначально, при создании файла его владельцем объявляется тот пользователь, который этот файл создал. Точнее— тот пользователь, от чьего имени запущен процесс, создающий файл. Группа тоже назначается при создании файла— по идентификатору группы процесса, создающего файл. Владельца и группу файла можно поменять в ходе дальнейшей работы с помощью команд chown и chgrp (подробнее о них будет сказано чуть позже).

Теперь давайте еще раз выполним команду ls -l. Но зададим ей в качестве дополнительного параметра имя конкретного файла, например, файла, задающего саму команду ls. (Обратите, кстати, внимание на эту возможность команды ls -l— получить информацию о конкретном файле, а не о всех файлах каталога сразу).

Далее следуют три группы по три символа, которые и определяют права доступа к файлу соответственно для владельца файла, для группы пользователей, которая сопоставлена данному файлу, и для всех остальных пользователей системы. В нашем примере права доступа для владельца определены как rwx, что означает, что владелец (root) имеет право читать файл (r), производить запись в этот файл (w), и запускать файл на выполнение (x). Замена любого из этих символов прочерком будет означать, что пользователь лишается соответствующего права. В том же примере мы видим, что все остальные пользователи (включая и тех, которые вошли в группу root) лишены права записи в этот файл, т.е. не могут файл редактировать и вообще как-то изменять.

Вообще говоря, права доступа и информация о типе файла в UNIX-системах хранятся в индексных дескрипторах в отдельной структуре, состоящей из двух байтов, т.е. из 16 бит (это естественно, ведь компьютер оперирует битами, а не символами r, w, x). Четыре бита из этих 16-ти отведены для кодированной записи о типе файла. Следующие три бита задают особые свойства исполняемых файлов, о которых мы скажем чуть позже. И, наконец, оставшиеся 9 бит определяют права доступа к файлу. Эти 9 бит разделяются на 3 группы по три бита. Первые три бита задают права пользователя, следующие три бита— права группы, последние 3 бита определяют права всех остальных пользователей (т.е. всех пользователей, за исключением владельца файла и группы файла).

При этом, если соответствующий бит имеет значение 1, то право предоставляется, а если он равен 0, то право не предоставляется. В символьной форме записи прав единица заменяется соответствующим символом (r, w или x), а 0 представляется прочерком.

Право на чтение (r) файла означает, что пользователь может просматривать содержимое файла с помощью различных команд просмотра, например, командой more или с помощью любого текстового редактора. Но, подредактировав содержимое файла в текстовом редакторе, вы не сможете сохранить изменения в файле на диске, если не имеете права на запись (w) в этот файл. Право на выполнение (x) означает, что вы можете загрузить файл в память и попытаться запустить его на выполнение как исполняемую программу. Конечно, если в действительности файл не является программой (или скриптом shell), то запустить этот файл на выполнение не удастся, но, с другой стороны, даже если файл действительно является программой, но право на выполнение для него не установлено, то он тоже не запустится.

Вот мы и узнали, какие файлы в Linux являются исполняемыми! Как видите, расширение имени файла тут не при чем, все определяется установкой атрибута "исполняемый", причем право на исполнение может быть предоставлено не всем!

Если выполнить ту же команду ls -l, но в качестве последнего аргумента ей указать не имя файла, а имя каталога, мы увидим, что для каталогов тоже определены права доступа, причем они задаются теми же самыми символами rwx. Например, выполнив команду ls –l /**,** мы увидим, что каталогу bin соответствует строка:

```
drwxr-xr-x 2 root root 2048 Jun 21 21:11 bin
```

Естественно, что по отношению к каталогам трактовка понятий "право на чтение", "право на запись" и "право на выполнение" несколько изменяется. Право на чтение по отношению к каталогам легко понять, если вспомнить, что каталог— это просто файл, содержащий список файлов в данном каталоге. Следовательно, если вы имеете право на чтение каталога, то вы можете просматривать его содержимое (этот самый список файлов в каталоге). Право на запись тоже понятно— имея такое право, вы сможете создавать и удалять файлы в этом каталоге, т.е. просто добавлять в каталог или удалять из него запись, содержащую имя какого-то файла и соответствующие ссылки. Право на выполнение интуитивно менее понятно. Оно в данном случае означает право переходить в этот каталог. Если вы, как владелец, хотите дать доступ другим пользователям на просмотр какого-то файла в своем каталоге, вы должны дать им право доступа в каталог, т.е. дать им "право на выполнение каталога". Более того, надо дать пользователю право на выполнение для всех каталогов, стоящих в дереве выше данного каталога. Поэтому в принципе для всех каталогов по умолчанию устанавливается право на выполнение как для владельца и группы, так и для всех остальных пользователей. И, уж если вы хотите закрыть доступ в каталог, то лишите всех пользователей (включая группу) права входить в этот каталог.

После прочтения предыдущего абзаца может показаться, что право на чтение каталога не дает ничего нового по сравнению с правом на выполнение. Однако разница в этих правах все же есть. Если задать только право на выполнение, вы сможете войти в каталог, но не увидите там ни одного файла (этот эффект особенно наглядно проявляется в том случае, если вы пользуетесь каким-то файловым менеджером, например, программой Midnight Commander).

Если вы имеете право доступа в каком-то из подкаталогов этого каталога, то вы можете перейти в него (командой cd), но, как говорится "вслепую", по памяти, потому что списка файлов и подкаталогов текущего каталога вы не увидите.

Алгоритм проверки прав пользователя при обращении к файлу можно описать следующим образом. Система вначале проверяет, совпадает ли имя пользователя с именем владельца файла. Если эти имена совпадают (т.е. владелец обращается к своему файлу), то проверяется, имеет ли владелец соответствующее право доступа: на чтение, на запись или на выполнение (не удивляйтесь, суперпользователь может лишить некоторых прав и владельца файла). Если право такое есть, то соответствующая операция разрешается. Если же нужного права владелец не имеет, то проверка прав, предоставляемых через группу или через группу атрибутов доступа для остальных пользователей, уже даже не проверяются, а пользователю выдается сообщение о невозможности выполнения затребованного действия (обычно что-то вроде "Permission denied").

Если имя пользователя, обращающегося к файлу, не совпадает с именем владельца, то система проверяет, принадлежит ли владелец к группе, которая сопоставлена данному файлу (далее будем просто называть ее группой файла). Если принадлежит, то для определения возможности доступа к файлу используются атрибуты, относящиеся к группе, а на атрибуты для владельца и всех остальных пользователей внимания не обращается. Если же пользователь не является владельцем файла и не входит в группу файла, то его права определяются атрибутами для остальных пользователей. Таким образом, третья группа атрибутов, определяющих права доступа к файлу, относится ко всем пользователям, кроме владельца файла и пользователей, входящих в группу файла.

Для изменения прав доступа к файлу используется команда chmod. Ее можно использовать в двух вариантах. В первом варианте вы должны явно указать, кому какое право даете или кого этого права лишаете:

```
[user]$ chmod wXp имя-файла
```

где вместо символа w подставляется

- либо символ u (т.е. пользователь, который является владельцем);
- либо g (группа);
- либо o (все пользователи, не входящие в группу, которой принадлежит данный файл);
- либо a (все пользователи системы, т.е. и владелец, и группа, и все остальные).

Вместо X ставится:

- либо + (предоставляем право);
- либо– (лишаем соответствующего права);
- либо = (установить указанные права вместо имеющихся),

Вместо p— символ, обозначающий соответствующее право:

- r (чтение);
- w (запись);
- x (выполнение).

Вот несколько примеров использования команды chmod:

```
[user]$ chmod a+x file_name
```

предоставляет всем пользователям системы право на выполнение данного файла.

```
[user]$ chmod go-rw file_name
```

удаляет право на чтение и запись для всех, кроме владельца файла.

```
[user]$ chmod ugo+rwx file_name
```

дает всем права на чтение, запись и выполнение.

Если опустить указание на то, кому предоставляется данное право, то подразумевается, что речь идет вообще обо всех пользователях, т.е. вместо `[user]$ chmod a+x file_name`

можно записать просто

```
[user]$ chmod +x file_name
```

Второй вариант задания команды chmod (он используется чаще) основан на цифровом представлении прав. Для этого мы кодируем символ r цифрой 4, символ w— цифрой 2, а символ x— цифрой 1. Для того, чтобы предоставить пользователям какой-то набор прав, надо сложить соответствующие цифры. Получив, таким образом, нужные цифровые значения для владельца файла, для группы файла и для всех остальных пользователей, задаем эти три цифры в качестве аргумента команды chmod (ставим эти цифры после имени команды перед вторым аргументом, который задает имя файла). Например, если надо дать все права владельцу (4+2+1=7), право на чтение и запись— группе (4+2=6), и не давать никаких прав остальным, то следует дать такую команду:

```
[user]$ chmod 760 file_name
```

Если вы знакомы с двоичным кодированием восьмеричных цифр, то вы поймете, что цифры после имени команды в этой форме ее представления есть не что иное, как восьмеричная запись тех самых 9 бит, которые задают права для владельца файла, группы файла и для всех пользователей.

Выполнять смену прав доступа к файлу с помощью команды chmod может только сам владелец файла или суперпользователь. Для того, чтобы иметь возможность изменить права группы, владелец должен дополнительно быть членом той группы, которой он хочет дать права на данный файл.

Чтобы завершить рассказ о правах доступа к файлам, надо рассказать еще о трех возможных атрибутах файла, устанавливаемых с помощью той же команды chmod. Это те самые атрибуты для исполняемых файлов, которые в индексном дескрипторе файла в двухбайтовой структуре, определяющей права на файл, занимают позиции 5-7, сразу после кода типа файла.

Первый из этих атрибутов— так называемый "бит смены идентификатора пользователя". Смысл этого бита состоит в следующем.

Обычно, когда пользователь запускает некоторую программу на выполнение, эта программа получает те же права доступа к файлам и каталогам, которые имеет пользователь, запустивший программу. Если же установлен "бит смены идентификатора пользователя", то программа получит права доступа к файлам и каталогам, которые имеет владелец файла программы (таким образом, рассматриваемый атрибут лучше называть "битом смены идентификатора владельца"). Это позволяет решать некоторые задачи, которые иначе было бы трудно выполнить. Самый характерный пример— команда смены пароля passwd. Все пароли пользователей хранятся в файле /etc/passwd, владельцем которого является суперпользователь root. Поэтому программы, запущенные обычными пользователями, в том числе команда passwd, не могут производить запись в этот файл. А, значит, пользователь как бы не может менять свой собственный пароль. Но для файла /usr/bin/passwd установлен "бит смены идентификатора владельца", каковым является пользователь root. Следовательно, программа смены пароля passwd запускается с правами root и получает право записи в файл /etc/passwd (уже средствами самой программы обеспечивается то, что пользователь может изменить только одну строку в этом файле).

Установить "бит смены идентификатора владельца" может суперпользователь с помощью команды

```
[root]# chmod +s file_name
```

Аналогичным образом работает "бит смены идентификатора группы".

Если используется цифровой вариант задания атрибутов в команде chmod, то цифровое значение этих атрибутов должно предшествовать цифрам, задающим права пользователя:

### Команды для работы с файлами и каталогами

Большинство из этих команд мы рассмотрели на предыдущей лекции (`pwd`, `cd`, `ls`, `ln`, `chmod`, *`chown`*, *`mkdir`*, *`cp`*, *`mv`*, *`find`*). Все опции этих программ можно посмотреть через запуск программы man.

## Оболочка bash

Хотя мы часто говорим, что "пользователь работает с операционной системой", фактически это не верно, поскольку на деле взаимодействие с пользователем организует специальная программа. Существует два вида таких программ — оболочка, или shell**,** для работы в текстовом режиме (интерфейс командной строки) и графический интерфейс пользователя GUI (Graphical User Interface), организующий взаимодействие с пользователем в графическом режиме.

Сразу надо сказать, что в принципе любая программа в Linux может быть запущена как через оболочку , так и через графический интерфейс пользователя. Запуск программ из оболочки эквивалентен (двойному) щелчку мышкой по иконке программы в GUI. Некоторые программы не приспособлены для запуска через GUI и, соответственно, могут быть исполнены, только из командной строки.

Когда-то (в первых UNIX-системах) это была программа с именем sh, которое было сокращением от *shell*. Потом были разработаны несколько ее улучшенных вариантов, в частности, *Bourne shell* — расширенная версия sh, написанная Стивом Борном (Steve Bourne). В рамках проекта GNU (проект Р.Столлмана по разработке свободного ПО, см. **www.gnu.org**) была создана оболочка ***bash***, название которой расшифровывается как *Bourne-again shell*, т. е. "снова оболочка Борна". По-английски в этом названии просматривается еще и игра слов, связанная с тем, что Bourne звучит как borne (рождаться, рожденный), и получается "заново рожденная shell". Оболочка bash была написана Брайеном Фоксом (Brian Fox — основной разработчик) и Четом Рэми (Chet Ramey). Именно ***bash*** мы и будем далее рассматривать, и всюду ниже, где говорится об оболочке вообще, вы смело можете считать, что речь идет о ***bash*.** Сама по себе оболочка bash не выполняет никаких прикладных задач. Но она обеспечивает выполнение всех приложений: нахождение вызываемых программ, их запуск и организацию ввода/вывода. Кроме того, оболочка отвечает за работу с переменными окружения и выполняет некоторые преобразования (подстановки) аргументов. Но главное свойство оболочки, которое делает ее мощным инструментом пользователя — это то, что она включает в себя простой язык программирования. Как давно доказано в математике, любой алгоритм можно построить из пары-тройки основных операций и одного условного оператора. Реализацию условных операторов (а также операторов цикла) и берет на себя оболочка. Она использует все остальные утилиты и программы (и те, которые имеются в составе операционной системы, и те, что устанавливаются отдельно) как базовые операции поддерживаемого ею языка программирования, обеспечивает передачу им аргументов, а также передачу результатов их работы другим программам или пользователю. В результате получается очень мощный язык программирования. И в этом основная сила и одна из существенных функций оболочки.

**Специальные символы**

Оболочка *bash* использует несколько символов из числа 256 символов набора ASCII в специальных целях, либо для обозначения некоторых операций, либо для преобразования выражений. В число таких символов входят символы:

` ~ ! @ # $ % ^ & * ( ) _ — [ ] { } : ; ' " / \ > <

а также символ с кодом 0, символ возврата каретки (генерируемый клавишей <Enter>) и пробел. В зависимости от ситуации эти специальные символы могут трактоваться либо в их специальном значении, либо в буквальном, т. е. как литералы. Но мы в основном будем предполагать, что все эти символы зарезервированы и не должны использоваться в качестве литералов. Это касается в первую очередь использования их в именах файлов и каталогов,*4*. Однако символы _, - и **.** (знак подчеркивания, дефис и точка) часто используются в именах файлов, так что именно этот пример показывает, что специальное значение эти символы имеют не всегда. В именах файлов только символы точки (.) и слэша (/) имеют специальное значение. Символ слэша служит для разделения имен отдельных каталогов, а точка имеет специальное значение только если она является первым символом в имени файла (что означает, что файл является «скрытым»).

