12. Настройки на командния интерпретатор и писане на прости скриптове  
• Конфигурационни файлове на средата;  
• Управление на променливите ;  
• Писане на скриптове – параметри, вход и изход, аритметични и логически операции,  
сравняване, условни оператори, цикли, функции.

Описание на командния интерпретатор. Видове командни интерпретатори.

Тук ще разгледаме няколко основни терминологични термина, които стоят в основата на всяка Linux и UNIX™ система. Първото от тях е понятието *конзола*. Това е исторически термин зад който стои дългогодишната история на компютъра въобще. В началото под конзола се е разбирало терминала на който е стоял администратора на машината и на който са излизали системните съобщения. Най-общо казано конзолата е текстово изходно устройство, което прихваща системните съобщения идващи от ядрото на операционната система или от програмата за системни съобщения. В големите компютри, конзолата най-често е била свързана със серийна връзка, работеща по стандарта RS-232 (серийния порт на компютъра). В съвременните персонални компютри под конзола се разбира монитора на компютъра. В Линукс съществува и понятието виртуална конзола, като това са шесте терминални прозореца, които може да се превключват с Alt+F1 до F6. Важно е да се отбележи, че системните съобщения излизат на първа виртуална конзола даже и никой да е логнат в нея. Типичен пример за това са системните съобщения на ядрото, които се появяват при стартирането на системата.

Друго важно понятие е *терминал*. Терминал с устройство (най-често хардуерно), което позволява комуникацията с компютъра. По принцип терминала е комбинация от монитор и клавиатура. Това не е така, обаче когато се говори за *отдалечен терминал*. Отдалечения терминал представлява специална програма съставена от клиент и сървър, чрез която потребителя може да се свързва отдалечено към компютър в който има валидно потребителско име и парола. Като за начало трябва да знаете, че най-често използваните програми са *telnet* и *ssh*.

Най-общо терминала е това което стои между потребителя и *шела* (командния интерпретатор). Шел се нарича програмата с помощта, на която системата комуникира с потребителя. Тя чете въведените от терминала редове и изпълнява различни операции в зависимост от това което е въведено. След това, шела се опитва да преобразува въвежданите конструкции в инструкции, които ядрото е в състояние да разбере.

Всеки потребител при логическото си включване в системата стартира свое копие на шел в паметта. Това се прави, за да може той да работи без да пречи на останалите потребители на системата.

Шела възприема всичко до първия интервал като команда, а всичко останало, като аргументи на тази команда, като аргументите също се разделят с интервали.

Всички (или поне повечето) Линукс системи имат повече от един валиден шел. Те са описани във файла */etc/shells*. Той има следния прост вид:

*/bin/bash*

*/bin/tcsh*

*/bin/csh*

*/bin/ash*

*/bin/ksh*

*/bin/zsh*

*/bin/rbash*

Линукс система може да има повече инсталирани командни интерпретатори, но те са недостъпни за използване ако не са описани в този файл. Всеки потребител може да промени своя шел чрез командата *chsh*. Без параметри командата влиза в интерактивен режим на работа, а чрез опцията *-s*, този режим се изключва. Формата на командата е следния:

*#chsh -s login\_shell user*

*или само*

*#chsh*

В някои дистрибуции е възможно проверка на валидните шелове чрез ключа *-l*към командата *chsh*, но *Slackware* не поддържа този ключ.

Въпреки, че съществуват множество командни интерпретатори, то не всички се използват масово. Най-често използваните командни интерпретатори са:

* *bash* – bash е команден интерпретатор съвместим с шела *sh*. Той може да изпълнява команди, както от ред, така и от файл. Bash притежава мощен скриптов език, чрез който може да се пишат програми, изпълнявани директно от командния интерпретатор. Освен това bash притежава и функции, които са присъщи на други командни интерпретатори като *ksh* и *csh*.
* *csh* – това е шел, който обединява типичните функции за един команден интерпретатор като автоматично завършване на файлов имена при натискането на *Tab*, управление на задачите и поддръжка на история на командите със синтаксиса на *C*.

* *tcsh* – подобрена версия на *csh*, включваща допълнително корекция на синтаксиса и др., като запазва пълна съвместимост с оригиналния *csh*. Повечето дистрибуции създават символична връзка *csh*, сочеща към *tsch*.

* *ksh* – KornShell e команден интерпретатор, поддържащ собствен език за програмиране и позволява изпълнение на команда както от команден ред, така и от файл.

* *zsh* – подобрена версия на *ksh*. Притежва множество предимства пред него, като функции, вградена синтактична проверка. За разлика от *tcsh* и *csh*, то *zsh* не е напълно съвместим с *ksh*.

Повечето от изброените командни интерпретатори (*bash, ksh, zsh*) създават символична връзка, която започва с ***r***и завършва с името на шела (*rbash, rksh, rzsh*), която указва на командния интерпретатор да превключи в защитен режим. В този режим потребителя е силно ограничен, като ограниченията зависят от конкретния шел. Описанието на *restricted* шеловете в /*etc/shells* не се препоръчва, защото след преминаване към такъв шел, потребителя не може да се върне към обичайния за него команден интерпретатор.

Най-използвания от гореизброените командни интерпретатори е bash. Той е и шела по-подразбиране в Linux. Bash е абревиатура от Bourne Again Shell, създаден от Free Software Foundation на базата на Bourne shell от UNIX™.

Всяка обвивка (обкръжение) има свои променливи на обкръжението. Техните стойности се установяват от командния интерпретатор при стартирането му и имат различни стойности за различните потребители. Ето ви още една причина за всеки потребител да се стартира отделно копие на интерпретатора. Стойностите на тези променливи могат да се променят в процеса на работа.

Например ако не ви харесва знака $ за промпт достатъчно е да смените променливата на обкръжението PS1.

Променливите на обкръжението може да се променят чрез командата:

*$export PS1=%*

*%*

само за текущата сесия или чрез промяна на файла *.bash\_profile* или само *.profile*, като в този случай промяната става активна след като потребителя се логне и е постоянна. *Slackware* използва файла *.bash\_profile* за въвеждане на команди или настройка на *bash*. Този файл се зарежда винаги когато потребителя влезе в системата. По-важните опции на bash са:

* *-r, --restricted* – указва на шела да влезе в ограничен режим. В този режим на потребителя са забранени следните операции: промяната на директории с командата *cd*; промяна на променливите на обкръжението SHELL, PATH, ENV и BASH\_ENV; изпълнението на команди съдържащи знака /; задаването на / като аргумент на вградената команда .; задаване на файл съдържащ / като аргумент на вградената команда hash; импортиране на дефиниции на функции при стартиране; пренасочване на изхода с*>, >|, <>, >&* и *>>*; изпълнението на командата exec; изключване на режима на ограничения чрез командите *set +r* или *set +o restricted*. Освен по този начин режима на ограничения може да се включи и чрез изпълнението на *rbash.*

* *-rcfile* file – изпълнява командите от файла, вместо от стандартния .*bashrc*.

Всички командни интерпретатори има множество променливи на обкръжението. Тези променливи описват различни характеристики на системата. Те се четат от останалите програми, които ги използват за настройка на вътрешните си параметри, като език на интерфейса, локализация, тип на системата, версия на операционната система, текуща директория и др. Bash притежава множество променливи на обкръжението, по-важните от които са:

* BASH – указва пълното име на файла използвано за извикването на това копие на bash.
* BASH\_VESRION – показва версията на командния интерпретатор.
* GROUPS – показва списък с групите на които е член потребителя.
* HOSTNAME – показва името на компютъра.
* OSTYPE – показва на каква операционна система се изпълнява копието на bash
* PWD – показва текущата директория
* RANDOM – при всяко извикване този параметър се променя от 1 до 32767 по случаен признак.

* UID – показва уникалния идентификационен номер на потребителя, който е стартирал копието на bash.
* HISTFILE – дефинира и показва файла в който се съхраняват изпълнените команди. По подразбиране това е ~/.bash\_history.

* HISTSIZE – дефинира и показва максималното количество реда, които може да се съдържат във файла с историята на командите.
* HOME – дефинира и показва домашната директория на потребителя.
* LANG – дефинира локала на операционната система за всички категории. Отделните категории може да са различни от този параметър.
* LC\_MESSAGES – дефинира езика на който да се показват различните съобщения.
* PATH – показва и дефинира пътя в който се търсят командите.