Символ \ (обратный слэш) можно назвать "символом отмены специального значения" для любого из специальных символов, который стоит сразу вслед за \. Например, если мы хотим использовать символ пробела в имени файла, мы должны вместо простого пробела поставить \. Например, возможна следующая команда:

```
[user]$ cp two_words two\ words
```

Символы **'** и **"** (одинарные и двойные кавычки) могут быть названы "символами цитирования". Любой из этих символов всегда используется в паре с его копией для обрамления какого-то выражения, совсем как в обычной прямой речи. Если какой-то текст взят в одинарные кавычки, то все символы внутри этих кавычек воспринимаются как литералы, никаким из них не придается специального значения. Если вернуться к тому же примеру с пробелами в имени файла, то можно сказать, что для того, чтобы дать файлу имя "two words" надо взять имя в кавычки:

```
[user]$ cp two_words 'two words'
```

Различие в использовании символов ' и " состоит в том, что внутри одинарных кавычек теряют специальное значение все символы, а внутри двойных кавычек — все специальные символы кроме $, ' и \ (знака доллара, одинарных кавычек и обратного слэша).

## Выполнение команд

Как было отмечено выше, одна из основных функций оболочки состоит в том, чтобы организовать исполнение команд пользователя, вводимых им в командной строке. В частности, оболочка предоставляет пользователю два специальных оператора для организации задания команд в командной строке: ; и &.

**Оператор ;**

Хотя чаще всего пользователь задает команды в командной строке по одной, имеется возможность задать в одной строке несколько команд, которые будут выполнены последовательно, одна за другой. Для этого используется специальный символ -оператор ;. Если не поставить этот разделитель команд, то последующая команда может быть воспринята как аргумент предыдущей. Таким образом, если написать в командной строке что-то вроде:

```
[user]$ command1 ; command2
```

то оболочка вначале запустит на выполнение команду command1, дождется, пока ее выполнение завершится, после чего запустит command2, дождется ее завершения, после чего снова выведет приглашение командной строки, ожидая следующих действий пользователя.

### Оператор *&*

Оператор & используется для того, чтобы организовать исполнение команд в фоновом режиме. Если поставить значок & после команды, то оболочка вернет управление пользователю сразу после запуска команды, не дожидаясь, пока выполнение команды завершится. Например, если задать в командной строке "command1 & command2 &", то оболочка запустит команду command1, сразу же затем команду command2, и затем немедленно вернет управление пользователю.

### Операторы *&&* и ||

Операторы && и || являются управляющими операторами. Если в командной строке стоит `command1 && command2`, то command2 выполняется в том, и только в том случае, если статус выхода из команды command1 равен нулю, что говорит об успешном ее завершении. Аналогично, если командная строка имеет вид `command1 || command2`, то команда command2 выполняется тогда, и только тогда, когда статус выхода из команды command1 отличен от нуля.

Сама техника организации запуска команд на выполнение не является предметом нашего рассмотрения. Можно только кратко сказать, что оболочка должна найти код команды, загрузить его в память, передать команде аргументы, заданные в командной строке, а после завершения выполнения соответствующего процесса передать каким-то образом пользователю или другому процессу результаты выполнения данной команды. Эти этапы мы кратко и рассмотрим.

Итак, первый этап — поиск кода команды. Команды бывают встроенные (те, код которых включен в код самой оболочки) и внешние (код которых расположен в отдельном файле на диске). Встроенную команду оболочка всегда найдет, а для поиска внешней команды пользователь, в принципе, должен указать оболочке полный путь до соответствующего файла. Однако для облегчения жизни пользователей оболочка умеет искать внешние команды в каталогах, которые перечислены в специально заданных "путях поиска". Только если она не находит нужных файлов в таких каталогах, она решает, что пользователь ошибся при вводе имени команды. О том, как включить каталог в пути поиска, будет сказано ниже, а сейчас рассмотрим, как оболочка организует передачу данных исполняемой команде и выдачу результатов пользователю.

## Стандартный ввод/вывод

### Потоки ввода-вывода

Когда программа запускается на выполнение, в ее распоряжение предоставляются три потока (или канала):

- **стандартный ввод** (standard input или **stdin**). По этому каналу данные передаются программе;
- **стандартный вывод** (standard output или **stdout**). По этому каналу программа выводит результаты своей работы;
- **стандартный поток сообщений об ошибках** (standard error или **stderr**). По этому каналу программы выдают информацию об ошибках.

Из стандартного входа программа может только читать, а два других потока могут использоваться программой только для записи.

По умолчанию входной поток связан с клавиатурой, а выходной поток и поток сообщений об ошибках направлены на терминал пользователя. Другими словами, вся выходная информация запущенной пользователем команды или программы, а также все сообщения об

ошибках, выводятся в окно терминала. Однако, как мы увидим чуть ниже, можно перенаправить выходные сообщения (например, в файл).

Для того, чтобы продемонстрировать, как работает стандартный поток ошибок, выполните команду `ls` с неверным аргументом, например, задав в качестве аргумента имя несуществующего файла. В таком случае ls выведет сообщение об ошибке в стандартный поток ошибок. Для нас, однако, в данном случае стандартный поток ошибок неотличим от выходного потока, поскольку сообщение об ошибке мы видим в окне терминала.

Работу со стандартными входным и выходным потоками лучше всего проиллюстрировать на примере команд `echo` и `cat`.

### Команда `echo`

Команда `echo` предназначена для выдачи на стандартный вывод строки символов, которая задана ей в качестве аргумента. После этого она выдает сигнал перевода строки и завершается. Попробуйте выполнить команду

```
[user]$ echo 'Привет, дружище!'
```

и, думаю, дальнейших пояснений не потребуется (только используйте именно одиночные кавычки, иначе результат может быть несколько иным).

### Команда `cat`

Мы уже рассматривали кратко команду `cat` в предыдущем разделе. По умолчанию выход команды cat направляется в выходной поток. Ксли запустить эту команду без параметров курсор переместится в новую строку, и более как будто ничего не будет происходить. В это время команда ожидает поступления символов во входном потоке.Если ввести символ, он сразу же появился на экране, что говорит о том, что программа сразу же направила его в выходной поток. Можно продолжить ввод символов, и они также появятся на экране.

Обычно клавиатура настроена на построчный ввод, поэтому если нажать клавишу <Enter>, последняя набранная строка передается команде `cat`, которая вновь выводит данные на монитор через стандартный вывод. Таким образом, каждая строка будет показана дважды: один раз при наборе и второй раз — командой cat.

Если нажать комбинацию клавиш <Ctrl>+<D>, которая служит командой окончания процедуры ввода, вы вновь вернетесь к подсказке в командной строке. Можно также использовать комбинацию клавиш <Ctrl>+<C>, которая является в оболочке командой завершения работы запущенной программы.

Если команде `cat` в качестве аргумента задать имя файла, это будет означать, что содержимое файла будет направлено во входной поток, откуда его примет команда `cat` и выдаст в выходной поток.

## Перенаправление ввода/вывода, каналы и фильтры

### Операторы >, < и >>

Для обозначения перенаправления используются символы ">", "<" и ">>". Чаще всего используется перенаправление вывода команды в файл. Вот соответствующий пример:

```
[user]$ ls -l > /home/jim/dir.txt
```

По этой команде в файле /home/jim/dir.txt будет сохранен перечень файлов и подкаталогов того каталога, который был текущим на момент выполнения команды ls; при этом если указанного файла не существовало, то он будет создан; если он существовал, то будет

перезаписан; если же вы хотите, чтобы вывод команды был дописан в конец существующего файла, то надо вместо символа > использовать >>. При этом наличие пробелов до или после символов **>** или **>>** несущественно и служит только для удобства пользователя.

Вы можете направить вывод не только в файл, но и на вход другой команды или на устройство (например, принтер). Так, для подсчета числа слов в файле /home/jim/report.txt можно использовать следующую команду:

```
[user]$ cat /home/jim/report.txt > wc -w
```

а для вывода файла на печать — команду:

```
[user]$ cat /home/jim/report.txt > lpr
```

Как видите, оператор **>** служит для перенаправления выходного потока. По отношению к входному потоку аналогичную функцию выполняет оператор <. Приведенный выше пример команды для подсчета числа слов в определенном файле можно переписать следующим образом (обратите внимание на отсутствие команды cat):

```
[user]$ wc -w < /home/jim/report.txt
```

Этот вариант перенаправления часто используется в различных скриптах, применительно к тем командам, которые обычно воспринимают ввод (или ожидают ввода) с клавиатуры. В скрипте же, автоматизирующем какие-то рутинные операции, можно дать команде необходимую информацию из файла, в который заранее записано то, что нужно ввести для выполнения этой команды.

В силу того, что символы <, > и >> действуют на стандартные потоки, их можно использовать не только тем привычным образом, как это делается обычно, но и несколько по-другому. Так, следующие команды эквивалентны:

```
[user]$ cat > file
[user]$ cat>file
[user]$ >file cat
[user]$ > file cat
```

Однако сам по себе (без какой-либо команды, для которой определены стандартные потоки) символ перенаправления не может использоваться, так что нельзя, например, введя в командной строке

```
[user]$ file1 > file2
```

получить копию какого-то файла.

Стандартные потоки определены для любой команды. При этом перенаправить можно не только стандартный ввод и вывод, но и другие потоки. Для этого надо указать перед символом перенаправления номер перенаправляемого потока. Стандартный ввод stdin имеет номер 0, стандартный вывод stdout — номер 1, стандартный поток сообщений об ошибках stderr — номер 2. Полный формат команды перенаправления:

```
command N > M
```

где N и M — номера стандартных потоков (0,1,2) или имена файлов. Употребление в некоторых случаях символов <, > и >> без указания номера канала или имени файла возможно только потому, что вместо отсутствующего номера по умолчанию подставляется 1, т. е. стандартный вывод. Так, оператор > без указания номера интерпретируется как 1 >.

Кроме простого перенаправления стандартных потоков существует еще возможность не просто перенаправить поток в тот или иной канал, а сделать копию содержимого стандартного потока. Для этого служит специальный символ &, который ставится перед номером канала, на который перенаправляется поток:

```
command N > &M
```

Такая команда означает, что выход канала с номером N направляется как на стандартный вывод, так и дублируется в канал с номером M. Например, для того, чтобы сообщения об ошибках дублировались на стандартный вывод, надо дать команду 2>&1, в то время как 1>&2 дублирует stdout в stderr. Такая возможность особенно полезна при перенаправлении вывода в файл, так как мы тогда одновременно и видим сообщения на экране, и сохраняем их в файле.

**Оператор |**

Особым вариантом перенаправления вывода является организация программного канала (иногда называет трубопроводом или конвейером). Для этого две или несколько команд, таких, что вывод предыдущей служит вводом для следующей, соединяются (или разделяются, если вам это больше нравится) символом вертикальной черты — "|". При этом стандартный выходной поток команды, расположенной слева от символа |, направляется на стандартный ввод программы, расположенной справа от символа |. Например:

```
[user]$ cat myfile | grep Linux | wc -l
```

Эта строка означает, что вывод команды `cat`, т. е. текст из файла myfile, будет направлен на вход команды `grep`, которая выделит только строки, содержащие слово "Linux". Вывод команды `grep` будет, в свою очередь, направлен на вход команды `wc -l` , которая подсчитает число таких строк.

Программные каналы используются для того, чтобы скомбинировать несколько маленьких программ, каждая из которых выполняет только определенные преобразования над своим входным потоком, для создания обобщенной команды, результатом которой будет какое-то более сложное преобразование.

Надо отметить, что оболочка одновременно вызывает на выполнение все команды, включенные в конвейер, запуская для каждой из команд отдельный экземпляр оболочки, так что как только первая программа начинает что-либо выдавать в свой выходной поток, следующая команда начинает его обрабатывать. Точно так же каждая следующая команда выполняет свою операцию, ожидая данных от предыдущей команды и выдавая свои результаты на вход последующей. Если вы хотите, чтобы какая-то команда полностью завершилась до начала выполнения последующей, вы можете использовать в одной строке как символ конвейера |, так и точку с запятой **;**. Перед каждой точкой с запятой оболочка будет останавливаться и ожидать, пока завершится выполнение всех предыдущих команд, включенных в конвейер.

Статус выхода (логическое значение, возвращаемое после завершения работы программы) из канала совпадает со статусом выхода, возвращаемым последней командой конвейера. Перед первой командой конвейера можно поставить символ "!", тогда статус выхода из конвейера будет логическим отрицанием статуса выхода из последней команды. Оболочка ожидает завершения всех команд конвейера, прежде чем установить возвращаемое значение.

## Параметры и переменные. Окружение оболочки

Понятие параметра в оболочке bash подобно понятию переменной в обычных языках программирования. Именем (или идентификатором) параметра может быть слово, состоящее из алфавитных символов, цифр и знаков подчеркивания (только первый символ этого слова не может быть цифрой), а также число или один из следующих специальных символов:

*, @, #, ?, - (дефис), $, !, 0, _ (подчеркивание).

Говорят, что параметр задан или установлен, если ему присвоено значение. Значением может быть и пустая строка.

Значения переменным присваиваются с помощью оператора следующего вида

```
[user]$ name=value
```

где `name` — имя переменной, а `value` — присваиваемое ей значение (может быть пустой строкой).

Значением может быть любой текст. Если значение содержит специальные символы, то его надо взять в кавычки. Присвоенное значение этих кавычек не содержит, естественно. Если переменная задана, то ее можно удалить, используя встроенную команду оболочки `unset`.

Для того чтобы вывести значение одной конкретной переменной, можно воспользоваться командой

```
[user]$ echo $name
```

(нужно использовать символ $ перед его именем). Так, команда

```
[user]$ echo name
```

выдаст на экран слово name, а команда

```
[user]$ echo $name
```

выдаст значение переменной name (если таковое, конечно, задано).

**Разновидности параметров**

Параметры разделяются на три класса: *позиционные параметры, специальные параметры* (именами которых как раз и служат перечисленные только что специальные символы) и *переменные оболочки*.

Имена (идентификаторы) *позиционных параметров* состоят из одной или более цифр (только не из одиночного нуля). Значениями позиционных параметров являются аргументы, которые были заданы при запуске оболочки (первый аргумент является значением позиционного параметра 1, и т. д.). Изменить значение позиционного параметра можно с помощью встроенной команды set. Значения этих параметров изменяются также на время выполнения оболочкой одной из функций

**Переменные окружения. Переменная *PATH***

Одним из важнейших понятий в ОС Linux является переменные окружения (или переменные среды, environment variables) -переменные, к которым имеют доступ все исполняемые команды. Они могут быть использованы как ОС, так и пользовательскими программами. Эти переменные задаются при помощи символьного имени и некоего значения.