Четенето на тези променливи може да стане с командата *echo*:

*$echo $HOME*

*/home/dino*

*$*

Записа на нова стойност става чрез командата *export*:

*$export HOME=/home/ftp*

*$echo $HOME*

*/home/ftp*

*$*

Важна особеност е това, че при четене от променливата е необходимо пред нея да се сложи знак за долар, докато при нейна промяна това не е необходимо.

Освен променливите на обкръжението, които се дефинират автоматично от командния интерпретатор, почти всички програми дефинира собствени променливи на обкръжението.

Bash притежава мощен език за програмиране, който позволява писането на скриптове за администриране на системата или за други дейности, които могат да се извършават автоматизирано. Bash скрипта може да представлява или поредица от команди поставени във файл, които се изпълняват последователно, или скрипт написан на езика на bash в който се съчетават външни команди, променливи, цикли, условия и вътрешни bash команди. Пример за прост bash скрипт е следната последователност от команди:

***$cat > simple\_script***

*echo "Building file...."*

*ls -l /dev > dev\_list*

*echo "Number of files:"*

*wc -l dev\_list*

*rm dev\_list*

*echo "Exiting"*

*Ctrl+D*

***$chmod +x simple\_script***

***$./simple\_scipt***

*Building file....*

*Number of files:*

*2633 dev\_list*

*Exiting*

***$***

Тази проста поредица от команди са обединени в един текстов файл наречен шел скрипт. Този файл трябва да има права за изпълнение, за да може командния интерпретатор да го изпълни. Повечето шел скриптове обаче не са толкова прости. Примери за сложни скриптове са командите *installpkg*, *pkgtool* и т.н, които служат за добавяне, премахване или обновяване на програмите под Slackware. Чрез използването на командите *kdialog* и *dialog* на шел скриптовете може да се направи и графичен интерфейс съответно за KDE и за текстова конзола. Пример за такъв скрипт е *pkgtool*и*netconfig*.

Писането на шел скриптове е една от трудните аспекти на Линукс, но тяхното разбиране би довело до пълен контрол върху системата чрез прост набор от предварително написани скриптове.

# Какви са скритите файлове в моя Linux Home Directory?

Във вашата Linux система вероятно сте запазили много файлове и папки във вашата домашна директория. Но под тези файлове, знаете ли, че вашата домашна директория идва с много скрити файлове и папки? Ако стартирате ls -a в домашната си директория, ще откриете куп скрити файлове и директории с точкови префикси. Какво представляват тези скрити файлове?

## За какво са скрити файлове в домашната директория?

Най-често скритите файлове и директории в домашната директория съдържат настройки или данни, достъпни от програмите на този потребител. Те не са предназначени за редактиране от потребителя, а само от приложението. Ето защо те са скрити от нормалния изглед на потребителя.

Като цяло файловете от собствената ви домашна директория могат да бъдат премахнати и променени без да се повреди операционната система. Приложенията, които разчитат на тези скрити файлове обаче, може и да не са толкова гъвкави. Когато премахвате скрит файл от домашната директория, обикновено губите настройките за приложението, свързано с него.

Програмата, която разчиташе на този скрит файл, обикновено го пресъздава. Въпреки това, ще започнете от настройките "излезли от кутията", като чисто нов потребител. Ако имате проблеми с приложението, това всъщност може да бъде огромна помощ. Той ви позволява да премахвате персонализации, които могат да причиняват проблеми. Но ако не сте, това просто означава, че ще трябва да върнете всичко по начина, по който ви харесва.

## Какви са някои специфични приложения на скритите файлове в домашната директория?

Всеки ще има различни скрити файлове в своята домашна директория. Има някои, които всеки има. Файловете обаче имат подобна цел, независимо от приложението на родителя.

### Системни настройки

Системните настройки включват конфигурацията за работната ви среда и вашето тяло.

* **Файлове** за **конфигуриране** на помощните инструменти за кошче и командния ред: В зависимост от конкретните помощни програми, които използвате, специфичното име на файла ще се промени. Ще видите файлове като ".bashrc", ".vimrc" и ".zshrc". Тези файлове съдържат всички настройки, които сте променили за операционната среда на вашия корпус или ощипванията, които сте направили в настройките на помощните програми на командния ред vim . Премахването на тези файлове ще върне съответното приложение в състояние по подразбиране. Като се има предвид, че много потребители на Linux създават масив от фини настройки и настройки през годините, премахването на този файл може да е огромно главоболие.
* **Потребителски профили:** Както горепосочените конфигурационни файлове, тези файлове (обикновено ".profile" или ".bash\_profile") запазват потребителските настройки за корпуса. Този файл често съдържа вашата PATH. Той съдържа и псевдоними, които сте задали. Потребителите могат също да поставят псевдоними в .bashrc или други места. PATH урежда къде обвивката търси изпълними команди. Чрез добавяне или модифициране на PATH можете да промените мястото, където черупката ви търси команди. Псевдонимите променят имената на командите. Един псевдоним може да настрои на ll да извика ls -l, например. Това осигурява текстови преки пътища към често използваните команди. Ако изтриете .profile, често можете да намерите стандартната версия в директорията "/ etc / skel".
* **Настройки на работната среда:** Това спестява персонализиране на работната ви среда. Това включва фон на работния плот, скрийнсейвъри, клавишни комбинации, ленти с менюта и икони на лентата на задачите и всичко останало, което потребителят е задал за работната среда на работния плот. Когато премахнете този файл, средата на потребителя се връща към новата потребителска среда при следващото влизане.

### Конфигурационни файлове на приложенията

Ще ги намерите в папката ".config" в Ubuntu. Това са настройки за вашите конкретни приложения. Те ще включват неща като списъците с предпочитания и настройките.

* **Файлове за конфигурация за приложения** : Това включва настройки от менюто с предпочитания за приложения, конфигурации на работното пространство и др. Точно това, което ще намерите тук, зависи от приложението на родителя.
* **Данни на уеб браузър:** Това може да включва неща като отметки и история на сърфиране. По-голямата част от файловете съставляват кеша. Това е мястото, където уеб браузърът съхранява временно файлове, като изображения. Премахването на това може да забави някои уебсайтове с тежки медии при първото посещение.
* **Кешира** : Ако приложение на потребител съхранява данни, които са подходящи само за този потребител (например Spotify, съхраняващ кеша на плейлиста ви), домашната директория е естествено място за съхранение. Тези кеширания може да съдържат маси от данни или само няколко реда код: зависи от това, от което се нуждае родителското приложение. Ако премахнете тези файлове, приложението ги пресъздава, ако е необходимо.
* **Записи:** Някои потребителски приложения могат да съхраняват и дневници тук. В зависимост от начина, по който разработчиците настройват приложението, може да намерите регистрационни файлове, съхранени в домашната ви директория. Това обаче не е обикновен избор.

## заключение

В повечето случаи скритите файлове в началната ви директория в Линукс, използвани за съхраняване на потребителските настройки. Това включва настройки за помощни програми на командния ред, както и приложения, базирани на GUI. Премахването им ще премахне потребителските настройки. Обикновено това няма да доведе до прекъсване на програмата.

# Променлива на средата

**Променливите на средата** (environment variables) представляват **динамични стойности, които засягат работещите процеси в една операционна система**.

## **Същност**

Съществува един особен вид променливи, наричани "променливи на обкръжението". Най-общо казано, това са променливи, поддържани от [сървъра](http://basaga.org/wiki/index.php?title=%D0%A1%D1%8A%D1%80%D0%B2%D1%8A%D1%80&action=edit&redlink=1) и PHP [интерпретатора](http://basaga.org/wiki/index.php?title=%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80&action=edit&redlink=1), описващи текущото състояние на връзката и характеристиките на двете страни в нея - клиент и сървър. Двете промливи - $HTTP\_USER\_AGENT и $HTTP\_REFERER, помагат да се открие с какъв браузър посетителят разглежда страницата и от къде е дошъл на нея. Събирането и анализирането на подобни данни е безкрайно полезно за всеки web разработчик. Така можете да научите кой [сайт](http://basaga.org/wiki/index.php?title=%D0%A1%D0%B0%D0%B9%D1%82&action=edit&redlink=1) ви праща най-много посетители и да организирате рекламната си стратегия, наблягайки на него. Полезно е също и да следите кои от страниците в сайта ви се посещават повече и предизвикват по-голям интерес, така че да наблегнете на тях и на подобен вид [съдържание](http://basaga.org/wiki/index.php?title=%D0%A1%D1%8A%D0%B4%D1%8A%D1%80%D0%B6%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) в по-нататъчното изграждане на сайта.