**Некоторые переменные окружения**

| Название | Назначение |
|---|---|
| PATH | Пути для поиска программ -- список директорий, разделенных двоеточиями |
| PROMPT, prompt, PS1 | Вид приглашения shell |
| DISPLAY | Имя дисплея для X-программ |

| | |
|---|---|
| TERM | Тип терминала |
| EDITOR | Текстовый редактор, который будут использовать программы mc, vipw, crontab и т.д. вместо vi |
| PAGER | Программа просмотра текстовых файлов, которую будут использовать команды man, apropos и т.д. (в Linux man и apropos по умолчанию используют /usr/bin/less -is |
| HOME* | Домашняя директория пользователя |
| USER*<br>LOGNAME* | Login-имя пользователя |
| SHELL* | Имя основного shell |

Символом "*" помечены те переменные, которые являются "информационными" и которые не следует изменять.

Чтобы присвоить значение переменной окружения или изменить его, используется команда export. Пример:

```
export DISPLAY=localhost:0
```

Вокруг символа "=" не должно быть пробелов, а если пробелы есть в присваиваемом значении, то его надо заключить в кавычки.

Для того ,чтобы посмотреть все переменные окружения и их значения, нужно запустить команду env или команду export без параметров.

Чтобы посмотреть значение переменной, можно воспользоваться командой echo:

echo $DISPLAY

echo $PATH

Для удаления переменной окружения (это не то же самое, что присвоение ей пустой строки!) используется команда unset.

Одной из важнейших переменных окружения является переменная PATH.

Она задает перечень путей к каталогам, в которых bash осуществляет поиск файлов (в частности, файлов с командами) в тех случаях, когда полный путь к файлу не задан в командной строке. Отдельные каталоги в этом перечне разделяются двоеточиями. По умолчанию переменная PATH включает каталоги /usr/local/bin, /bin, /usr/bin, /usr/X11R6/bin, т. е. имеет вид:

```
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:
```

Для того, чтобы добавить каталог в этот список, нужно выполнить следующую команду:

```
[root]# export PATH=$PATH:new_path.
```

При осуществлении поиска оболочка просматривает каталоги именно в том порядке, как они перечислены в переменной PATH.

в Unix-подобных ОС (включая и Linux) команды ищутся *только* в директориях, перечисленных в PATH - если "." там не указана (а обычно так и есть), то в текущей директории программа искаться не будет. Для запуска программы из текущей директории надо явно указать путь,например:

```
./myprog
```

## Скрипты оболочки

Скрипт оболочки – это файл, содержащий команды оболочки. Скрипты можно выполнять как обычные команды. Если при запуске такого файла заданы аргументы, на время выполнения скрипта они становятся позиционными параметрами.

В каждом файле, задающем скрипт первая строка имеет вид:

```
#!/bin/bash
```

Это означает, что когда мы запускаем скрипт на выполнение как обычную команду, `/bin/bash` будет выполнять ее для нас.

Помимо использования позиционных параметров, возможно использование и других переменных, определяемых внутри скрипта.

Например:

`fruit = apple` (определение);

`echo $fruit` (доступ);

Возможна конкатенация строк:

```
$ fruit = apple
$ fruit = pine$fruit
$ echo $fruit
pineapple
$ fruite = apple
$ wine = ${fruite}jack
$ echo $wine
applejack
```

**Ввод с клавиатуры.**

Переменные можно считывать со стандартного ввода. Для этого используется команда `read`

```
echo -n Enter number of elements:
   read x
```

## Управляющие структуры shell

Оболочка bash поддерживает операторы выбора if … then … else и case, а также операторы организации циклов `for, while, until,` благодаря чему она превращается в мощный язык программирования.

**Операторы *if* и *test* (или *[ ]*)**

Конструкция условного оператора в слегка упрощенном виде выглядит так:

```
if list1 then list2 else list3 fi
```

где `list1, list2 и list3` — это последовательности команд, разделенные запятыми и оканчивающиеся точкой с запятой или символом новой строки. Кроме того, эти последовательности могут быть заключены в фигурные скобки: `{list}`.

Оператор `if` проверяет значение, возвращаемое командами из `list1`. Если в этом списке несколько команд, то проверяется значение, возвращаемое последней командой списка. Если

это значение равно 0, то будут выполняться команды из `list2`; если это значение не нулевое, будут выполнены команды из `list3`. Значение, возвращаемой таким составным оператором `if`, совпадает со значением, выдаваемым последней командой выполняемой последовательности.

Полный формат команды `if` имеет вид:

```
if list then list [ elif list then list ] ... [ else list ] fi
```

(здесь квадратные скобки означают только необязательность присутствия в операторе того, что в них содержится).

В качестве выражения, которое стоит сразу после `if` или `elif`, часто используется команда test, которая может обозначаться также квадратными скобками [ ]. Команда `test` выполняет вычисление некоторого выражения и возвращает значение 0, если выражение истинно, и 1 в противном случае. Выражение передается программе `test` как аргумент. Вместо того, чтобы писать

```
test expression,
```

можно заключить выражение в квадратные скобки:

```
[ expression ].
```

Заметьте, что `test` и `[` — это два имени одной и той же программы, а не какое-то магическое преобразование, выполняемое оболочкой bash (только синтаксис [ требует, чтобы была поставлена закрывающая скобка). Заметьте также, что вместо `test` в конструкции `if` может быть использована любая программа.

В заключение приведем пример использования оператора if:

```
if [ -e textmode2.htm ] ; then
ls textmode*
else
pwd
fi
```

Об операторе test (или […]) надо бы поговорить особо.

**Оператор *test* и условные выражения**

Условные выражения, используемые в операторе `test`**,** строятся на основе проверки файловых атрибутов, сравнения строк и обычных арифметических сравнений числовых значений. Сложные выражения строятся из следующих унарных или бинарных операций (для каждого типа проверок существуют свои примитивы):

**Проверка файловых атрибутов**

- `-a file`

Верно, если файл с именем `file` существует.

- `-b file`

Верно, если file существует и является специальным файлом блочного устройства.

- `-c file`

Верно, если file существует и является специальным файлом символьного устройства.

- `-d file`

Верно, если file существует и является каталогом.

- `-e file`

Верно, если файл с именем file существует.

- `-f file`

Верно, если файл с именем file существует и является обычным файлом.

- `-g file`

Верно, если файл с именем file существует и для него установлен бит смены группы.

- `-h file` или `-L file`

Верно, если файл с именем file существует и является символической ссылкой.

- `-k file`

Верно, если файл с именем file существует и для него установлен "sticky" bit.

- `-p file`

Верно, если файл с именем file существует и является именованным каналом (FIFO).

- `-r file`

Верно, если файл с именем file существует и для него установлено право на чтение

- `-s file`

Верно, если файл с именем file существует и его размер больше нуля.

- `-t fd`

Верно, если дескриптор файла fd открыт и указывает на терминал.

- `-u file`

Верно, если файл с именем file существует и для него установлен бит смены пользователя.

- `-w file`

Верно, если файл с именем file существует и для него установлено право на запись.

- `-x file`

Верно, если файл с именем file существует и является исполняемым.

- `-O file`

Верно, если файл с именем file существует и его владельцем является пользователь, на которого указывает эффективный идентификатор пользователя.

- `-G file`

Верно, если файл с именем file существует и принадлежит группе, определяемой эффективным идентификатором группы.

- `-S file`

Верно, если файл с именем file существует и является сокетом.

- `-N file`

Верно, если файл с именем file существует и изменялся с тех пор, как был последний раз прочитан.

- `file1 -nt file2`

Верно, если файл file1 имеет более позднее время модификации, чем file2.

- `file1 -ot file2`

Верно, если файл file1 старше, чем file2.

- `file1 -ef file2`

Верно, если файлы file1 и file2 имеют одинаковые номера устройств и индексных дескрипторов (inode).

**Оценка строк**

- `-z string`

Верно, если длина строки равна нулю.

- `-n string`

Верно, если длина строки не равна нулю.

- `string1 == string2`

Верно, если строки совпадают. Вместо == может использоваться =.

- `string1 !== string2`

Верно, если строки не совпадают.

- `string1 < string2`

Верно, если строка string1 лексикографически предшествует строке string2 (для текущей локали).

- `string1 > string2`

Верно, если строка string1 лексикографически стоит после строки string2 (для текущей локали).

**Арифметические сравнения**

- `arg1 OP arg2`

Здесь OP — это одна из операций арифметического сравнения: `-eq` (равно), `-ne` (не равно), `-lt` (меньше чем), `-le` (меньше или равно), `-gt` (больше), `-ge` (больше или равно). В качестве аргументов могут использоваться положительные или отрицательные целые.

Из этих элементарных условных выражений можно строить сколь угодно сложные с помощью обычных логических операций ОТРИЦАНИЯ, И и ИЛИ:

- `!(expression)`

Булевский оператор отрицания.

- `expression1 -a expression2`

Булевский оператор AND (И). Верен, если верны оба выражения.

- `expression1 -o expression2`

Булевский оператор OR (ИЛИ). Верен, если верно любое из двух выражений.

Такие же условные выражения используются и в операторах while и until, которые мы рассмотрим чуть ниже.

**Оператор *case***

Формат оператора case таков:

```
case word in [ [(] pattern [ | pattern ] ... ) list ;; ] ... esac
```

Команда case вначале производит раскрытие слова word, и пытается сопоставить результат с каждым из образцов pattern поочередно. После нахождения первого совпадения

дальнейшие проверки не производятся, выполняется список команд, стоящий после того образца, с которым обнаружено совпадение. Значение, возвращаемое оператором, равно 0, если совпадений с образцами не обнаружено. В противном случае возвращается значение, выдаваемое последней командой из соответствующего списка.

Каждая строка с условием должна завершаться правой (закрывающей) круглой скобкой ).Каждый блок команд, отрабатывающих по заданному условию, должен завершаться *двумя* символами точка-с-запятой ;;.

```
#!/bin/bash
echo -n " Какую оценку ты получил на экзамене?: "
     read z
     case $z in
        5) echo Отлично!!!!
         ;;
        4) echo Хорошо !
         ;;
        3) echo удовлетворительно !
         ;;
        2) echo Неудовлетворительно!
         ;;
        *) echo  !
         ;;
     esac
```

**Оператор *select***

Оператор select позволяет организовать интерактивное взаимодействие с пользователем. Он имеет следующий формат:

```
select name [ in word; ] do list ; done
```

Вначале из шаблона word формируется список слов, соответствующих шаблону. Этот набор слов выводится в стандартный поток ошибок, причем каждое слово сопровождается порядковым номером. Если шаблон word пропущен, таким же образом выводятся позиционные параметры. После этого выдается стандартное приглашение bash, и оболочка ожидает ввода строки на стандартном вводе. Если введенная строка содержит число, соответствующее одному из отображенных слов, то переменной name присваивается значение, равное этому слову. Если введена пустая строка, то номера и соответствующие слова выводятся заново. Если введено любое другое значение, переменной name присваивается нулевое значение. Введенная пользователем строка запоминается в переменой REPLY. Список команд list выполняется с выбранным значением переменной name.

```
#!/bin/bash
echo "Какую ОС Вы предпочитаете?"
select var in "Linux" "Windows" "Free BSD" "Other"; do
break
done
echo "Вы бы выбрали $var"
```

Если сохранить этот текст в файле, сделать файл исполняемым и запустить, на экран

будет выдан следующий запрос:

Какую ОС Вы предпочитаете?
1) Linux
2) Windows
3) Free BSD
4) Other
#?

Нажмите любую из 4 предложенных цифр (1,2,3,4). Если вы, например, введете 1, то увидите собщение:

“Вы бы выбрали Linux”

**Оператор *for***

Оператор for работает немного не так, как в обычных языках программирования. Вместо того, чтобы организовывать увеличение или уменьшение на единицу значения некоторой переменной при каждом проходе цикла, он при каждом проходе цикла присваивает переменной очередное значение из заданного списка слов. В целом конструкция выглядит примерно так:

```
for name in words do list done.
```

Правила построения списков команд (`list`) такие же, как и в операторе `if`.

Пример. Следующий скрипт создает файлы `foo_1, foo_2 и foo_3`:

```
for a in 1 2 3 ; do
touch foo_$a
done
```

В общем случае оператор for имеет формат:

```
for name [ in word; ] do list ; done
```

Вначале производится раскрытие слова word в соответствии с правилами раскрытия выражений, приведенными выше. Затем переменной name поочередно присваиваются полученные значения, и каждый раз выполняется список команд list. Если "in word" пропущено, то список команд list выполняется один раз для каждого позиционного параметра, который задан.

В Linux имеется программа seq, которая воспринимает в качестве аргументов два числа и выдает последовательность всех чисел, расположенных между заданными. С помощью этой команды можно заставить for в bash работать точно так же, как аналогичный оператор работает в обычных языках программирования. Для этого достаточно записать цикл for следующим образом:

```
for a in $( seq 1 10 ) ; do
cat file_$a
done
```

Эта команда выводит на экран содержимое 10-ти файлов: "file_1", ..., "file_10".

**Операторы *while* и *until***

Оператор `while` работает подобно if, только выполнение операторов из списка list2 циклически продолжается до тех пор, пока верно условие, и прерывается, если условие не верно. Конструкция выглядит следующим образом:

```
while list1 do list2 done.
```

```
Пример:
while [ -d mydirectory ] ; do
ls -l mydirectory >> logfile
echo -- SEPARATOR -- >> logfile
sleep 60
done
```

Такая программа будет протоколировать содержание каталога "mydirectory" ежеминутно до тех пор, пока директория существует.

Оператор until аналогичен оператору while:

```
until list1 do list2 done.
```

Отличие заключается в том, что результат, возвращаемый при выполнении списка операторов list1, берется с отрицанием: list2 выполняется в том случае, если последняя команда в списке list1 возвращает ненулевой статус выхода.

## Функции

### Синтаксис

Оболочка bash позволяет пользователю создавать собственные функции. Функции ведут себя и используются точно так же, как обычные команды оболочки, т. е. мы можем сами создавать новые команды. Функции конструируются следующим образом:

```
function name () { list }
```

Причем слово function не обязательно, name определяет имя функции, по которому к ней можно обращаться, а тело функции состоит из списка команд list, находящегося между { и }. Этот список команд выполняется каждый раз, когда имя name задано как имя вызываемой команды. Отметим, что функции могут задаваться рекурсивно, так что разрешено вызывать функцию, которую мы задаем, внутри нее самой.

Функции выполняются в контексте текущей оболочки: для интерпретации функции новый процесс не запускается (в отличие от выполнения скриптов оболочки).