## **Пример**

Пълен преглед на променливите на средадата може да видите, използвайки вградената в PHP [функция](http://basaga.org/wiki/index.php?title=%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) - phpinfo(). Напишете в текстов файл следните редове :

<?

phpinfo();

?>

Запишете го на сървъра с име info.php, примерно, и го отворете в браузъра си. Ще видите най-важната информация за PHP инсталацията, както и всички създадени в текущата връзка променливи на [средата](http://basaga.org/wiki/index.php?title=%D0%92%D1%8A%D0%BD%D1%88%D0%BD%D0%B0_%D1%81%D1%80%D0%B5%D0%B4%D0%B0) на PHP и сървъра. Изписаното в левите [полета](http://basaga.org/wiki/index.php?title=%D0%9F%D0%BE%D0%BB%D0%B5&action=edit&redlink=1) на [таблицата](http://basaga.org/wiki/index.php?title=%D0%A2%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0&action=edit&redlink=1) представляват имената на променливите, а в дясната част - [стойностите](http://basaga.org/wiki/index.php?title=%D0%A1%D1%82%D0%BE%D0%B9%D0%BD%D0%BE%D1%81%D1%82&action=edit&redlink=1) им. Нека сега разширим [примера](http://basaga.org/wiki/index.php?title=%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%80&action=edit&redlink=1) и добавим още малко [информация](http://basaga.org/wiki/index.php?title=%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) в него.

[](http://basaga.org/wiki/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Psr.png)

Нека сега обясним отделните редове в него. Нямаше нужда да използвам променливите $browser, $ref, $user\_ip, $forw, $serv и $rem\_host, но ги включих за по-голяма прегледност. Всяка от тях получи за стойност стойността на някоя от променливите на средата.

На променливата с име "browser" бе присвоена стойността на $HTTP\_USER\_AGENT, която винаги съдържа идентификационната информация на браузъра. Браузърът Opera позволява на потребителите си да променят [идентификацията](http://basaga.org/wiki/index.php?title=%D0%98%D0%B4%D0%B5%D0%BD%D1%82%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) й, така че тя може да се представя като IE или Netscape. Причината е в рестрикциите, които някои сайтове налагат на използващите [браузър](http://basaga.org/wiki/index.php?title=%D0%91%D1%80%D0%B0%D1%83%D0%B7%D1%8A%D1%80&action=edit&redlink=1), различен от Internet Expolorer. Обикновено това също става с помощтта на $HTTP\_USER\_AGENT. Имайки предвид особеностите на различните браузъри, можете да покажете на посетителя на сайта си страница, оптимизирана за неговата програма, съдейки по съдържанието на тази променлива на средата. В скоби казано, възможността на Opera да се представя по различен начин може да ви изправи пред куриозната информация, че посетителят използва IE под [операционна система](http://basaga.org/wiki/index.php?title=%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0&action=edit&redlink=1) Linux.

На променливата с име "ref" бе присвоена стойността на $HTTP\_REFERER, съдържаща URL на мястото, от което идва посетителят. Вече обяснихме по-подрбоно тази променлива, но трябва да имате предвид, че тя не винаги дава полезна информация. В конкретния случай едва ли мога да очаквам, че Yahoo са поставили на началната си страница [линк](http://basaga.org/wiki/index.php?title=%D0%9B%D0%B8%D0%BD%D0%BA&action=edit&redlink=1) към моя тестов файл.

Променливата, наименувана "user\_ip" получи стойността на променливата на средата $REMOTE\_ADDR. Тя винаги съдържа в себе си IP адреса на посетителя. Тук много рядко може да се получи разминаване на стойността с истинската. Защото сървърът използва стойността й за да определи на кого да изпрати поискания файл.

Много често се случва множество потребители да се намират зад едно или повече прокси сървъри. Няма да коментираме сега защо и как се получва това. В случая трябва да имаме предвид, че ако посетителят е зад прокси, то $REMOTE\_ADDR ще покаже IP адреса на проксито. За адреса на клиентската [машина](http://basaga.org/wiki/index.php?title=%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%B0) се създава нова променлива - $HTTP\_X\_FORWARDED\_FOR. Повечето прокси сървъри осигуряват тази информация, особено ако [администраторите](http://basaga.org/wiki/index.php?title=%D0%90%D0%B4%D0%BC%D0%B8%D0%BD%D0%B8%D1%81%D1%82%D1%80%D0%B0%D1%82%D0%BE%D1%80&action=edit&redlink=1) им не желаят да изкарат живота си по съдебните зали. Използването на последните две описани променливи на средата е полезно, когато давате достъп до чувствителна информация или по една или друга причина е важно да знаете кой точно е отсреща. Така можете да ограничите достъпа до сайта си на хора, които нарушават добрия тон в него или да подсигурите сигурността му при e-commerce приложения.

Тук използвахме и още една променлива на [средата](http://basaga.org/wiki/index.php?title=%D0%92%D1%8A%D0%BD%D1%88%D0%BD%D0%B0_%D1%81%D1%80%D0%B5%D0%B4%D0%B0) - $SERVER\_SOFTWARE, която съдържа в себе си информация за сървърния софтуер - версия на сървъра и операционна система. В горния пример използвахме това, като "подпис" на [съобщението](http://basaga.org/wiki/index.php?title=%D0%A1%D1%8A%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1).

Писане на скриптове – параметри, вход и изход, аритметични и логически операции,  
сравняване, условни оператори, цикли, функции.

Shell скриптове

# 1. Въведение

## **Цел на този урок**

Този урок е написан, за да помогне на хората да разберат някои от основите на програмирането на **скриптове** (известен още като **скриптове за скриптове** ) и да се надяваме да въведем някои от възможностите за просто, но мощно програмиране, достъпно под обвивката на Борн. Като такава, тя е написана като основа за индивидуални или групови уроци и упражнения и като референция за последваща употреба.

## **Първи най-новата версия на този урок**

Вие четете версия 2.1b , последно актуализирана на 11 февруари 2018 година .   
Най-новата версия на този урок е винаги достъпна на [адрес](https://www.shellscript.sh/) : [https://www.shellscript.sh](https://www.shellscript.sh/) . Винаги проверявайте там за последното копие. (Ако четете това на друг адрес, вероятно е копие на реалния сайт и следователно може да е остарял).

## **Кратка история на sh**

Стив Борн пише черупката на Борн, която се появява в Седмото издание на Bell Labs Research версия на Unix.   
Бяха написани много други черупки; този урок се концентрира върху снарядите Bourne и Bourne Again.   
Другите черупки включват Korn Shell (ksh), C Shell (csh) и вариации като tcsh.   
Този урок *не* покрива тези черупки.

## **Публика**

Този урок предполага известен предишен опит; а именно:

* **Използване на** един **интерактивен** Unix / Linux черупка
* **Минималното програмно знание** - използването на променливи, функции, е полезно фоново знание
* Разбиране на **някои** Unix / Linux команди и **компетентност** при използването на **някои** от най **-често срещаните** . ( *ls* , *cp* , *echo* и т.н.)
* Програмистите на **ruby** , **perl** , **python** , **C** , **Pascal** или всеки език за програмиране (дори BASIC), които могат да четат скриптове, но не чувстват, че разбират точно как работят.

Може да искате да прегледате някои от [отзивите, които този урок е получил, за](https://www.shellscript.sh/feedback.html) да видите колко полезни можете да откриете.

## **Типографски конвенции, използвани в този урок**

Значителни думи ще бъдат написани в *курсив,* когато се споменава за първи път.

Кодовите сегменти и изходът на скрипта ще се показват като моноспектен текст.   
Записите от командния ред ще бъдат предшествани от знака за долар ($). Ако подканата ви е различна, въведете командата:

PS1 = "$" ; изнася PS1

Тогава вашите взаимодействия трябва да съвпадат с дадените примери (като по- **./my-script.sh**долу).   
Изходът на скрипта (като "Hello World" по-долу) се показва в началото на реда.

$ echo '#!/bin/sh' > my-script.sh

$ echo 'echo Hello World' >> my-script.sh

$ chmod 755 my-script.sh

$ ./my-script.sh

Hello World

$

Цели скриптове ще бъдат показани със сив фон и ще включват препратка към обикновения текст на скрипта, когато е наличен:

[my-script.sh](https://www.shellscript.sh/eg/my-script.sh.txt)

#!/bin/sh

# This is a comment!

echo Hello World # This is a comment, too!