### Аргументы

Когда функция вызывается на выполнение, аргументы функции становятся *позиционными параметрами (positional parameters)* на время выполнения функции. Они именуются как $n, где n — номер аргумента, к которому мы хотим получить доступ. Нумерация аргументов начинается с 1, так что $1 — это первый аргумент. Мы можем также получить все аргументы сразу с помощью $*, и число аргументов с помощью $#. Позиционный параметр 0 не изменяется.

Если в теле функции встречается встроенная команда return, выполнение функции прерывается и управление передается команде, стоящей после вызова функции. Когда выполнение функции завершается, позиционным параметрам и специальному параметру # возвращаются те значения, которые они имели до начала выполнения функции.

### Локальные переменные (local)

Если мы хотим создать локальный параметр, можно использовать ключевое слово local. Синтаксис ее задания точно такой же, как и для обычных параметров, только определению предшествует ключевое слово `local: local name=value.`

Вот пример задания функции, реализующей упоминавшуюся выше команду seq:

```
seq()
{
```

```
local I=$1;
while [ $2 != $I ]; do
{
echo -n "$I ";
I=$(( $I + 1 ))
};
done;
echo $2
}
```

**Функция вычисления факториала *fact***

Еще один пример:

```
fact()
{
if [ $1 = 0 ]; then
echo 1;
else
{
echo $(( $1 * $( fact $(( $1 – 1 )) ) ))
};
fi
}
```

Это функция факториала, пример рекурсивной функции. Обратите внимание на арифметическое расширение и подстановку команд.

# Scilab

**Scilab** – это свободно распространяемая система компьютерной математики, которая предназначена для выполнения инженерных и научных вычислений, таких как:

- решение нелинейных уравнений и систем;
- решение задач линейной алгебры;
- решение задач оптимизации;
- дифференцирование и интегрирование;
- задачи обработка экспериментальных данных (интерполяция и аппроксимация, метод наименьших квадратов);
- решение обыкновенных дифференциальных уравнений и систем.

Scilab предоставляет широкие возможности по созданию и редактированию различных видов графиков и поверхностей.
Система имеет достаточно мощный собственный язык программирования высокого уровня.

Scilab был разработан в 1994 году во Франции, в Национальном исследовательском институте информатики и автоматизации (Institut national de recherche en informatique et en automatique, INRIA) и Национальной школе дорожного ведомства (École Nationale des Ponts et Chaussées, ENPC). С 2003 года поддержкой Scilab занимается консорциум Scilab Consortium.

Отличительные особенности пакета:

- Бесплатность
- Маленький размер (13Мб против более чем двухгигабайтного пакета MATHLAB )
- Возможность запуска в консоли без использования графического интерфейса. Это позволяет производить автоматизированные вычисления.

Пакет свободно распространяется через Интернет вместе с исходными кодами и снабжен обширной документацией. Scilab свободно распространяется вместе с исходными кодами. Использование, копирование, изменение, распространение - свободные. Пакет защищен специальной лицензией, основное отличие которой от стандартной GNU лицензии, по утверждению авторов, определяется стремлением избежать появления клонов. Существенным является то, что пакет Scilab работает в UNIX (включая Linux) и Windows. Включен в стандартную поставку SuSE.

Пакет Scilab является средой создания законченных приложений, включающую в себя реализацию численного метода и визуализацию результата.

Пакет Scilab применим для инженерных расчетов и при этом прост в обращении, имеет интерфейс, систему помощи и возможность программирования, использования русского языка, обширную библиотеку алгоритмов базовой математики. Возможно проведение вычислений как в численном, так и в формульном виде. Для визуализации результатов существует графическая библиотека с возможностью интерактивного вмешательства. Имеется возможность программирования. Особо важным являются наличие интерфейса с откомпилированными функциями языков Си и Фортран, а также существование пакета Scilab//, обеспечивающий возможность параллельных вычислений.

Имеется возможность расширения возможностей программы внешними программами и модулями, написанными на разных языках программирования. Программа имеет открытый исходный код (не GPL) и позволяет использование её как в персональных, так и коммерческих целях (с разрешения INRIA)

Свободно распространяемую версию пакета вместе с полной документацией на английском языке в формате pdf можно получить по адресу http://www.scilab.org.

**Доступные инструменты**:

- 2D и 3D графики, анимация
- Линейная алгебра, разреженные матрицы (sparse matrices)
- Полиномиальные и рациональные функции
- Интерполяция, аппроксимация
- Симуляция: решение ОДУ и ДУ
- Scicos: гибрид системы моделирования динамических систем и симуляции
- Дифференциальные и не дифференциальные оптимизации
- Обработка сигналов
- Параллельная работа
- Статистика
- Работа с СКА (системами компьютерной алгебры - такие программные продукты, как Maple, Maxima, Reduce, Derive и другие.)

Scilab имеет схожий с MATHLAB язык программирования, в составе имеется утилита, позволяющая конвертировать документы Matlab → Scilab.

Scilab позволяет работать с элементарными и большим числом специальных функций (Бесселя, Неймана, интегральные функции), имеет мощные средства работы с матрицами, полиномами (в том числе и символьно), производить численные вычисления (например численное интегрирование) и решение задач линейной алгебры, оптимизации и симуляции, мощные статистические функции, а также стредство для построения и работы с графиками.

Scilab состоит из 3-х частей:

1. интерпретатор
2. библиотека функций (Scilab-процедуры)
3. библиотека Fortran и C процедур

## Основы работы в Scilab

### Текстовые комментарии

*Текстовый комментарий* в **Scilab** это строка, начинающаяся с символов //.

Использовать текстовые комментарии можно как в рабочей области, так и в тексте файла-сценария. Строка после символов // не воспринимается как команда и нажатие клавиши **Enter** приводит к активизации следующей командной строки.

```
-->//6+8

-->
```

### Элементарные математические выражения

Для выполнения *простейших арифметических операций* в Scilab применяют следующие операторы:+ сложение, – вычитание, * умножение, / деление слева направо, \ деление справа

налево, ^ возведение в степень.

Вычислить значение арифметического выражения можно, если ввести его в командную строку и нажать клавишу ENTER. В рабочей области появится результат.

```
--> 2.35*(1.8-0.25)+1.34^2/3.12
ans =
4.2180
```

Если вычисляемое выражение *слишком длинное*, то перед нажатием клавиши ENTER следует набрать три или более точек. Это будет означать продолжение командной строки.

```
--> 1+2+3+4+5+6....
+7+8+9+10+....
+11+12+13+14+15
ans =
  120
```

Если символ точки с запятой «;» указан в конце выражения, то результат вычислений не выводится, а активизируется следующая командная строка.

```
--> 1+2;
--> 1+2
ans =
3
```

В рабочей области **Scilab** можно определять *переменные*, а затем использовать их в выражениях.

Любая переменная до использования в формулах и выражениях должна быть определена. Для определения переменной необходимо набрать имя переменной, символ «=» и значение переменной. Здесь знак равенства – это *оператор присваивания*, действие которого не отличается от аналогичных операторов языков программирования. То есть, если в общем виде оператор присваивания записать как

```
имя переменной = значение выражения
```

то в *переменную*, имя которой указано слева, будет записано *значение выражения*, указанного справа.

Имя переменной не должно совпадать с именами встроенных процедур, функций и встроенных переменных системы и может содержать до 24 символов. Система различает большие и малые буквы в именах переменных. То есть ABC, abc, Abc, aBc – это имена разных переменных.Выражение в правой части оператора присваивания может быть числом, арифметическим выражением, строкой символов или символьным выражением. Если речь идет о символьной или строковой переменной, то выражение в правой части оператора присваивания следует брать в одинарные кавычки.

Если символ «;» в конце выражения отсутствует, то в качестве результата выводится имя переменной и ее значение. Наличие символа «;» передает управление следующей командной строке. Это позволяет использовать имена переменных для записи промежуточных результатов в память компьютера

Для *очистки значения переменной* можно применить команду

```
clear имя переменной;
```

Команда `clear;` отменяет определения всех переменных данной сессии.

**Системные переменные Scilab**

Если команда не содержи знака присваивания, то по умолчанию вычисленное значение

присваивается специальной *системной переменной* `ans`. Причем полученное значение можно использовать в последующих вычислениях, но важно помнить, что значение `ans` изменяется после каждого вызова команды без оператора присваивания.

```
--> 25.7-3.14
ans =
22.5600
--> //Значение системной переменной равно 22.5600
--> 2*ans
ans =
45.1200
```

Результат последней операции без знака присваивания хранится в переменной `ans`.

Другие *системные переменные* в Scilab начинаются с символа **`%:`**

- `%i` – мнимая единица ( $\sqrt{-1}$ );
- `%pi` – число π (3.141592653589793);
- `%e` – число e=2.7182818;
- `%inf` – машинный символ бесконечности (∞);
- `%NaN`– неопределенный результат (0/0, ∞/∞, $1^{\infty}$ и т.п.);
- `%eps` – условный ноль %eps=2.220E-16.

Все перечисленные переменные можно использовать в математических выражениях.

**Пример** вычисления выражения $F=\cos(\pi/3)+(a-b)\cdot e^2$.

```
-->a=5.4;b=0.1;
-->F=cos(%pi/3)+(a-b)*%e^2
 F  =
  39.661997
```

**Пример** неверного обращения к системной переменной.

```
-->sin(pi/2)
      !--error 4
undefined variable : pi
```

Восемь значащих цифр – это формат вывода вещественного числа по умолчанию. Для того, чтобы контролировать количество выводимых на печать разрядов применяют команду `printf` с заданным форматом, который соотвествует правилам принятым для этой команды в языке С:

```
-->printf("%1.12f",%pi)
3.141592653590
-->printf("%1.15f",%pi)
3.141592653589793
-->printf("%1.2f",q)
123.46
-->printf("%1.10f",q)
123.4567890123
-->//По умолчанию 6 знаков после запятой
-->printf("%f",q)
123.456789
```

**Функции в Scilab**

Все функции, используемые в Scilab, можно разделить на два класса:

- встроенные;
- определенные пользователем.

В общем виде обращение к функции в Scilab имеет вид:

```
имя_переменной = имя_функции(переменная1 [, переменная2, …])
```

где

- имя_переменной – переменная, в которую будут записаны результаты работы функции; этот параметр может отсутствовать, тогда значение, вычисленное функцией будет присвоено системной переменной ans ;
- имя_функции – имя встроенной функции или ранее созданной пользователем;
- переменная1, переменная2, … – список аргументов функции.

**Элементарные математические функции**

Пакет Scilab снабжен достаточным количеством всевозможных *встроенных функций*.

| Функция | Описание функции |
|---|---|
| **Тригонометрические** | |
| `sin(x)` | синус числа x |
| `cos(x)` | косинус числа x |
| `tan(x)` | тангенс числа x |
| `cotg(x)` | котангенс числа x |
| `asin(x)` | арксинус числа x |
| `acos(x)` | арккосинус числа x |
| `atan(x)` | арктангенс числа x |
| **Экспоненциальные** | |
| `exp(x)` | Экспонента числа x |
| `log(x)` | Натуральный логарифм числа x |
| **Другие** | |
| `sqrt(x)` | корень квадратный из числа x |
| `abs(x)` | модуль числа x |
| `log10(x)` | десятичный логарифм от числа x |
| `log2(x)` | логарифм по основанию два от числа x |

С функциями, определяемыми пользователем, ознакомимся позже, после рассмотрения таких структур данных, как матрицы, и основных управляющих операторов.

# Массивы и матрицы в Scilab

Массив – представляет собой упорядоченный набор однотипных данных, объединенных одним именем.

Массив состоит из пронумерованной последовательности элементов. Номера в этой последовательности называются индексами.

Переменную, представляющую собой просто список данных, называют *одномерным* массивом или *вектором*. Для доступа к данным, хранящимся в определенном элементе массива, необходимо указать имя массива и порядковый номер этого элемента *(индекс)*.

Если возникает необходимость хранения данных в виде таблиц, в формате строк и столбцов, то необходимо использовать *двумерные* массивы (*матрицы*). Для доступа к данным, хранящимся в таком массиве, необходимо указать имя массива и *два индекса*, первый должен соответствовать номеру строки, а второй номеру столбца в которых хранится необходимый элемент.

*Значение нижней границы* индексации в `Scilab` равно единице. Индексы могут быть только целыми положительными числами.

Ввод и формирование массивов и матриц

Самый простой способ *задать одномерный массив* в Scilab имеет вид

```
[name]=Xn:dX:Xk
```

где `name` – имя переменной, в которую будет записан сформированный массив, `Xn` – значение первого элемента массива, `Xk` – значение последнего элемента массива, `dX` – шаг, с помощью которого формируется каждый следующий элемент массива, то есть значение второго элемента составит `Xn+dX`, третьего `Xn+ dX+dX` и так далее до `Xk`.

Возможен и такой способ *определения одномерного массива*:

```
[name]=Xn:Xk
```

Если параметр `dX` в конструкции отсутствует, это означает, что по умолчанию он принимает значение равное единице, то есть каждый следующий элемент массива равен значению предыдущего плюс один.

Переменную заданную как массив можно использовать в арифметических выражениях и в качестве аргумента математических функций. Результатом работы таких операторов являются массивы

```
--> Xn=-3.5;dX=1.5;Xk=4.5;
--> X=Xn:dX:Xk
X =
-3.5000 -2.0000 -0.5000 1.0000 2.5000 4.0000
--> Y=sin(X/2)
Y =
-0.9840   -0.8415   -0.2474   0.4794   0.9490   0.9093
--> A=0:5
```

```
A =
0 1 2 3 4 5
--> 0:5
ans =
0  1    2    3    4    5
```

Еще один способ задания векторов и матриц в Scilab это их *поэлементный ввода*.

Так для *определения вектор–строки* следует ввести имя массива, а затем после знака присваивания, в квадратных скобках через пробел или запятую перечислить элементы массива:

[name]=$x_1$ $x_2$ … $x_n$ или [name]=$x_1$, $x_2$, …, $x_n$

```
--> V=[1 2 3 4 5]
V =
1 2 3 4 5
--> W=[1.1,2.3,-0.1,5.88]
W =
1.1000 2.3000 -0.1000 5.8800
```

Элементы *вектора–столбца* вводятся через точку с запятой:

[name]=$x_1$; $x_2$; …; $x_n$

```
--> X=[1;2;3]
X =
1
2
3
```

Обратиться к элементу вектора можно, указав имя массива и порядковый номер элемента в круглых скобках: name(индекс).