Обърнете внимание, че за да направите изпълним файл, трябва да зададете изпълним бит, а за скрипт на черупката трябва да се зададе също и Readable bit:

$ chmod a+rx my-script.sh

$ ./my-script.sh

# 2. Философия

Програмата за скриптове на Shell има малко лоша преса сред някои системни администратори на Unix. Това обикновено се дължи на едно от следните две неща:

* Скоростта, с която ще се изпълнява интерпретираната програма в сравнение с програма C или дори интерпретирана Perl програма.
* Тъй като е лесно да се напише обикновен скрипт от типа на пакетна работа, наоколо има много лоши качествени скриптове.

Това отчасти се дължи на това, че има известен мачизъм, свързан със създаването на *добри* скриптове. Скриптове, които могат да се използват като CGI програми, например, без да губят прекалено много скорост към Perl (макар че и двете ще загубят до C, в много случаи скоростта е единственият критерий).   
Има редица фактори, които могат да влязат в добри, чисти и бързи скриптове.

* Най-важните критерии трябва да бъдат ясен и разбираем план.
* Второ е избягването на ненужни команди.

Ясното оформление прави разликата между скрипт, който се появява като "черна магия" и който лесно се поддържа и разбира.   
Може да ви се прости, ако мислите, че с прост скрипт това не е твърде важен проблем, но две неща тук си струват да се имат предвид.

1. Първо, обикновеният скрипт, по-често от очакваното, ще се превърне в голям, сложен.
2. На второ място, ако никой друг не може да разбере как работи, вие ще бъдете подготвени да го поддържате сами до края на живота си!

Нещо за скриптовете на черупките изглежда, че ги правят особено вероятно да бъдат зле направени, и тъй като основните контролни структури са if / then / else и loops, отстъпът е от решаващо значение за разбирането на това какво прави скрипта.

Една от слабостите в много скриптове на обвивката са линии като:

cat /tmp/myfile | grep "mystring"

което щеше да работи много по-бързо, като:

grep "mystring" /tmp/myfile

Не много, може да помислите; операционната система трябва да зареди **/bin/grep** изпълнимия файл, който е сравнително малък 75600 байта в моята система, да отвори **pipe**в паметта за прехвърляне, да зареди и да изпълни **/bin/cat**изпълнимия файл, който е още по-малък 9528 байта в моята система, прикрепете го към входа на тръбата и я оставете да работи.

Разбира се, това е нещо, за което операционната система е там и обикновено е много ефективна. Но ако тази команда е била в цикъл, който се изпълнява многократно, спестяването на не локализиране и зареждане на **cat** изпълнимия файл, настройка и освобождаване на тръбата, може да направи някаква разлика, особено в CGI среда, където има достатъчно други фактори. да забавя нещата, без самият сценарий да е прекалено пречка. Някои единици са по-ефективни от другите в това, което те наричат ​​"изграждане и разрушаване на процеси" - т.е. зареждането им, изпълнението им и изчистването им отново. Но колкото и добър да е вашият вкус на Unix, по-скоро не е нужно да го правите изобщо.

В резултат на това можете да чуете спомена за безполезната употреба на наградата Cat (UUoC), известна в някои кръгове и като **наградата за най-безразборното използване на думата Word In Serious Shell Script, която** се съдържа в **comp.unix.shell**дискусионната група от от време на време. Това е единствено начин връстниците да се държат взаимно под контрол и да се уверят, че нещата са правилни.

Което ме води към нещо друго: Никога не *се* чувствайте твърде близо до собствените си скриптове; по своята същност източникът не може да бъде затворен. Ако предоставите на клиента скрипт, той / тя може да го провери доста лесно. Затова можете да приемете, че ще бъдете проверени от всеки, на когото го предавате; използвайте това в своя полза с [GPL](http://www.gnu.org/copyleft/gpl.html) - насърчете хората да ви дават обратна връзка и корекции за грешки безплатно!

# 3. Първи скрипт

За първия ни скрипт ще напишем скрипт, който казва "Hello World". След това ще се опитаме да   
извлечем повече от програмата Hello World, отколкото всеки друг урок, който сте прочели :-) Създайте файл (first.sh) както следва:

[first.sh](https://www.shellscript.sh/eg/first.sh.txt)

#!/bin/sh

# This is a comment!

echo Hello World # This is a comment, too!

Първият ред казва на Unix, че файлът трябва да бъде изпълнен от / bin / sh. Това е стандартното местоположение на обвивката на Bourne върху почти всяка Unix система. Ако използвате GNU / Linux, / bin / sh обикновено е символична връзка към bash (или по-скоро на dash).

Вторият ред започва със специален символ: **#**. Това маркира реда като коментар и се игнорира напълно от обвивката.   
Единственото изключение е, когато *първият* ред на файла започва с **#!**- както правим. Това е специална директива, която Unix третира специално. Това означава, че дори ако използвате csh, ksh или нещо друго като интерактивна обвивка, това, което следва, трябва да се интерпретира от обвивката на Bourne.   
По същия начин, Perl скрипт може да започне с линията, за **#!/usr/bin/perl**да каже на вашата интерактивна обвивка, че програмата, която следва, трябва да бъде изпълнена от perl. За програмирането на обвивката на Борн ще се придържаме**#!/bin/sh.**

Третият ред изпълнява команда:, **echo**с два параметъра, или аргументи - първата е **"Hello"**; вторият е **"World"**.   
Забележете, че **echo**автоматично ще постави едно пространство между неговите параметри.   
Най- **#**символ все още стои на коментар; # и всичко след него се игнорира от обвивката.

сега стартирайте, за **chmod 755 first.sh**да направите изпълнимия текстов файл и стартирайте **./first.sh**.   
След това екранът ви трябва да изглежда така:

$ chmod 755 first.sh  
$ ./first.sh  
Hello World  
$

Вероятно сте очаквали това! Можете дори да пуснете:

$ echo Hello World  
Hello World  
$

Сега нека направим няколко промени.   
Първо, имайте предвид, че **echo**поставя едно пространство между неговите параметри. Поставете няколко интервала между "Hello" и "World". Какво очаквате да бъде продукцията? Какво ще кажете за поставянето на TAB знак между тях?   
Както винаги с програмиране на черупки, опитайте го и вижте.   
Изходът е същият! Извикваме **echo**програмата с два аргумента; не го е грижа повече, отколкото **cp**за пропуските между тях. Сега променете кода отново:

#!/bin/sh  
# This is a comment!  
echo "Hello      World"       # This is a comment, too!

Този път работи. Вероятно сте очаквали и това, ако имате опит с други езици за програмиране. Но ключът към разбирането на това какво става с по-сложна команда и скрипт, е да се разбере и да се обясни: ЗАЩО?   
**echo**вече е наречен само с един аргумент - низ "Hello World". Това се отпечатва точно.   
Тук трябва да разберем, че черупката анализира аргументите ПРЕДИ да ги предаде на извиканата програма. В този случай той премахва кавичките, но предава низа като един аргумент.   
Като последен пример, напишете следния скрипт. Опитайте се да предвидите резултата, преди да го стартирате:

[first2.sh](https://www.shellscript.sh/eg/first2.sh.txt)

#!/bin/sh  
# This is a comment!  
echo "Hello      World"       # This is a comment, too!  
echo "Hello World"  
echo "Hello \* World"  
echo Hello \* World  
echo Hello      World  
echo "Hello" World  
echo Hello "     " World  
echo "Hello "\*" World"  
echo `hello` world  
echo 'hello' world

Дали всичко, което очаквате? Ако не, не се притеснявайте! Това са само някои от нещата, които ще покриваме в този урок ... и да, ще използваме по-мощни команди, отколкото **echo**!

# 4. Променливи - част I

Всяко съществуване на всеки програмен език има понятието за *променливи* - символично име за парче памет, към което можем да присвоим стойности, да четем и да манипулираме съдържанието му. Корпусът на Борн не е изключение и този раздел представя тази идея. Това е взето по-нататък в [променливите - част II,](https://www.shellscript.sh/variables2.html) която разглежда променливите, които са ни поставени от околната среда.   
Нека да погледнем назад към първия пример на Hello World. Това може да бъде направено с помощта на променливи (макар че е толкова прост пример, че това наистина не го оправдава!)   
Имайте предвид, че не трябва да има интервали около знака "=": **VAR=value**работи;**VAR = value**не работи. В първия случай, черупката вижда символа "=" и третира командата като присвояване на променлива. Във втория случай черупката приема, че VAR трябва да е името на команда и се опитва да я изпълни.   
Ако мислите за това, това има смисъл - как иначе бихте могли да му кажете да изпълни командата VAR, като първият му аргумент е "=" и вторият му аргумент е "стойност"?   
Въведете следния код в var.sh:

[var.sh](https://www.shellscript.sh/eg/var.sh.txt)

#!/bin/sh  
MY\_MESSAGE="Hello World"  
echo $MY\_MESSAGE

Това присвоява низ "Hello World" на променливата, **MY\_MESSAGE**след което **echo**извежда стойността на променливата.   
Имайте предвид, че се нуждаем от кавичките около низа Hello World. Докато бихме могли да се измъкнем, **echo Hello World**защото ехото ще поеме произволен брой параметри, една променлива може да притежава само една стойност, така че низ с интервали трябва да бъде цитиран така, че черупката да знае, че всичко се третира като едно. В противен случай, черупката ще се опита да изпълни командата **World** след присвояване**MY\_MESSAGE=Hello**

Черупката не се интересува от типовете променливи; те могат да съхраняват низове, цели числа, реални числа - всичко, което ви харесва.   
Хората, свикнали с Perl, може да са доволни от това; ако сте пораснали с C, Pascal или още по-лошо, Ada, това може да изглежда доста странно.   
В действителност, всички те се съхраняват като низове, но рутинните процедури, които очакват редица, могат да ги третират като такива.   
Ако присвоите низ на променлива, опитайте да добавите 1 към него, няма да се измъкнете с него:

$ x="hello"  
$ expr $x + 1+ 1  
expr: non-numeric argument  
$

Това е така, защото външната програма **expr**очаква само номера. Но няма синтактична разлика между:

MY\_MESSAGE="Hello World"  
MY\_SHORT\_MESSAGE=hi  
MY\_NUMBER=1  
MY\_PI=3.142  
MY\_OTHER\_PI="3.142"  
MY\_MIXED=123abc

Обърнете внимание, че специалните знаци трябва да бъдат правилно избягани, за да се избегне интерпретация от обвивката.   
Това е обсъдено по-нататък в Глава 6,  [Escape Characters](https://www.shellscript.sh/escape.html).

Можем да зададем имена на променливи с помощта на **read**командата; Следният скрипт ви пита за вашето име и ви поздравява лично:

[var2.sh](https://www.shellscript.sh/eg/var2.sh.txt)

#!/bin/sh  
echo What is your name?  
read MY\_NAME  
echo "Hello $MY\_NAME - hope you're well."

Марио Бачински ми любезно посочи, че първоначално съм пропуснал двойните кавички в ред 3, което означава, че единичният цитат в думата "вие" е несравним, причинявайки грешка. Това е нещо, което може да подлуди програмист на черупки, така че внимавайте за тях!

Това е командата shell-builtin, **read**която чете линия от стандартния вход в предоставената променлива.   
Обърнете внимание, че дори да му дадете пълното си име и да не използвате двойни кавички около **echo**командата, тя все още прави коректни изходи. Как се прави това? С **MY\_MESSAGE**променливата трябваше да сложим двойни кавички около нея, за да я зададем.   
Това, което се случва, е, че **read**командата автоматично поставя кавички около входа, така че пространствата да се третират правилно. (Трябва да цитирате изхода, разбира се - напр **echo "$MY\_MESSAGE"**.).

## **Обхват на променливите**

Променливите в обвивката на Bourne не трябва да се декларират, както в езиците като C. Но ако се опитате да прочетете недекларирана променлива, резултатът е празен низ. Не получавате предупреждения или грешки. Това може да предизвика някои фини бъгове - ако зададете   
**MY\_OBFUSCATED\_VARIABLE=Hello**  
и след   
**echo $MY\_OSFUCATED\_VARIABLE**   
това ще получите нищо (тъй като вторият OBFUSCATED е неправилно написан).

Има наречена команда, **export**която има фундаментален ефект върху обхвата на променливите. За да знаете какво наистина се случва с променливите ви, трябва да разберете нещо за това как се използва това.

Създайте малък скрипт на обвивката **myvar2.sh**:

[myvar2.sh](https://www.shellscript.sh/eg/myvar2.sh.txt)

#!/bin/sh  
echo "MYVAR is: $MYVAR"  
MYVAR="hi there"  
echo "MYVAR is: $MYVAR"

Стартирайте скрипта

$ ./myvar2.sh  
MYVAR is:  
MYVAR is: hi there

MYVAR не е настроена на никаква стойност, така че е празна. След това му даваме стойност и тя има очаквания резултат.   
Сега стартирайте:

$ MYVAR=hello  
$ ./myvar2.sh  
MYVAR is:  
MYVAR is: hi there

Тя все още не е настроена! Какво става?!   
Когато се обадите **myvar2.sh**от вашата интерактивна обвивка, се създава нова обвивка, за да стартирате скрипта. Това отчасти се дължи на **#!/bin/sh** линията в началото на скрипта, която обсъждахме [по-рано](https://www.shellscript.sh/first.html) .   
Необходимо **export**е променливата да бъде наследена от друга програма - включително шел скрипт. Тип:

$ export MYVAR  
$ ./myvar2.sh  
MYVAR is: hello  
MYVAR is: hi there

Сега вижте ред 3 на скрипта: това променя стойността на **MYVAR**. Но няма начин това да бъде предадено на интерактивната ви обвивка. Опитайте да прочетете стойността на **MYVAR**:

$ echo $MYVAR  
hello  
$

След като скриптът излезе, неговата среда е унищожена. Но **MYVAR**запазва стойността си в **hello** рамките на вашата интерактивна обвивка.   
За да получите среда се променя обратно от сценария, ние трябва да *си набави* сценария - това ефективно работи скрипта, в рамките на нашата интерактивна черупки, вместо хайвер друга черупка, за да го стартирате.   
Можем да създадем скрипт чрез "." Команда (точка):

$ MYVAR=hello  
$ echo $MYVAR  
hello  
$ . ./myvar2.sh  
MYVAR is: hello  
MYVAR is: hi there  
$ echo $MYVAR  
hi there

Промяната вече е излязла отново в нашата черупка! Така например работи вашето **.profile**или **.bash\_profile**файла.   
Обърнете внимание, че в този случай не е необходимо **export MYVAR**.   
Благодарение на *влиянието* , което посочих, че първоначално бях казал **echo MYVAR**по-горе, не **echo $MYVAR**както трябва. Друг пример за лесна грешка с черупки. Едно друго нещо, което си струва да се спомене в този момент за променливите, е да разгледаме следния скрипт:

#!/bin/sh  
echo "What is your name?"  
read USER\_NAME  
echo "Hello $USER\_NAME"  
echo "I will create you a file called $USER\_NAME\_file"  
touch $USER\_NAME\_file

Помислете какъв резултат бихте очаквали. Например, ако въведете „steve“ като свой USER\_NAME, трябва ли да създаде скрипт **steve\_file**?   
Всъщност не. Това ще доведе до грешка, освен ако не е извикана променлива **USER\_NAME\_file**. Черупката не знае къде свършва променливата и останалото започва. Как можем да определим това?   
Отговорът е, че ние включваме самата променлива в *фигурни скоби* :

[user.sh](https://www.shellscript.sh/eg/user.sh.txt)

#!/bin/sh  
echo "What is your name?"  
read USER\_NAME  
echo "Hello $USER\_NAME"  
echo "I will create you a file called ${USER\_NAME}\_file"  
touch "${USER\_NAME}\_file"

Понастоящем обвивката знае, че се отнасяме към променливата **USER\_NAME** и че искаме тя да бъде наложена с " **\_file**". Това може да бъде падението на много нови програмисти на скриптове, тъй като източникът на проблема може да бъде труден за проследяване.

Също така обърнете внимание на кавичките **"${USER\_NAME}\_file"**- ако потребителят е въвел "Стив Паркър" (обърнете внимание на мястото), тогава без кавичките, аргументите, които са били предадени, **touch**ще бъдат **Steve**и **Parker\_file**- това е, бихме казали ефективно **touch Steve Parker\_file**, което е два файла **touch**, не един. Котировките избягват това. Благодаря на Крис за това, че подчерта това.