Ввод элементов *матрицы* так же осуществляется в квадратных скобках, при этом элементы строки отделяются друг от друга пробелом или запятой, а строки разделяются между собой точкой с запятой:

[name]=[$x_{11}$, $x_{12}$, …, $x_{1n}$; $x_{21}$, $x_{22}$, …, $x_{2n}$;…; $x_{m1}$, $x_{m2}$, …, $x_{mn}$;

Обратиться к *элементу матрицы* можно, указав после имени матрицы, в круглых скобках, через запятую, номер строки и номер столбца на пересечении которых элемент расположен: name(индекс1, индекс2).

Кроме того, матрицы и векторы можно *формировать*, составляя их из ранее заданных матриц и векторов (листинг 3.6).

```
--> v1=[1 2 3]; v2=[4 5 6]; v3=[7 8 9];
--> //Горизонтальная конкатенация векторов-строк.
--> V=[v1 v2 v3]
```

```
  V =
  1  2    3    4    5    6    7    8    9
  --> //Вертикальная конкатенация векторов-строк, результат
матрица
  --> V=[v1; v2; v3]
  V =
  1  2    3
  4  5    6
  7  8    9
```

**Знак двоеточия «:»**

Указывая его вместо индекса при обращении к массиву, можно иметь доступ к группам его элементов

**Действия над матрицами**

Для работы с матрицами и векторами в Scilab предусмотрены следующие операции:

- + – сложение;
- – – вычитание;
- ’ – транспонирование;
- * – матричное умножение;
- * – умножение на число;
- ^ – возведение в степень;
- \ – левое деление, $(\boldsymbol{A}\backslash\boldsymbol{B}) \Rightarrow (\boldsymbol{A}^{-1}\cdot\boldsymbol{B})$, операция может быть применима для решения уравнения матричного уравнения вида $\boldsymbol{A}\cdot\boldsymbol{X}=\boldsymbol{B}$;
- / – правое деление, $(\boldsymbol{B}/\boldsymbol{A}) \Rightarrow (\boldsymbol{B}\cdot\boldsymbol{A}^{-1})$, используют для *решения матричных уравнений* вида $\boldsymbol{X}\cdot\boldsymbol{A}=\boldsymbol{B}$;
- .* – поэлементное умножение матриц;
- .^ – поэлементное возведение в степень;
- .\ – поэлементное левое деление;
- `./` – поэлементное правое деление;

Кроме того, если к некоторому заданному вектору или матрице применить *математическую функцию*, то результатом будет новый вектор или матрица той же размерности, но элементы будут преобразованSы в соответствии с заданной функцией

Специальные матричные функции

Для работы с матрицами и векторами в Scilab существуют специальные функции. Рассмотрим наиболее часто используемые из них:

- length(X) – определяет количество элементов матрицы X, если X – вектор, его длину
- prod(X) – вычисляет произведение элементов матрицы или вектора X
- sum(X) – вычисляет сумму элементов матрицы или вектора X, кроме того с помощью этой функции можно вычислить скалярное произведение векторов
- min(X) – находит минимальный элемент матрицы или вектора X
- max(X) – находит максимальный элемент матрицы или вектора X
- mean(X) – определяет среднее арифметическое матрицы или вектора X
- sort(X) – выполняет упорядочивание массива X, если X – матрица, сортировка выполняется по столбцам
- eye(n, m) – возвращает единичную матрицу соответствующей размерности ;
- ones(n, m) – формирует матрицу, состоящую из единиц
- zeros(n, m) – возвращает нулевую матрицу соответствующей размерности
- diag(V [, k]) – возвращает квадратную матрицу с элементами V на главной диагонали или на k–й; функция diag(A [, k]), где A ранее определенная матрица, в качестве результата выдаст вектор столбец, содержащий элементы главной или k–ой диагонали матрицы A;
- rand([n, m, p, …]) – возвращает матрицу случайных чисел, rand без аргументов возвращает одно случайно число;
- cat(n, A, B, [C, …]) – объединяет матрицы A и B или все входящие матрицы A, B, C, … При n=1 – по строкам, при n=2 – по столбцам. То же что [A; B] или [A, B] );
- tril(A [, k]) – формирует из матрицы A нижнюю треугольную матрицу начиная с главной или с k–й диагонали;
- size(A) – определяет число строк и столбцов матрицы A, результатом ее работы является вектор [n, m];
- det(A) – вычисляет определитель квадратной матрицы A ;

- `trace(A)` – вычисляет след матрицы `A` (сумму элементов главной диагонали);
- `inv(A)` – возвращает матрицу обратную к `A`;
- `linsolve(A, b)` – возвращает решение системы линейных уравнений ***Ax=b***

# Программирование в Scilab

### Функции ввода-вывода в Scilab

Для организации простейшего ввода в Scilab можно воспользоваться функциями

```
x=input('title');
```

или

```
x=x_dialog('title', 'stroka');
```

Функция input выводит в командной строке Scilab подсказку `title` и ждет пока пользователь введет значение, которое в качестве результата возвращается в переменную `x`. Функция `x_dialog` выводит на экран диалоговое окно, после чего пользователь может щелкнуть OK и тогда `stroka` вернется в качестве результата в переменную x, либо ввести новое значение вместо `stroka`, которое и вернется в качестве результата в переменную `x`.

Функция input преобразовывает введенное значение к числовому типу данных, а функция `x_dialog` возвращает строковое значение. Поэтому при использовании функции `x_dialog` для ввода числовых значений, возвращаемую ею строку следует преобразовать в число с помощью функции `evstr`. Поэтому можно предложить следующую форму использования функции x_dialog для ввода числовых значений.

```
x=evstr(x_dialog('title', 'stroka'));
```

Для вывода в текстовом режиме можно использовать функцию disp следующей структуры

```
disp(b)
```

Здесь `b` – имя переменной или заключенный в кавычки текст.

### Оператор присваивания

Оператор присваивания имеет следующую структуру

a=b

здесь a – имя переменной или элемента массива, b – значение или выражение. В результате выполнения оператора присваивания переменной a присваивается значение выражения b.

### Условный оператор

Одним из основных операторов, реализующим ветвление в большинстве языков программирования, является условный оператор if. Существует обычная и расширенная формы оператора if в Scilab. Обычный if имеет вид

```
if условие
операторы1
else
```

```
операторы2
end
```

В **Scilab** для построения логических выражений могут использоваться условные операторы: (**&**, **and** – логическое **и**, |, **or** – логическое или, ~, **not** - логическое отрицание) и операторы отношения: < (меньше), **>** (больше),**==** (равно), **~=, <>** (не равно), **<=** (меньше или равно), **>=** (больше или равно).

Зачастую при решении практических задач недостаточно выбора выполнения или невыполнения одного условия. В этом случае можно, конечно, по ветке **else** написать новый оператор **if**, но лучше воспользоваться расширенной формой оператора **if.**

```
if условие1
операторы1
elseif условие2
операторы2
elseif условие 3
операторы3
…
elseif условие n
операторыn
else
операторы
end
```

**Оператор выбора**

Еще одним способом организации разветвлений является оператор альтернативного выбора следующей структуры:

```
select параметр
case значение1 then операторы1
case значение2 then операторы2
…
else операторы
end
```

**Оператор while**

Оператор цикла **while** имеет вид

```
while условие
операторы
end
```

Здесь условие – логическое выражение; операторы будут выполняться циклически, пока логическое условие истинно.

**Оператор for**

```
for x=xn:hx:xk
операторы
end
```

Здесь `x` – имя скалярной переменной – параметра цикла, `xn` – начальное значение параметра цикла, `xk` – конечное значение параметра цикла, `hx` – шаг цикла. Если шаг цикла равен 1, то `hx` можно опустить, и в этом случае оператор `for` будет таким.

```
for x=xn:xk
операторы
end
```

Выполнение цикла начинается с присвоения параметру стартового значения (x=xn). Затем следует проверка, не превосходит ли параметр конечное значение (x>xk). Если результат проверки утвердительный, то цикл считается завершенным, и управление передается следующему за телом цикла оператору. В противном случае выполняются операторы в цикле (тело цикла). Далее параметр меняет свое значение (x=x+hx). Далее снова производится проверка значения параметра цикла, и алгоритм повторяется

## Функции в Scilab

Помимо использования стандартных встроенных функций, пользователь может создавать и использовать свои функции. Для создания функции можно воспользоваться 2-мя способами.

**1. Использование оператора `deff`**

Синтаксис

```
deff('[имя1,...,имяN]=имя_функции(переменная_1,...,переменная_M)
',
            'имя=выражение;...;имя1=выражение1;...;имяN=выражениеN
')
```

где `имя1,...,имяN` – список выходных параметров (от 1 до N), то есть переменных, которым будет присвоен конечный результат вычислений, имя_функции – имя с которым эта функция будет вызываться, `переменная_1,...,переменная_M` – входные параметры (от 1 до M).

Примеры:

```
-->deff('[x]=myplus(y,z)','x=y+z')
-->deff('[x]=mymacro(y,z)',['a=3*y+1'; 'x=a*z+y'])
```

**2. Использование конструкции function**

```
function [<output_args>] = <function_name>(<input_args>)
тело функции
endfunction
```

где function_name – имя функции;

`output_args` – список выходных параметров функции;

`input_args` – список входных параметров функции.

Если вызываемая функция находится не в текущем файле, то перед ее вызовом следует загрузить файл, в котором находится функция `exec('file',-1)`.

Все имена переменных внутри функции, а так же имена из списка входных и выходных параметров воспринимаются системой как *локальные*, то есть эти переменные считаются определенными только внутри функции.

Вообще говоря, функции в Scilsb играют роль *подпрограмм*. Поэтому целесообразно набирать их тексты в редакторе и сохранять в виде отдельных файлов. Причем имя файла должно обязательно совпадать с именем функции. Расширение файлам-функциям обычно присваивают `*.sci` или `*.sce`.

## Работа с файлами в Scilab

### Открытие файла

`[fd,err]=mopen(file, mode)`, где

`err` - индикатор ошибки;

`fd` - параметр `fd`, возвращаемый функцией mopen используется как файловый идентификатор

`file` – путь к файлу;

`mode` – режим открытия файла. сушествуют следующие режимы:

`r` или `rb`: чтение (файл должен существовать);

`w` или `wb`: запись(если файл не существовал, то он создается, если существовал, то предыдущее содержимое удаляется;.

`a` или `ab`: добавление в конец файла (если файл не существовал, то он создается).

`r+` или `r+b`: чтение и запись(файл должен существовать)

`w+` или `w+b`: чтение и запись (принцип работы как в `w` и `wb`).

`a+` или `a+b`: чтение и запись (принцип работы как в `a` и `ab`).

### Закрытие файла

`mclose([fd])`

С помощью функции `mclose('all')` можно закрыть сразу все открытые файлы, кроме стандартных системных файлов.

Если идентиыикатор фаула опущен, то закрывается последний открытый файл.

### Запись в файл

**Функция `mfprintf`**

mfprintf(fd, format, s), где

format – форматная строка.

s – список выводимых параметров.

В форматной строке указываются форматы вывода параметров:

%[ширина][.точность]тип.

**Тип**

c При вводе символьный тип char, при выводе один байт.

d,i Десятичное со знаком

i Десятичное со знаком

o Восьмеричное int unsigned

u Десятичное без знака

x, X Шестнадцатеричное int unsigned, при x используются символы a-f, при X – A-F.

f Значение со знаком вида [-]dddd.dddd

e Значение со знаком вида [-]d.ddde[+|-]ddd

E Значение со знаком вида [-]d.ddddE[+|-]ddd

g Значение со знаком типа e или f в зависимости от значения и точности

G Значение со знаком типа E или F в зависимости от значения и точности

s Строка символов

**Флаги**

- Выравнивание числа влево. Правая сторона дополняется пробелами. По умолчанию выравнивание вправо.

+ Перед числом выводится знак «+» или «-»

Пробел Перед положительным числом выводится пробел, перед отрицательным – «-»

# Выводится код системы счисления: 0 – перед восьмеричным числом, 0x (0X) пе-ред шестнадцатеричным числом.

**Ширина**

n Ширина поля вывода. Если n позиций недостаточно, то поле вывода расширяется до минимально необходимого. Незаполненные позиции заполняются пробелами.

0n То же, что и n, но незаполненные позиции заполняются нулями.

**Точность**

ничего Точность по умолчанию

n Для типов e, E, f выводить n знаков после десятичной точки

**Модификатор**

h Для d, i, o, u, x, X короткое целое

l Для d, i, o, u, x, X длинное целое

В строке вывода могут использоваться некоторые специальные символы

\b Сдвиг текущей позиции влево

\n Перевод строки

\r Перевод в начало строки, не переходя на новую строку

\t Горизонтальная табуляция

\' Символ одинарной кавычки

\'' Символ двойной кавычки

\? Символ ?

**Функция mput**

mput(x [,type,fd]), где

x: число или вектор

fd: дескриптор файла.

type: формат записи числа:

"l","i","s","ul","ui","us","d","f","c","uc": соответственно long, int, short, unsigned long, unsigned int, unsigned short, a double, float, char и unsigned char.

**Запись матрицы в файл – функция write**

write(filename,a,[format])

a: матрица.

**Чтение из файла**

**Функция mfscanf**

A=mfscanf(fd, s), где

Из файла с идентификатором fd считываются в переменную A значения в соответствии с форматом s. При чтении числовых значений из текстового файла следует помнить, что два числа считаются разделенными, если между ними есть хотя бы один пробел, символ табуляции или символ перехода на новую строку.

**Функция mget**

x=mget([n,type,fd]), где

x – вектор ли число;

n – число считываемых параметров;

type – формат числа(см mput)

**Чтение матрицы – функция read**

[x]=read(filename,m,n,[format]), где

m и n – размерности матрицы. m=-1 если заранее неизвестно количество строк матрицы

**Функция определения конца файла**

```
e=meof(fd)
```

Функция определяет, достигнут ли конец файла

## Создание приложений в Scilab

### Создание графического окна

Для создания пустого графического окна служит функция figure.

```
F=figure();
```

В результате выполнения этой команды будет создано данном графическое окно с именем `objfigure1`. По умолчанию первое окно получает имя `objfigure1`, второе – `objfigure2` и т.д. Указатель на графическое окно записывается в переменную `F`. Размер и положение окна на экране монитора можно задавать с помощью параметра `'position',[x y dx dy]`, где

- `x, y` - положение верхнего левого угла окна (по горизонтали и вертикали соответственно) относительно верхнего левого угла экрана;
- `dx` - размер окна по горизонтали (ширина окна) в пикселях;
- `dy` - размер окна по вертикали (высота окна) в пикселях.

Параметры окна можно задавать одним из двух способов.

1. Непосредственно при создании графического окна задаются его параметры. В этом случае обращение к функции `figure` имеет вид

   ```
   F=figure('Свойство1', 'Значение1', 'Свойство2', 'Значение2', ...,'Свойствоn', 'Значениеn')
   ```

   здесь `'Свойство1'` – название первого параметра, `Значение1` – его значение, `'Свойство2'` – название второго параметра, `Значение2` – значение второго параметра и т.д.