# 5. Заместващи символи

Символите наистина не са нищо ново, ако преди сте използвали Unix.   
Не е непременно очевидно как те са полезни при скриптовете с черупки. Този раздел е наистина само за да накарате старите сиви клетки да мислят как изглеждат нещата, когато сте в скрипт - прогнозирате какъв е ефектът от използването на различни синтаксиси. Това ще се използва по-късно, особено в секцията [Loops](https://www.shellscript.sh/loops.html) .   
Мислете първо как да копирате всички файлове от **/tmp/a** в **/tmp/b**. Всички .txt файлове? Всички .html файлове?

$ cp / tmp / a / \* / tmp / b / $ cp / tmp / a / \*. txt / tmp / b / $ cp / tmp / a / \*. html / tmp / b /

Сега как бихте изброили файловете **/tmp/a/**без да ги използвате **ls /tmp/a/**?   
Какво ще кажете **echo /tmp/a/\***? Какви са двете ключови разлики между това и **ls**резултатите? Как това може да бъде полезно? Или пречка?   
Как може да преименувате всички .txt файлове на .bak? Отбележи, че

$ mv \*.txt \*.bak

няма да има желания ефект; Помислете как това се разширява от черупката, преди да бъде прехвърлена **mv**. Опитайте с това, **echo** вместо **mv**ако това ви помага.   
Ще разгледаме това по-късно, тъй като използва няколко концепции, които все още не са обхванати.

# 6. Escape Characters

Някои символи са важни за черупката; видяхме например, че използването на символи с двойни кавички (") влияе на начина, по който се третират пространствата и символите TAB, например:

$ echo Hello WorldHello World

Hello WorldHello World

$ echo "Hello World""Hello World"

Hello WorldHello World

И как ще покаже: **Hello    "World"**?

$ echo "Hello \"World\"""Hello \"World\""

Първият и последният "символи" обгръщат цялата партида в един параметър, който се предава, **echo**така че разстоянието между двете думи се запазва както е. Но кодът:

$ echo "Hello " World ""

"Hello " World ""

ще се тълкува като три параметъра:

1. "Здравейте "
2. свят
3. ""

Така че продукцията ще бъде

Hello World

World

Обърнете внимание, че губим изцяло котировките. Това е така, защото първата и втората кавички отразяват Hello и следващите пространства; Вторият аргумент е некотиран "Свят" и третият аргумент е празният низ; "".

Благодаря на Патрик за това, че това:

$ echo "Hello "World"""Hello "World""

всъщност е само един параметър (без интервали между цитираните параметри) и че можете да тествате това, като замените **echo**командата с (например) **ls**.

Повечето символи ( **\***, **'**и т.н.) не се интерпретират (т.е. те се вземат буквално) чрез поставянето им в двойни кавички (""). Те се вземат както е и се предават на командата, която се нарича. Пример за използване на звездичката (\*) е:

$ echo \*\*

case.shtml escape.shtml first.shtml case.shtml escape.shtml first.shtml

functions.shtml hints.shtml index.shtml .shtml hints.shtml index.shtml

ip-primer.txt raid1+0.txt-primer.txt raid1+0.txt

$ echo \*txt\*txt

ip-primer.txt raid1+0.txt-primer.txt raid1+0.txt

$ echo "\*""\*"

\*\*

$ echo "\*txt""\*txt"

\*txt\*txt

В първия пример, \* се разширява, за да означава всички файлове в текущата директория.   
Във втория пример \* txt означава всички файлове, завършващи на **txt**.   
В третия поставяме \* в двойни кавички и се тълкува буквално.   
В четвъртия пример се прилага същото, но ние сме добавили **txt**към низа.

Въпреки това, **"**, **$**, **`**, и **\**все още се тълкува от черупката, дори и когато те са в двойни кавички.   
Символът "обратно наклонена черта" се използва, за да се маркират тези специални символи, така че да не се интерпретират от обвивката, а да се прехвърлят към изпълняваната команда (например **echo**).   
Така че за извеждане на низ: (Ако приемем, че стойността **$X**е 5):

A quote is ", backslash is \, backtick is `.

A few spaces are and dollar is $. $X is 5.

we would have to write:

$ echo "A quote is \", backslash is \\, backtick is \`."

A quote is ", backslash is \, backtick is `.

$ echo "A few spaces are and dollar is \$. \$X is ${X}."

A few spaces are and dollar is $. $X is 5.

Видяхме защо "е специален за запазване на разстоянието. Dollar ( **$**) е специален, защото маркира променлива, така че **$X**е заменен от обвивката със съдържанието на променливата **X**. Обратната наклонена черта ( **\**) е специална, защото се използва за маркиране на други символи. ние се нуждаем от следните опции за пълна обвивка:

$ echo "This is \\ a backslash"

This is \ a backslash

$ echo "This is \" a quote and this is \\ a backslash"

This is " a quote and this is \ a backslash

So backslash itself must be escaped to show that it is to be taken literally. The other special character, the backtick, is discussed later in Chapter 12, [External Programs](https://www.shellscript.sh/external.html).

# 7. Loops

Повечето езици имат концепцията за цикли: Ако искаме да повторим задача двадесет пъти, не искаме да въвеждаме кода двадесет пъти, а може би с лека промяна всеки път.   
В резултат на това имаме **for**и **while**примки в обвивката на Борн. Това е малко по-малко функции от другите езици, но никой не твърди, че черупковото програмиране има силата на C.

## **За цикли**

**for** циклите се преливат през набор от стойности, докато списъкът не бъде изчерпан:

[for.sh](https://www.shellscript.sh/eg/for.sh.txt)

#!/bin/sh

for i in 1 2 3 4 5

do

echo "Looping ... number $i"

done

Опитайте този код и вижте какво прави. Обърнете внимание, че стойностите могат да бъдат всичко:

[for2.sh](https://www.shellscript.sh/eg/for2.sh.txt)

#!/bin/sh

for i in hello 1 \* 2 goodbye

do

echo "Looping ... i is set to $i"

done

Това си струва да се опита. Уверете се, че разбирате какво се случва тук. Опитайте го, без да го **\***схванете и след това прочетете отново секцията [Wildcards](https://www.shellscript.sh/wildcards.html) и опитайте отново с **\***мястото. Опитайте го и в различни директории, и с **\***заобиколени от двойни кавички, и опитайте го предшествано от обратна наклонена черта ( **\***)

В случай, че в момента нямате достъп до черупката (много е полезно да имате черупка на ръка, докато четете този урок), резултатите от горните два скрипта са:

Looping .... number 1

Looping .... number 2

Looping .... number 3

Looping .... number 4

Looping .... number 5

and, for the second example:

Looping ... i is set to hello

Looping ... i is set to 1

Looping ... i is set to (name of first file in current directory)

... etc ...

Looping ... i is set to (name of last file in current directory)

Looping ... i is set to 2

Looping ... i is set to goodbye

Така че, както можете да видите, **for**просто прескачате каквото и да е въвеждане, докато не изтече входа.

## **Докато циклите**

**while**примки могат да бъдат много по-забавно! (в зависимост от идеята ви за забавление и колко често излизате от къщата ...)

[while.sh](https://www.shellscript.sh/eg/while.sh.txt)

#!/bin/sh

INPUT\_STRING=hello

while [ "$INPUT\_STRING" != "bye" ]

do

echo "Please type something in (bye to quit)"

read INPUT\_STRING

echo "You typed: $INPUT\_STRING"

done

Това, което се случва тук, е, че ехото и четенето ще се изпълняват безкрайно, докато не напишете "bye", когато бъдете подканени.   
Преглед на [променливите - част I, за](https://www.shellscript.sh/variables1.html) да разберем защо сме го поставили **INPUT\_STRING=hello**преди да го тестваме. Това го прави повтарящ се цикъл, а не традиционен цикъл.

Двоеточие ( **:**) винаги се оценява на true; докато използването на това може да бъде необходимо понякога, често е за предпочитане да се използва реално изходно условие. Сравнете затварянето на цикъла по-горе с този по-долу; виж коя е по-елегантна. Също така помислете за някои ситуации, в които всеки от тях би бил по-полезен от другия:

[while2.sh](https://www.shellscript.sh/eg/while2.sh.txt)

#!/bin/sh

while :

do

echo "Please type something in (^C to quit)"

read INPUT\_STRING

echo "You typed: $INPUT\_STRING"

done

Друг полезен трик е **while read f**цикълът. Този пример използва декларацията за [случая](https://www.shellscript.sh/case.html) , която ще [разгледаме](https://www.shellscript.sh/case.html) по-късно. Той чете от файла **myfile**и за всеки ред ви казва кой език се използва. Всеки ред трябва да завършва с LF (нов ред) - ако **cat myfile**не завършва с празен ред, последният ред няма да бъде обработен.

[while3a.sh](https://www.shellscript.sh/eg/while3a.sh.txt)

#!/bin/sh

while read f

do

case $f in

hello) echo English ;;

howdy) echo American ;;

gday) echo Australian ;;

bonjour) echo French ;;

"guten tag") echo German ;;

\*) echo Unknown Language: $f

;;

esac

done < myfile

В много Unix системи това може да се направи и като:

[while3b.sh](https://www.shellscript.sh/eg/while3b.sh.txt)

#!/bin/sh

while f=`line`

do

.. process f ..

done < myfile

Но тъй като **while read f**работи с всички \* nix, и не зависи от външната програма **line**, първата е за предпочитане. Вж. [Външни програми, за](https://www.shellscript.sh/external.html) да разберете защо този метод използва обратното (`).   
Ако по-горе **$i**(не **$f**) споменах (не ) в случая по подразбиране („Неизвестен език“), нямаше да има предупреждения или грешки, въпреки че **$i**не беше обявен или дефиниран. Например:

$ i=THIS\_IS\_A\_BUG

$ export i

$ ./while3.sh something

Unknown Language: THIS\_IS\_A\_BUG

$

Така че се уверете, че избягвате правописни грешки. Това е и друга добра причина за използване, **${x}** а не просто **$x**- ако **x="A"**и искате да кажете "A1", трябва **echo ${x}1**, както **echo $x1**ще се опитате да използвате променливата **x1**, която може да не съществува, или може да бъде настроена на " **B2**," или нещо друго неочаквано ,

Наскоро намерих стара нишка в Usenet, в която участвах, където всъщност научих повече ... [Google я има тук.](http://groups.google.com/groups?hl=en&safe=off&selm=D9I1pp.JM%40info.bris.ac.uk),

A handy Bash (but not Bourne Shell) tip I learned recently from the [Linux From Scratch](http://www.linuxfromscratch.org/) project is:

mkdir rc{0,1,2,3,4,5,6,S}.d

вместо по-тромавите:

for runlevel in 0 1 2 3 4 5 6 S

do

mkdir rc${runlevel}.d

done

И това може да се направи и рекурсивно:

$ cd /

$ ls -ld {,usr,usr/local}/{bin,sbin,lib}

drwxr-xr-x 2 root root 4096 Oct 26 01:00 /bin

drwxr-xr-x 6 root root 4096 Jan 16 17:09 /lib

drwxr-xr-x 2 root root 4096 Oct 27 00:02 /sbin

drwxr-xr-x 2 root root 40960 Jan 16 19:35 usr/bin

drwxr-xr-x 83 root root 49152 Jan 16 17:23 usr/lib

drwxr-xr-x 2 root root 4096 Jan 16 22:22 usr/local/bin

drwxr-xr-x 3 root root 4096 Jan 16 19:17 usr/local/lib

drwxr-xr-x 2 root root 4096 Dec 28 00:44 usr/local/sbin

drwxr-xr-x 2 root root 8192 Dec 27 02:10 usr/sbin

# 8. Test

Тестът се използва от почти всеки написан скрипт. Може да не изглежда така, защото **test**често не се нарича директно. **test**по-често се нарича като **[**. **[**е символична връзка към **test**, само за да се направи четливите програми по-четливи. Също така обикновено е вграден shell (което означава, че самата обвивка ще интерпретира **[**като смисъл **test**, дори ако вашата Unix среда е настроена по различен начин):

$ type [

[ is a shell builtin

$ which [

/usr/bin/[

$ ls -l /usr/bin/[

lrwxrwxrwx 1 root root 4 Mar 27 2000 /usr/bin/[ -> test

$ ls -l /usr/bin/test

-rwxr-xr-x 1 root root 35368 Mar 27 2000 /usr/bin/test

Това означава, че " **[**" всъщност е програма, точно както **ls**и други програми, така че трябва да бъде заобиколена от интервали:

if [$foo = "bar" ]

няма да работи; той се приема **if test$foo = "bar" ]**, което е ' **]**' без начало " **[**. Поставете пространства около всичките си оператори. Подчертах задължителните интервали с думата „ПРОСТРАНСТВО“ - заменете „ПРОСТРАНСТВО“ с действително пространство; ако там няма място, то няма да работи:

if SPACE [ SPACE "$foo" SPACE = SPACE "bar" SPACE ]

Забележка: Някои черупки също приемат " **==**" за сравнение на низове; това не е преносимо, един " **=**" трябва да се използва за низове, или **-eq**за цели числа.

Тестът е прост, но мощен инструмент за сравнение. За пълни подробности, стартирайте **man test** на вашата система, но ето някои приложения и типични примери.

Тест най-често се позовава косвено чрез **if**и **while**отчети. Също така е и причината **test**да попаднете в затруднение, ако създадете програма, наречена и се опитате да я стартирате, тъй като тази вградена черупка ще се нарича вместо вашата програма!   
Синтаксисът за **if...then...else...**е:

if [ ... ]

then

# if-code

else

# else-code

fi

Забележете, че **fi**е **if**назад! Това се използва отново по-късно в [случай](https://www.shellscript.sh/case.html) и **esac**.   
Също така, имайте предвид синтаксиса - " **if [ ... ]**" и **then**командите трябва да са на различни линии. Алтернативно, точка и запетая **;**могат да ги разделят: "

if [ ... ]; then

# do something

fi

Можете също да използвате **elif**следното:

if [ something ]; then

echo "Something"

elif [ something\_else ]; then

echo "Something else"

else

echo "None of the above"

fi

Това ще **echo "Something"**стане, ако **[ something ]**тестът успее, в противен случай ще тества **[ something\_else ]**и **echo "Something else"**ако успее. Ако всичко друго се провали, ще го направи **echo "None of the above"**.

Опитайте следния кодов фрагмент, преди да го стартирате, задайте променливата X на различни стойности (опитайте -1, 0, 1, hello, bye и т.н.). Можете да направите това по следния начин (благодарение на Дейв за посочване на необходимостта от експортиране на променливата, както е отбелязано в [Променливи - Част I.](https://www.shellscript.sh/variables1.html) ):

$ X=5

$ export X

$ ./test.sh

... output of test.sh ...

$ X=hello

$ ./test.sh

... output of test.sh ...

$ X=test.sh

$ ./test.sh

... output of test.sh ...

Then try it again, with **$X** as the name of an existing file, such as **/etc/hosts**.

[test.sh](https://www.shellscript.sh/eg/test.sh.txt)

#!/bin/sh

if [ "$X" -lt "0" ]

then

echo "X is less than zero"

fi

if [ "$X" -gt "0" ]; then

echo "X is more than zero"

fi

[ "$X" -le "0" ] && \

echo "X is less than or equal to zero"

[ "$X" -ge "0" ] && \

echo "X is more than or equal to zero"

[ "$X" = "0" ] && \

echo "X is the string or number \"0\""

[ "$X" = "hello" ] && \

echo "X matches the string \"hello\""

[ "$X" != "hello" ] && \

echo "X is not the string \"hello\""

[ -n "$X" ] && \

echo "X is of nonzero length"

[ -f "$X" ] && \

echo "X is the path of a real file" || \

echo "No such file: $X"

[ -x "$X" ] && \

echo "X is the path of an executable file"

[ "$X" -nt "/etc/passwd" ] && \

echo "X is a file which is newer than /etc/passwd"

Обърнете внимание, че можем да използваме точка и запетая ( **;**), за да свържем две линии заедно. Това често се прави, за да се спести малко място в прости **if**изявления. Обратната   
наклонена черта ( **\**) служи за подобна, но противоположна цел: тя казва на обвивката, че това не е краят на реда, но че следващият ред трябва да се третира като част от текущия ред. Това е полезно за четливост. Обичайно е да отстъпите следния ред след обратна наклонена черта ( **\**) или точка и запетая ( **;**).