   Например,

   ```
   F=figure('position', [10 100 300 200]);
   ```

2. После создания графического окна с помощью функции `set(f,'Свйоство','Значение')` устанавливается значение параметров, здесь `f` - указатель на графическое окно, `'Свойство'` – имя параметра, `'Значение'` – его значение.

```
f=figure();
set(f,'position',[20,40,600,450])
```

Для изменения заголовка окна используется параметр `'figure_name', 'name'` определяющий заголовок окна (`'name'`).

```
f=figure();
set(f,'position',[20,40,600,450]);
set(f,'figure_name','FIRST WINDOW');
```

```
f=figure('position',[20,40,600,450],'figure_name','FIRST WINDOW');
```

Графическое окно можно закрыть с помощью функция `close(f)` (здесь `f` – указатель на окно). Удаляется окно с помощью функции `delete(f)`, где `f`- указатель на окно.

**Создание объектов (функция `uicontrol`)**

В Scilab используется динамический способ создания интерфейсных компонентов. Он заключается в том, что на стадии выполнения программы могут создаваться (и удаляться) те или иные графические объекты (кнопки, метки, флажки и т.д.) и их свойствам присваиваются соответствующие значения.

```
C=uicontrol(F, 'Style', 'тип_компонента', 'Свойство_1',
Значение_1, 'Свойство_2', Значение_2,... 'Свойство_k',Значение_к);
```

где `C` – указатель на создаваемый компонент;

`F` –указатель на объект, внутри которого будет создаваться компонент; первый аргумент функции `uicontrol` не является обязательным, и если он отсутствует, то родителем (владельцем) создаваемого компонента является текущий графический объект – текущее графическое окно;

`'Style'`– служебная строка `Style`, указывает на стиль создаваемого компонента(символьное имя);

'тип_компонента'– определяет, к какому классу принадлежит создаваемый компонент, это может быть `PushButton`, `Radiobutton`, `Edittext`, `StaticText`, `Slider`, `Panel`, `Button Group`, `Listbox` или др компоненты;

'`Свойство_к`', `Значение_к` – определяют свойства и значения отдельных компонентов, они будут описаны ниже конкретно для каждого компонента.

У существующего интерфейсного объекта можно изменить те или иные свойства с помощью функции `set`:

```
set(C,'Свойство_1',Значение_1,...)
```

где `C` – указатель на компонент, параметры которого будут меняться;

Получить значение параметра компонентов можно с помощью функции `get` следующей структуры:

```
get(C,'Свойство')
```

где `C` – указатель на динамический интерфейсный компонент, значение параметра которого необходимо узнать;

'`Свойство`'- имя параметра, значение которого нужно узнать.

Функция возвращает значение параметра. Далее мы поговорим об особенностях создания различных компонентов.

**Типы компонентов:**

`pushbutton` – кнопка;

`text` – текстовое поле для отображения текстовой информации;

`edit` – окно редактирования;

`radiobutton` – кнопка со значением on или off – переключатель;

`checkbox` – флажок;

`listbox` – список строк.

**Свойства компонентов:**

`string` – заголовок;

`position` – координаты расположения компонента;

`callback` – обработка события (например, при нажатии на кнопку);

`BackgroundColor` – цвет фона (значением является ибо строка, либо вектор, состоящий из трех величин типа real. В случае, если это строка, т значения компонент цвета разделяются с помощью знака «|»: «R|G|B». Каждая величина предствляет значение в интервале [0,1]);

`Horizontalalignment` – выравнивание текста (используется в компонентах `'text'`, `'edit'` и `'checkbox'`, значения: `left` – выравнивание по левому краю; `center` – выравнивание текста по центру (значение по умолчанию); `right` – выравнивание по правому краю).

**Пример создания текстового поля.**

```
w=figure('Position',[50,50,200,200]);
t=uicontrol('Style','text','Position',[100,100,50,20],'String',
'0.1');
set(t,'BackgroundColor',[1 1 1]);
set(t,'HorizontalAlignment','left');
```

**Пример создания кнопки**

```
w=figure();
pbtn=uicontrol(w,'Style','pushbutton',      'string',      'OK',
'Callback', 'sinus');

function y=sinus()
x=-5*%pi:0.2*%pi:5*%pi;
y=sin(x);
plot(x,y);

endfunction
```

**Пример создания переключателя**

```
hFig=figure('Position',[50,50,200,200]);
```

```
  //Создание радиокнопок
  hRb1=uicontrol('Style','radiobutton','String','sin(x)','value',
0, 'Position',[25,100,60,20],'callback','Radio1');
  hRb2=uicontrol('Style','radiobutton','String','cos(x)','value',
0, 'Position',[25,140,60,20],'callback','Radio2');

  //обработчик нажатия на кнопки
  function Radio1()
  newaxes;
  x=-2*%pi:0.1:2*%pi;
  set(hRb2,'value',0);
  y=sin(x);
  plot(x,y);
  xgrid();
  endfunction

  function Radio2()
  newaxes;
  x=-2*%pi:0.1:2*%pi;
  set(hRb1,'value',0);
  y=cos(x);
  plot(x,y);
  xgrid();
  endfunction
```

**Пример создания списка строк**

```
f=figure();
h=uicontrol(f,'style','listbox','position', [10 10 150 160]);
set(h, 'string', "item 1|item 2|item3");
set(h, 'value', [1 3]);
```

# Язык программирования Pascal

## История

Pascal разрабатывался с 1968 по 1970 г. Николаусом Виртом. Цель заключалась в том, чтобы создать язык, лишенный многочисленных недостатков ALGOL. Pascal был назван в честь французского математика Блеза Паскаля, который еще в 1642 г. изобрел цифровой калькулятор. С конца 70-х до конца 80-х гг. этот язык доминировал среди языков, используемых на начальном этапе обучения программированию; позже его заменили C и C++, а затем Java.

ALGOL 60 был первой попыткой создания языка на основе формального описания, однако его реализация оказалась сложной. В частности, оказалось достаточно трудно реализовать передачу параметров по имени, хотя это довольно элегантный механизм. В языке ALGOL 60 не были определены операторы ввода-вывода, поскольку в то время считалось, что они зависят от реализации, да и собственную статическую память также трудно было реализовать. Помимо того, в 60-х гг. были разработаны новые практические решения, например типы данных и структурное программирование. Языки типа FORTRAN были популярны благодаря своей эффективности при выполнении программ, несмотря на отсутствие элегантности.

В 1965 г., во время работы в Стенфордском университете (Stanford University), Вирт разработал новую, расширенную версию ALGOL 60 для компьютеров серии IBM 360, в которую вошло определение указателей и структур данных. Этот язык, известный как ALGOL W, использовался в нескольких университетах, но его реализация ограничивалась только компьютерами IBM 360. Для выполнения программ на этом языке требовался значительный по размерам пакет программ поддержки обработки строк, вещественных чисел двойной точности и других сложных типов данных. Таким образом, ALGOL W в качестве системного языка программирования оказался малоэффективным.

В 1968 г. Вирт вернулся в Швейцарию и начал работу над преемником ALGOL W - языком, который мог бы компилироваться за один проход. Для создания исходного компилятора был использован алгоритм рекурсивного спуска. Этот компилятор выполнялся на компьютере Control Data. Также был разработан широко известный теперь интерпретатор P-кода. Компилятор языка Pascal сначала транслировал исходную программу в программу на языке гипотетической машины со стековой архитектурой. Благодаря такой своей организации Pascal легко переносился на компьютеры других систем. Компилятор Pascal был написан на одноименном языке. Все, что требовалось для перехода в другую систему, - это переписать соответствующим образом интерпретатор P-кода. Появившийся в 1970 г. Pascal начал завоевывать признание. В 1983 г. был разработан американский стандарт языка (IEEE 770/ ANSI X3.97), а вскоре был разработан стандарт ISO (ISO 7185).

## Пример простейшей программы на Pascal

```
  program Hello;
begin
writeln('Hello, world!');
readln;
end.
```

## Структура программы на Pascal

Любая программа на Pascal состоит из трех блоков: блока объявлений, блока описания процедур и функций и блока операторов (основной блок программы)

Блок объявлений:

program ... (название программы)

uses ... (используемые программой внешние модули)

const ... (подраздел описания констант)

type ... (подраздел объявления типов)

var ... (подраздел объявления переменных)

Блок описания процедур и функций:

procedure (function)

begin

...

end;

...

Блок основной программы:

begin

(операторы основной программы) ...

end;

выработать практические навыки работы с системой Borland Pascal, научиться создавать, вводить в компьютер, выполнять и исправлять простейшие программы на языке Pascal в режиме диалога, познакомиться с диагностическими сообщениями компилятора об ошибках при выполнении программ, реализующих линейные алгоритмы.

**Раздел `program`**

состоит из зарезервированного слова PROGRAM и имени программы. В Турбо Паскале эта строка не обязательна, и ее можно без ущерба исключить. Но правила хорошего тона в программировании требуют задания некоторого имени программы, чтобы уже при первом знакомстве можно было получить хоть какую-нибудь информацию об ее назначении.

Имя, или идентификатор, строится по следующим правилам: оно может начинаться с большой или малой буквы латинского алфа-вита или знака "_", далее могут следовать буквы, цифры или знак "_"; внутри идентификатора не может стоять пробел. После имени программы следует поставить ";", этот знак служит в Паскале для разделения последовательных инструкций. Имя программы может не совпадать с именем соответствующего файла на диске.

**Раздел описания модулей**

Раздел описания модулей определяется служебным словом **USES** и содержит имена подключаемых модулей (библиотек) как входящих в состав системы PASCAL, так и написанных пользователем.

Понятия "библиотека", "модуль", "блок" составляют основу терминологии программирования на Паскале. Библиотека включает набор модулей, каждый из которых замкнут, имеет свое имя, компилируется отдельно и к нашей программе подключается уже как "черный ящик" с известным интерфейсом. Каждый модуль (блок (UNIT), как его называют на Паскале) представляет собой программу, включающую декларации типов и переменных, процедуры и функции.

Раздел описания модулей должен быть первым среди разделов описаний. Имена модулей отделяются друг от друга запятыми: `uses CRT, Graph`.

**Раздел описания констант**

Описание констант позволяет использовать имена как синонимы констант, их необходимо определить в разделе описаний констант:

```
CONST
   Year=2007;
   Month='november';
   Day='Saturday';
```

При присвоении значений константам вместо оператора присвоения ":=" используется просто знак равенства "=". Тип константы определяется автоматически по виду значения, присваемового константе и не может быть сложным.

**Раздел описания типов**

Раздел описания типов `type` позволяет определить новый тип в программе.

**Раздел описания переменных**

Здесь содержится список используемых в программе переменных и определеяется их тип.

Объявления переменных записываются в следующей форме: `<переменная> : <тип>;`

Если описываются несколько переменных одного типа, то достаточно записать их имена через запятую, а после двоеточия поставить общий тип.

```
var
   a,b,c: integer;
   x,y: real;
```

Переменные могут хранить данные различной природы: числа, строки тек-ста, отдельные символы и т. п.

**Блок операторов**

Основной блок программы, ограниченный операторами `begin` и `end.`. Как уже говорилось, оператор end. указывает компилятору, что программа закончена, в отличие от операторов `end;`, которые завершают блоки, процедуры, модули и т.п.
Текст, следующий за оператором `end`., игнорируется транслятором.

**Символ ";"**

Этот символ завершает каждый оператор

Наличие точки с запятой обязательно, т.к. этот символ показывает компилятору, где заканчивается один оператор и начинается следующий. Благодаря тому, что Паскаль - язык со "свободной формой записи", можно без опасений "растянуть" оператор на несколько строк. С помощью символа точка с запятой Вы сообщаете компилятору, какую часть текста

программы следует рассматривать как цельный, неделимый фрагмент. Зарезервированное слово `begin`, с которого начинаются блоки программы, не требует после себя символа точка с запятой.

## Типы данных

***Тип переменной*** задает вид того значения, которое ей присваивается и правила, по которым операторы языка действуют с переменной

Если переменные A и B целочисленного типа, то программа:

```
x:=3;
y:=2;
writeln(x,' ',y,' ',x+y);
```

Выведет на экран строку: "`3 2. 5`"

Если же они строкового типа, то программа:

```
x:='3';
y:='2';
writeln(x,' ',y,' ',x+y);
```

выведет: "`3 2 32`", так как оператор сложения просто добавит строку `y` в конец строки `x`.

***Тип константы*** определяется способом записи ее значения:

```
const
     c1=17;
     c2=3.14;
     c3='a';
     c4=false;
     c5=c2+c1;
```

При определении констант можно использовать выражения. Выражения должны в качестве операторов содержать только константы, в том числе ранее объявленные, а так же знаки математических операций, скобки и стандартные функции.

В Pascal предопределены следующие простейшие типы переменных:

| | |
|---|---|
| | **Целочисленные типы** |
| `byte` | целое число от 0 до 255, занимает одну ячейку памяти (байт). |
| `word` | целое число от 0 до 65535, занимает два байта. |
| `integer` | целое число от –32768 до 32767, занимает два байта. |
| `shortint` | целое число от -128 до 127, занимает 1 байт |
| `longint` | целое число от –2147483648 до 2147483647, занимает четыре байта. |
| | **Вещественные типы данных** |
| `real` | число с дробной частью от $2.9*10^{-39}$.до $1.7*10^{38}$, может принимать и отрицательные значения, на экран выводится с точностью до 12-го знака после запятой, если результат какой либо операции с `real` меньше, чем $2.9*10^{-39}$, он трактуется как ноль. Переменная типа `real` занимает шесть байт. |
| `single` | число с дробной частью от $1.5*10^{-45}$ до $3.4*10^{38}$, может принимать и отрицательные значения, на экран выводится с точностью до 8-го знака после запятой, если результат какой либо операции с `real` меньше, чем $1.5*10^{-45}$, он |

| | |
|---|---|
| | трактуется как ноль. Переменная типа `real` занимает шесть байт. |
| `double` | число с дробной частью от $5.0*10^{-324}$.до.$1.7*10^{308}$, может принимать и отрицательные значения, на экран выводится с точностью до 16-го знака после запятой ,если результат какой либо операции с `double` меньше, чем $5.0*10^{-324}$, он трактуется как ноль. Переменная типа `double` занимает восемь байт. |
| `char` | **Символьный тип**<br>символ, буква, при отображении на экран выводится тот символ, код которого хранится в выводимой переменной типа `char`, переменная занимает один байт. Каждому символу приписывается целое число в диапазоне от 0 до 255. Для кодировки используется код ASCII. |
| `string` | **Строковый тип**<br>строка символов, на экран выводится как строка символов, коды которых хранятся в последовательности байт, занимаемой выводимой переменной типа STRING; в памяти занимает от 1 до 256 байт – по количеству символов в строке, плюс один байт, в котором хранится длина самой строки. |
| `boolean` | **Логический тип**<br>логическое значение (байт, заполненный единицами, или нулями), `true`, или `false`. |

При обьявлении переменной строкового типа можно заранее указать ее длину в байтах – **X**:

```
MyString:string[X];
```

При присвоении этой переменной строки длиннее `X`, присваиваемая строка будет обрезана с конца после `X`-того символа.

Размер переменной типа `STRING` в памяти можно узнать следующим способом:

```
Size:=SizeOf(MyString);
```

Функция `SizeOf()` возвращает размер, занимаемый переменной, служащей параметром. Параметром может служить и тип переменной; строка:

```
writeln(SizeOf(string));
```

Выведет на экран число 256, так как по умолчанию под все строки отводится по 256 байт.

Кроме того, можно узнать, сколько символов в строке (индекс последнего непустого символа в строке):

```
Size:=Ord(MyString[0]);
```

Используется ибращение к нулевому элементу (символу) строки, в котором хранится ее длина, но `MyString[0]` – значение типа `char`, тоесть символ, код которого равен длине строки, нужный нам код – число возвращает функция `Ord()` Таким же образом можно обратиться к любому N – тому элементу строки:

```
MyChar:=MyString[N];
```

```
{MyChar:CHAR}
```

| | |
|---|---|
| `array[a..b,c..d,….] of "тип элемента";` | массив некоторой размерности, содержащий элементы указанного типа. |

Диапазоны индексов для каждого измерения указываются парами чисел или констант, разделенных двумя точками, через запятую `(a..b,c..d)`. После `OF` записывается тип элементов массива. В памяти массив занимает место, равное: `(b-a)*(d-c)*..* SizeOf("тип элемента")`. Размер массива не может превосходить 65536 байт.

Обращение к элементам массива происходит следующим образом:

```
X:=MyArray[a,b,c,..];
```

При этом переменная X должна быть того же типа, что и элементы массива или приводимого типа. Число индексов (a,b,c,..) должно быть равно числу обьявленных при описании измерений массива.

**Приводимость типов**

В Pascal существуют ограничения на присвоение значений одних переменных другим. Если переменные которую и которой присваивают одного типа, то никаких проблем не возникнет. Но если они разных типов, присвоение не всегда может быть произведено.

например:

```
x:=y; {x:integer; y:real}
a:=b; {a:char; b:string}
```

В то же время, такие присвоения будут выполнены вполне корректно:

```
y:=x;
b:=a;
```

При этом переменная y примет значение с нулевой дробной частью, `a b` – станет строкой, содержащей один символ – из `a`.

В первом же случае, можно поизвести следующие операции:

`x:=trunc(y);` {функция `trunc()` возвращает целую часть аргумента}

`x:=round(y);` {`round()` – округляет аргумент стандартным способом}

Кроме рассмотренного случая может существовать множество других, но наиболее общее правило таково: следить за однозначностью присвоения с потерями информации и не удивляться, а экспериментировать переделывать программу, если компилятор выдает сообщение о невозможности присвоения.

## Арифметические операции и стандартные функции в Pascal

**Арифметические операции**

| Операция | Действие | Тип операндов | Тип результата |
|---|---|---|---|
| бинарные | | | |
| + | сложение | целый, вещественный | целый, вещественный |
| - | вычитание | целый, вещественный | целый, вещественный |
| * | умножение | целый, вещественный | целый, вещественный |
| / | деление | целый, вещественный | вещественный |
| div | целочисленное деление | целый | целый |
| mod | остаток от деления | целый | целый |
| унарные | | | |
| + | сохранение знака | целый, вещественный | целый, вещественный |
| - | отрицание знака | целый, вещественный | целый, вещественный |

**Операции отношения**

Операции отношения выполняют сравнение двух операндов и определяют, истинно значение или ложно. Сравниваемые величины могут принадлежать к любому типу данных, и результат всегда имеет логический тип, принимая одно значение из двух: истина или ложь.

| Операция | Название | Выражение |
|---|---|---|
| = | Равно | A=B |
| <> | Неравно | A<>B |
| > | Больше | A>B |
| < | Меньше | A<B |
| >= | Больше или равно | A>=B |
| <= | Меньше или равно | A<=B |

**Стандартные математические функции**

| Обращение | Тип аргумента | Тип результата | Функция |
|---|---|---|---|
| `abs(x)` | целый, вещественный | целый, вещественный | модуль аргумента |
| `arctan(x)` | целый, вещественный | вещественный | арктангенс |
| `cos(x)` | целый, вещественный | вещественный | косинус |
| `exp(x)` | целый, вещественный | вещественный | $e^x$ - экпонента |
| `frac(x)` | целый, вещественный | вещественный | дробная часть x |
| `int(x)` | целый, вещественный | вещественный | целая часть x |
| `ln(x)` | целый, вещественный | вещественный | натуральный логарифм |
| `random` | | вещественный | псевдослучайное число [0,1] |
| `random(x)` | целый | целый | псевдослучайное число [0,x] |
| `round(x)` | вещественный | целый | округление до ближайшего целого |
| `sin(x)` | целый, вещественный | вещественный | синус |
| `sqr(x)` | целый, вещественный | вещественный | квадрат x |
| `sqrt(x)` | целый, вещественный | вещественный | корень квадратный из x |
| `trunc(x)` | вещественный | целый | ближайшее целое, не превышающее x по модулю |

**Логические операции**

Логические выражения в результате вычисления принимают логические значения True и False. Операндами это выражения могут быть логические константы, переменные, отношения. Идентификатор логического типа в Pascal: boolean.

В Паскале имеется 4 логические операции: отрицание -`not`, логическое умножение -`and`, логическое сложение - `or`, исключающее “или” - `xor` . Использованы обозначения: T (true),

F ( false).

| A | B | not A | A and B | A or B | A xor B |
|---|---|---|---|---|---|
| T | T | F | T | T | F |
| T | F | F | F | T | T |
| F | F | T | F | F | F |
| F | T | T | F | T | T |

Приоритеты операций: `not, and, or, xor`. Операции отношения ( = , <> ) имеют более высокий приоритет, чем логические операции, поэтому их следует заключать в скобки при использовании по отношению к ним логических операций.

Приоритет операций (в порядке убывания):

- вычисление функции;
- унарный минус, not;
- умножение, деление, div, mod, and;
- сложение, вычитание, or, xor;
- операции отношения

## Процедуры ввода/вывода

**`write(p1,p2,... pn);`** - выводит на экран значения выражений `p1,p2,... pn.`

Выражения могут быть числовые, строковые, символьные и логические. Под выражением будем понимать совокупность некоторых действий, применённых к переменным, константам или литералам, например: арифметические действия и математические функции для чисел, функции для обработки строк и отдельных символов, логические выражения и т.п.

Возможен форматный вывод, т. е. явное указание того, сколько выделять позиций на экране для вывода значения.

Например, для того, чтобы вывести значение выражения a+b с выделением для этого 10 позиций, из них 5 - после запятой

```
write(a+b:10:5);
```

Или например, вывести значение выражения p любого другого типа , выделив под него 10 позиций

```
write(p:10);
```

Вывод на экран в любом случае производится по правому краю выделенного поля.

**`writeln(p1,p2,... pn);`** - аналогично `write`, выводит значения `p1,p2,... pn`, после чего переводит курсор на новую строку.

Существует вариант `writeln;` (без параметров), что означает лишь перевод курсора на начало новой строки.

**`readln(v1,v2,...vn);`** - ввод с клавиатуры значений переменных `v1,...vn`. Переменные могут иметь строковый, символьный или числовой тип. При вводе следует

разделять значения пробелами, символами табуляции или перевода строки.

**read(v1,v2,...vn)**; - аналогочно readln;

## Управляющие структуры в языке Pascal

### Условный оператор

if <условие> then <оператор 1> [else <оператор 2>]

Условие – значение типа boolean или логическая операция. Если условие верно, выполняется оператор, или блок операторов, следующий за then, в противном случае выполняется блок операторов после else, если он есть.

Условия могут быть вложенными и в таком случае, любая встретившаяся часть else соответствует ближайшей к ней "сверху" части then.

### Оператор выбора одного из вариантов.

```
case Выражение of
Вариант1: Оператор1;
Вариант2: Оператор2;
ВариантN: ОператорN;
[else ОператорN1;]
end;
```

Выражение в простейших случаях может быть целочисленным или символьным. В качест-ве вариантов можно применять:

1. Константное выражение такого же типа, как и выражение после case. Константное выражение отличается от обычного тем, что не содержит переменных и вызовов функций, тем самым оно может быть вычислено на этапе компиляции программы, а не во время выполнения.

2. Интервал, например: 1..5, 'a'..'z'.

3. Список значений или интервалов, например: 1,3,5..8,10,12.

Выполняется оператор case следующим образом: вычисляется выражение после слова case и по порядку проверяется, подходит полученное значение под какой-либо вариант, или нет. Если подходит, то выполняется соответствующий этому варианту оператор, иначе - есть два варианта. Если в операторе case записана часть else, то выполняется оператор после else, если же этой части нет, то не происходит вообще ничего.

Рассмотрим пример. Пусть пользователь вводит целое число от 1 до 10, программа должна приписать к нему слово "ученик" с необходимым окончанием (нулевое, "а" или "ов").

```
program SchoolChildren;
var n: integer;
begin
     write('Число учеников --> ');
     readln(n);
```

```
     write(n,' ученик');
     case n of
          2..4: write('а');
          5..10: write('ов');
     end;
     readln;
end.
```

## Операторы циклов

**Цикл с параметром (со счетчиком)**

`for <переменная>:=<нач_значение> to <кон_значение> do <оператор>.`

Вместо `to` возможно слово `downto`. Рассмотрим такой пример: требуется вывести на экран таблицу квадратов натуральных чисел от 2 до 20.

```
var i: integer;
begin
   for i:=2 to 20 do
   writeln(i,' ',sqr(i));
end.
```

**Цикл с предусловием**

`while <условие> do <оператор>.`

пока условие истинно, выполняется оператор (в этом случае оператор может не выполниться ни разу, т.к. условие проверяется до выполнения). Под оператором здесь понимается либо простой, либо составной оператор (т.е. несколько операторов, заключённых в `begin ... end`).

**Цикл с постусловием**

`repeat <оператор> until <условие>`

Цикл работает следующим образом: выполняется оператор, затем проверяется условие, если оно пока еще не выполнилось, то оператор выполняется вновь, затем проверяется условие, и т. д. Когда условие, наконец, станет истинным выполнение оператора, расположенного внутри цикла, прекратится, и далее будет выполняться следующий за циклом оператор.

## Работа с массивами

Массив – упорядоченный набор однотипных переменных , объединенных одним именем. В качестве типа элементов массива можно использовать все типы, известные нам на дан-ный момент (к ним относятся все числовые, символьный, строковый и логический типы).Каждый элемент массива имеет свой номер (индекс). для индексов массивов подходит любой порядковый тип, то есть такой, который в па-мяти машины представляется целым числом.

Единственное ограничение состоит в том, что размер массива не должен превышать 64 Кб. Каждый элемент является переменной, т.е. обладает своим именем и значением.

Массив относится к так называемым структурированным данным, то есть таких, что имеют фиксированную внутреннюю структуру (организацию).

При обращении к отдельному элементу массива необходимо указать его индекс (местонахождение в массиве):

```
A[7]    i:=7;     A[i]
```

Здесь i - индекс элемента массива

**Объявление массива**

Массивы, как и другие переменные, должны быть объявлены в разделе var

```
var
Mas: array [1..15] Of real;
Work: array [(Mon, Tue, Wed)] Of integer;
B: array ['A'..'Z'] Of boolean;
C: array [1..3, 1..5] Of real;
```

**Ввод массива**

Чтобы заполнить массив данными существуен несколько способов:

- непосрерственное присваивание значений элементам;
- генерация и присваивание значений с помощью функции random;
- ввод значений элементов с клавиатуры;

1) Ввод элементов одномерного массива с клавиатуры:

```
var
A : array[1..20] of real;
begin
writeln('Введите элементы массива:');
for i:=1 to n do readln(A[i]);
...
```

2) Заполнение массива случайными числами.

В этом случае необходимо перезапустить генератор случайных чисел. Затем в цикле(например, в цикле с параметром, где в качестве параметра выступает индекс массива) сгенерровать значения для всех элементов.

```
randomize;
for i:=1 to n do
a[i]:=random(100);
...
```

**Двумерные массивы**

Двумерный массив можно задать 2-мя способами:

1. `Mas : array [1..3] of array [1..5] of integer;`
2. `Mas : array [1..3, 1..5] of integer;`

Ссылка на элемент матрицы `Mas`, лежащий на пересечении i-той строки ij-ого столбца выглядит следующим образом `Mas[i][j]`.

## Символьный тип данных

Тип данных, переменные которого хранят ровно один символ (букву, цифру, знак препина-ния и т.п.) называется символьным, а в Паскале — char.

Символы в компьютере сохраняются не в виде букв, а в виде чисел. Так, каждой букве, числу, вобще, любому иероглифу, способному виводится на экран, соответствует число в кодовой таблице. Кодовая таблица - это специальная система перевода чисел в начертания на мониторе.

Объявить переменную такого типа можно так: `var ch: char;`. Для того чтобы положить в эту переменную символ, нужно ис-пользовать оператор присваивания, а символ записывать в апострофах, например: `ch:='R';`. Для символьных переменных возможно также использование процедуры readln, например:

```
write('Exit? (Yes/No)'); readln(ch);
if ch='Y' then ...
else ...;
```

Символьные переменные в памяти компьютера хранятся в виде числовых кодов, иначе говоря, у каждого символа есть порядковый номер. К примеру, код пробела равен 32, код 'A' - 65, 'B' - 66, 'C' - 67, код символа '1' - 48, '2' - 49, '.' - 46 и т. п. Некоторые символы (с кодами, меньшими 32) являются управляющими, при выводе таких символов на экран происхо-дит какое либо действие, например, символ с кодом 10 переносит курсор на новую строку, с ко-дом 7 - вызывает звуковой сигнал, с кодом 8 - сдвигает курсор на одну позицию влево. Под хранение символа выделяется 1 байт (байт состоит из 8 бит, а бит может принимать значения 0 или 1), поэтому всего можно закодировать 28=256 различных символов. Кодировка символов, которая используется Турбо-Паскале, называется ASCII (American Standard Code for Information Interchange - американский стандартный код для обмена информацией).

Операции над символами

Над символами возможны операции перевода их в числовой эквивалент и обратно. Как уже говорилось, в Паскале символы связаны с числами в соответствии с кодовой таблицей ASCII. Обратите на это внимание, поскольку в Windows символы представлены в таблице ANSI, поэтому вы можете обнаружить несоответствие вашей программы, открытой в Windows, например в Блокноте, и в среде Паскаль.

Для получения ASCII-кода любого символа используется функция `ord();` . В качестве параметра записывается переменная типа char или же непосредственно в кавычках нужный символ. Например:

```
writeln( Ord('л') );
```

или

```
writeln( Ord(c3) );
```

Если в c3 была записана буква л, то на экране появится цифра 171 - ASCII-код строчной русской буквы л.

Обратное действие - получение символа по его коду делает функция `chr();`. В скобках записывается число - от 0 до 255, то есть код необходимого символа или числовая переменная. Подобная строчка

```
writeln( Chr(102) );
```

напишет нам символ f

Стоит заметить, что если вы используете в функции `chr()` не числовую переменную, а готовое число, можете вместо самой функции писать символ `'#'`, а после него - число:

```
chr(102)
```

аналогично

```
#102
```

```
program ASCII;
 var ch: char;
 begin
 for ch:=#32 to #255 do write(ord(ch),'->',ch,' ');
 readln;
 end.
```

В этой программе в качестве счётчика цикла была использована символьная переменная, это разрешается, поскольку цикл `for` может использовать в качестве счётчика переменные любого типа, значения которого хранятся в виде целых чисел.

**Сравнение символов.** Также как и числа, символы можно сравнивать на =, <>, <, >, <=, >=. В этом случае Паскаль сравнивает не сами символы, а их коды. Таблица ASCII составлена та-ким образом, что коды букв (латинских и большинства русских) возрастают при движении в алфавитном порядке, а коды цифр расположены по порядку.

Для хранения строк (то есть последовательностей из символов) в Турбо-Паскале имеется тип `string`. Значениями строковых переменных могут быть последовательности различной дли-ны (от нуля и более, длине 0 соответствует пустая строка). Объявить строковую переменную можно двумя способами: либо var `s: string`; (максимальная длина строки - 255 симво-лов), либо `var s: string[n];` (максимальная длина - `n` символов, `n` - константа или кон-кретное число).

Для того чтобы положить значение в строковую переменную используются те же приёмы, что и при работе с символами. В случае присваивания конкретной строки, это строка должна записываться в апострофах `(s:='Hello, world!')`. Приведём простейший пример со стро-ками: программа спрашивает имя у пользователя, а затем приветствует его:

```
program Hello;
 var s: string;
 begin
 write('Как Вас зовут: ');
 readln(s);
 write('Привет, ',s,'!');
 readln;
 end.
```

**Хранение строк**. В памяти компьютера строка хранится в виде последовательности из символьных переменных, у них нет индивидуальных имён, но есть номера, начинающиеся с 1). Перед первым символом строки имеется ещё и нулевой, в котором хранится символ с

кодом, равным длине строки.

**Сравнение строк**. Строки сравниваются последовательно, по символам. Сравниваются первые символы строк, если они равны - то вторые, и т. д. Если на каком-то этапе появилось различие в символах, то меньшей будет та строка, в которой меньший символ. Если строки не различались, а затем одна из них закончилась, то она и считается меньшей. Примеры: 'ананас'<'кокос', 'свинья'>'свинина', "<'A', 'hell'<'hello'.

**Склеивание (конкатенация) строк**. К строкам можно применять операцию "+", при этом результатом будет строка, состоящая из последовательно записанных "слагаемых". Пример: по-сле действия s:= 'abc'+'def'+'ghi'; переменная s будет содержать 'abcdefghi'.

**Процедуры и функции для работы со строками**.

`length(s: string): integer` (после двоеточия записан тип значения, возвращаемого функцией, в нашем случае - целое число). Эта функция возвращает длину строки `s`.

`copy(s: string; start: integer; len: integer): string` Возвращает вырезку из строковой переменной `s`, начиная с символа с номером start, длина которой `len`

`pos(s1: string; s: string): byte` Ищет подстроку `s1` в строке `s`. Если находит, то возвращает номер символа, с которого начинается первое вхождение `s1` в `s`; если `s1` не входит в s, то функция возвращает 0

`insert(s1: string; s: string; start: integer)` Вставляет строку `s1` в строковую переменную `s` начиная с символа с номером `start.`

`delete(s: string; start: integer; len: integer)` Удаляет из строковой переменной `s` фрагмент, начинающийся с символа с номером `start` и длиной `len.`

`str(x,st)` : преобразует число, записаное в `x` (целого типа) в строковой тип и записывает его в `st;` Например, если в `x` будет записано число `132,` то в `st` мы получим строчку `'132'.`

`val(st,x,er)` : обратная процедура к `str,` то-есть, переводит строковое значение `(st)` в числовое `(x).` В переменную `er (Integer)` записывается код ошибки. Код ошибки необходим, поскольку в строке могут содержатся не только цифры, но и буквы, которые не преобразуются в числа. Поэтому, если в `st` были не только числа, то в `x` ничего не запишется, а в `er` будет не ноль, а какое-то число. Обычно, после этой процедуры проверяют, ноль ли записан в `er` чтобы убедиться в правильности операции.

## Записи

Тип запись, также как и массив, является структурированным типом данных, то есть таким, переменные которого составлены из нескольких частей. В Pascal существует возмож-ность объединить в одну переменную данные разных типов (тогда как в массиве все элементы имеют одинаковый тип). Приведём пример такого типа. Пусть в переменной требуется хранить сведения о некотором человеке: ФИО, пол, адрес, телефон. Тогда для хранения этих данных бу-дет удобен такой тип:

```
type tPerson = record
   Name,Surname,SecondName: string[30];
   Male: boolean;
   Address: string[50];
   Phone: string[11];
end;
```

Объявление переменной типа запись выполняется стандартно, с помощью var. Части записи (в нашем случае: Name, Surname, SecondName, Male, Address, Phone) называются полями. Обращение к полю записи в программе производится с помощью знака '.' (точка). Пример обращения к полям:

```
var emp: tPerson;
...
begin
...
emp.Surname:='Иванов';
emp.Name:='Иван';
emp.SecondName:='Иванович';
...
```

В случаях, когда приходится много раз обращаться к полям одной и той же записи, можно воспользоваться ключевым оператором with, который упрощает ссылку к структурированным переменным:

```
with <имя_записи> do <оператор>;
```

Пример:

```
with emp do begin
Surname:=' Иванов';
Name:='Иван';
SecondName:='Иванович';
...
end;
```

Записи можно включать в состав более сложных переменных, например массивов и других записей:

```
type tStaff = array [1..30] of tPerson;
```

## Работа с файлами

В паскале работа с файлами осуществляется через специальные файловые типы, которые определяют тип файла, то есть фактически указывают его содержимое. С помощью этой переменной, которой присвоен необходимый тип, и осуществляется вся работа с файлами - открытие, запись, чтение, закрытие и т.п.

При работе с файлами существует определенный порядок действий, которого необходимо придерживаться. Вот все эти действия:

1. Создание (описание) файловой переменной;
2. Связывание этой переменной с конкретным файлом на диске или с устройством ввода-вывода (экран, клавиатура, принтер и т.п.);
3. Открытие файла для записи либо чтения;
4. Действия с файлом: чтение либо запись;
5. Закрытие файла.

### Типы файловых переменных

Перед тем, как начинать работу с файлами, давайте посмотрим, какие существуют переменные для работы с ними. В Pascal имеется три типа таких переменных, которые определяют тип файла:

1. **text** - текстовый файл. Из переменной такого типа мы сможем читать строки и символы.
2. **file of _любой_тип_** - так называемые "типизированные" файлы, то есть файлы, имеющие тип. Этот тип определяет, какого рода информация содержится в файле и задается в параметре_любой_тип_. Например:

```
F: file of integer;
```

Файл F содержит числа типа integer; Соответсвенно, читать из такого файла можно только переменные типа integer, ровно как и писать.

Пример

```
type
A = record
I,J: Integer;
S: String[20];
end;
var
F: File of A;
```

3. **file** - нетипизированный файл. Когда мы указываем в качестве типа файла просто File, то есть без типа:

```
F: File;
```

То получаем "нетипизированный" файл, чтение и запись в который отличается от работы с файлами других типов. Эти действия производятся путем указания количества байт, которые нужно прочитать, а также указанием области памяти, в которую нужно прочитать эти данные.

## Связывание переменной с файлом

Выполняется одинаково для всех типов файлов:

```
assign(<переменная_файлового_типа>, '<путь к файлу>');
```

В качестве параметров задаются переменная любого файлового типа и строка - путь к файлу:

```
var
T: text;
Fi: file of integer;
F: file;
begin
Assign(T, 'text.txt');
Assign(F1, 'int.txt');
Assign(F2, 'file1.dat');
```

## Открытие файла

При открытии файла необходимо учитывать, зачем открывается файл - для записи или чтения. Более того, в зависимости от типа файла процедуры выполняют различные действия.

1. **Reset**(<любая_файловая_переменная>);

   Открыват файл на чтение. В качестве параметра - файловая переменная любого из перечисленных выше типов. Это может быть текстовый, типизированный либо не типизированный файл. В случае с текстовым файлом, он открывается только на

чтение. В случае с типизированным и нетипизированным файлом - он открывается на чтение и запись.

2. **Append**(T: Text);

   Эта процедура открывает текстовый файл (только текстовый!) на запись. Reset при задании параметра типа Text не позволит писать в него данные, открыв файл лишь для чтения. То есть если вы используете текстовый файл и хотите производить в него запись, нужно использовать Append. Если чтение - Reset.

   Также обратите внимание, что если вы до этого уже открыли файл на чтение, вам не нужно закрывать его и открывать снова на запись. В этом случае файл закрывается сам и открывается заново. При записи данных в файл при открытии его с помощью этой процедуры они записываются в конец файла.

3. **ReWrite**(F) - создает новый файл либо перезаписывает существующий. Необходимо быть осторожным при использовании этой процедуры, т.к. файл, открытый таким образом будет полностью перезаписан.

## Закрытие файла

Закрытие файла производиться с помощью процедуры **Close(F)**, где F - это переменная файлового типа. Эта процедура одна для всех типов файлов.

## Запись и чтение файлов

**Текстовые и типизированные файлы**

**Чтение файлов**. Чтение файлов производится с помощью отлично известных нам процедур `read` и `readln`. Они используются также, как и при чтении информации с клавитуры. Отличие лишь в том, что перед переменной, в которую помещается считанное значение, указывается переменная файлового типа (дескриптор файла):

```
read(F, C);
```

где `F` - дескриптор файла, `C` - переменная (char, string - для текстовых, любого типа - для типизированных файлов).

**Запись в файлы**. Запись в файлы производиться точно так же, как и запись на экран - с помощью процедур `write` и `writeln`. Как и в случае с чтением, перед записываемой в файл переменной указывается тескриптор файла:

```
write(F, S);
```

где `F` - дескриптор, `S` - переменная.

При этом, естественно, переменная должна соответствовать типу файла.

```
program cat;
var
   f: text;
   c: char;
begin
   assign(f,'prog.pas');
   reset(f);
   while not eof(f) do
   begin
        while not eoln(f) do
        begin
```

```
            read(f,c);
            write(c);
        end;
        readln(f);
        writeln;
    end;
    close(f);
end.
```

Довольно часто в программе бывает необходимо определить, дошёл ли указатель файла до конца строки или до конца файла. В этом случае полезно использовать такие функции:

```
eoln(TxtFile: text): boolean;
eof(TxtFile: text): boolean;
```

Первая принимает значение true (истина), если указатель стоит на конце строки, вторая - то же самое для конца файла.

**Нетипизированные файлы**

Суть таких файлов заключается в следующем: имея файл без определенного типа, мы можем читать из него любые данные, будь то строки, символы или записи.

Читая данные из файла без типа мы получаем блоки информации, которые составляют обычный набор байт. Указывая переменную, в которую эти байты надо поместить, мы как бы "на ходу преобразуем" эти данные к нужному типу.

**Чтение из файлов без типа**

Сама процедура связывания файловой переменной с внешним файлом и его открытие ничем чем отличаются от обычного порядка действий. Разве что переменная в данном случае должна иметь тип File; , то есть быть файлом без типа.

```
var
F: File;
begin
{ связываем файл с переменной }
Assign(F, '1.txt');
Reset(F);
end.
```

Чтение происводиться с помощью процедуры `blockread`:

```
blockread(F: file, Buf: var, size: word, result: word)
```

`F: file;` - переменная типа `file`; Именно из этой переменной и происходит чтение данных.

`Buf: var;` - переменная любого типа. В эту переменную помещаются прочитанные данные.

`Size: word;` - количество считываемых байт.

`Result: word;` - в эту переменную помещается реальное количество байт, которые были прочитаны.

Работает эта процедура следующим образом: из файла `F` считывается `Size` записей, которые помещаются в память, начиная с первого байта переменной `Buf`. После выполнения процедуры реальное количество прочитанных байт помещается в переменную `Result`.

Здесь надо сказать, что эта переменная совсем не обязательно должна присутствовать в качестве параметра, то есть ее попросту можно опустить. Однако иногда она довольно полезна и даже необходима - например, если чтение было окончено до того, как было прочитано требуемое количество байт (достигнут конец файла), мы можем это отследить через переменную Result. Если же в этом случае (чтение данных после конца файла) переменная Result не будет указана, то образуется ошибка времени выполнения N100 "Disk read error" (Runtime error 100).

**Запись в файлы без типа**

Ну что ж, с чтением данных вроде разобрались, пора переходить к записи. Для этого в Паскале имеется еще одна, отдельная процедура, а именно BlockWrite. Она очень похожа на предыдущуюBlockRead, по крайней мере параметры у этих двух процедур абсолютно одинаковы.

```
BlockWrite(F: File, Buf: Var, Size: Word, Result: Word)
```

F: File; - переменная типа File;

Buf: Var; - переменная любого типа. Начиная с этой переменной, данные будут записываться в файл.

Size: Word; - количество записываемого блока данных в байтах.

Result: Word; - в эту переменную помещается реальное количество байт, которые были записаны.

```
var
   Fin, fout : file;
   NumRead,NumWritten : word;
   Buf : Array[1..2048] of byte;
   Total : longint;

begin
   Assign (Fin, Paramstr(1));
   Assign (Fout,Paramstr(2));
   Reset (Fin,1);
   Rewrite (Fout,1);
   Total:=0;
   Repeat
        BlockRead (Fin,buf,Sizeof(buf),NumRead);
        BlockWrite (Fout,Buf,NumRead,NumWritten);
        inc(Total,NumWritten);
   Until (NumRead=0) or (NumWritten<>NumRead);
   Write ('Copied ',Total,' bytes from file ',paramstr(1));
   Writeln (' to file ',paramstr(2));
   close(fin);
   close(fout);
end.
```