Например, точка и запетая ( **;**) често се използва по този начин да се присъединят към **if**и **then**ключови думи:

if [ "$X" -nt "/etc/passwd" ]; then

echo "X is a file which is newer than /etc/passwd"

fi

докато обратната червена черта ( **\**) се използва за разделяне на командата на една линия върху два реда в файла на скрипта за четене:

[ "$X" -nt "/etc/passwd" ] && \

echo "X is a file which is newer than /etc/passwd"

Както виждаме от тези примери, **test**може да извърши много тестове на числа, низове и имена на файлове.

Благодарение на Aaron за това **-a**, че **-e**(и двете означават „файл съществува“), **-S**(файлът е Socket), **-nt**(файлът е по-нов от), **-ot**(файлът е по-стар от), **-ef**(пътищата се отнасят за един и същ файл) и **-O**(файлът е собственост на моя потребител), не са достъпни в традиционната обвивка на Bourne (напр. / bin / sh на Solaris, AIX, HPUX и т.н.).

Има по-лесен начин за писане на **if**отчети: В **&&**и **||**команди съобщи кода да се изпълнява, ако резултатът е вярно или невярно, съответно.

#!/bin/sh

[ $X -ne 0 ] && echo "X isn't zero" || echo "X is zero"

[ -f $X ] && echo "X is a file" || echo "X is not a file"

[ -n $X ] && echo "X is of non-zero length" || \

echo "X is of zero length"

Този синтаксис е възможен, защото има извикан файл (или shell-builtin), към **[**който е свързан **test**. Бъдете внимателни, като използвате този конструкт, макар че прекалената употреба може да доведе до много трудноразбираем код. В **if...then...else...**структурата е много по-разбираемо. Използването на **[...]**конструкцията се препоръчва, докато цикли и тривиални проверки за здрав разум, с които не искате да отвличате внимателно читателя.

Обърнете внимание, че когато зададете X на не-числова стойност, първите няколко сравнения водят до съобщението:

test.sh: [: integer expression expected before -lt

test.sh: [: integer expression expected before -gt

test.sh: [: integer expression expected before -le

test.sh: [: integer expression expected before -ge

Това е така, защото сравненията -lt, -gt, -le и -ge са предназначени само за цели числа и не работят по низове. Сравненията на низове, като например **!=**щастливо ще третират "5" като низ, но няма разумен начин за третиране на "Hello" като цяло число, така че целочислените сравнения се оплакват.   
Ако искате вашият скрипт да се държи по-грациозно, ще трябва да проверите съдържанието на променливата, преди да го тествате - може би нещо подобно:

echo -en "Please guess the magic number: "

read X

echo $X | grep "[^0-9]" > /dev/null 2>&1

if [ "$?" -eq "0" ]; then

# If the grep found something other than 0-9

# then it's not an integer.

echo "Sorry, wanted a number"

else

# The grep found only 0-9, so it's an integer.

# We can safely do a test on it.

if [ "$X" -eq "7" ]; then

echo "You entered the magic number!"

fi

fi

По този начин можете да **echo**получите по-смислено послание за потребителя и да излезете грациозно. В **$?**променливата е обяснено в [Променливи - част II](https://www.shellscript.sh/variables2.html) , и **grep**е сложен звяр, така че тук върви: **grep [0-9]**намира реда текст, който да съдържа цифри (0-9) и евентуално други знаци, така че точката на вмъкване ( **^**) в **grep [^0-9]**открие само тези линии, които дон " t се състои само от числа. Тогава можем да вземем обратното (като действаме на провал, а не на успех). Добре? Той **>/dev/null 2>&1**насочва всеки изход или грешки към специално "null" устройство, вместо да отиде на екрана на потребителя.   
Много благодарности на Пол Schermerhorn за поправянето ми - тази страница е използвана, за да твърди, че **grep -v [0-9]**ще работи, но това очевидно е твърде опростено.

Можем да използваме тест в циклично време, както следва:

[test2.sh](https://www.shellscript.sh/eg/test2.sh.txt)

#!/bin/sh

X=0

while [ -n "$X" ]

do

echo "Enter some text (RETURN to quit)"

read X

echo "You said: $X"

done

Този код ще продължава да пита за въвеждане, докато не натиснете RETURN (X е нулева дължина). Благодарение на Джъстин Хийт, че посочи, че скриптът не работи - пропуснах цитатите около $ X в **while [ -n "$X" ]**. Без тези цитати няма нищо, което да тества, когато $ X е празно.   
Александър Вебер посочи, че стартирането на този скрипт ще завърши неочаквано:

$ ./test2.sh

Enter some text (RETURN to quit)

fred

You said: fred

Enter some text (RETURN to quit)

wilma

You said: wilma

Enter some text (RETURN to quit)

You said:

$

Това може да се приключи с друг тест в цикъла:

#!/bin/sh

X=0

while [ -n "$X" ]

do

echo "Enter some text (RETURN to quit)"

read X

if [ -n "$X" ]; then

echo "You said: $X"

fi

done

Имайте предвид също, че съм използвал два различни синтаксиса за **if**изявления на тази страница. Това са:

if [ "$X" -lt "0" ]

then

echo "X is less than zero"

fi

.......... and ........

if [ ! -n "$X" ]; then

echo "You said: $X"

fi

Трябва да имате прекъсване между **if**изявлението и **then**конструкцията. Това може да бъде точка или запетая или нов ред, няма значение кой, но трябва да има един или друг между **if**и **then**. Би било хубаво просто да кажем:

if [ ! -n "$X" ]

echo "You said: $X"

но **then**и **fi**са абсолютно необходими.

# 9. Case

В **case**изявлението спестява преживява цял набор от **if .. then .. else**отчети. Синтаксисът му е наистина много прост:

[talk.sh](https://www.shellscript.sh/eg/talk.sh.txt)

#!/bin/sh

echo "Please talk to me ..."

while :

do

read INPUT\_STRING

case $INPUT\_STRING in

hello)

echo "Hello yourself!"

;;

bye)

echo "See you again!"

break

;;

\*)

echo "Sorry, I don't understand"

;;

esac

done

echo

echo "That's all folks!"

Добре, значи не е най-добрият събеседник в света; това е само пример!

Опитайте да я пуснете и проверете как работи ...

$ ./talk.sh

Please talk to me ...

hello

Hello yourself!

What do you think of politics?

Sorry, I don't understand

bye

See you again!

That's all folks!

$

Синтаксисът е съвсем прост:   
Самата **case**линия винаги е от един и същ формат и означава, че ние тестваме стойността на променливата **INPUT\_STRING**.

След това опциите, които разбираме, са изброени и последвани от дясна скоба, както **hello)**и **bye)**.   
Това означава, че ако **INPUT\_STRING**съвпада, **hello**тогава секцията от кода се изпълнява до двойната точка и запетая.   
Ако **INPUT\_STRING**съвпада, **bye**тогава съобщението за сбогом се отпечатва и цикълът излиза. Забележете, че ако искаме да излезем от скрипта напълно, ще използваме командата **exit** вместо **break**.   
Третата опция тук **\*)**е условието за подмяна по подразбиране; то не е задължително, но често е полезно за отстраняване на грешки, дори ако мислим, че знаем какви стойности ще има променливата.

Целият случай се завършва с **esac**(случай назад!), След което завършваме цикъла while с a **done**.

Това е толкова сложно, колкото **case**условията, но те могат да бъдат много полезен и мощен инструмент. Често те се използват за разбор на параметрите, предавани на скрипт на обвивката, наред с други приложения.

# Променливи - част II

Има набор от променливи, които вече са зададени за вас и повечето от тях не могат да имат присвоени им стойности.   
Те могат да съдържат полезна информация, която може да се използва от скрипта, за да се знае за средата, в която се изпълнява.

Първият набор от променливи, които ще разгледаме, са **$0 .. $9** и **$#**.   
Променливата **$0**е *основното име* на програмата, както се нарича.   
**$1 .. $9**са първите 9 допълнителни параметъра, с които е извикан скрипта.   
Променливата **$@**е всички параметри **$1 .. whatever**. Променливата **$\***е сходна, но не запазва никое празно пространство и цитира, така че "Файл с интервали" става "Файл" с "" пространства. Това е подобно на **echo**нещата, които разглеждахме в [A Script](https://www.shellscript.sh/first.html) . Като общо правило, използвайте **$@**и избягвайте **$\***.   
**$#**е броят на параметрите, с които е извикан скрипта.   
Да вземем примерен скрипт:

[var3.sh](https://www.shellscript.sh/eg/var3.sh.txt)

#!/bin/sh

echo "I was called with $# parameters"

echo "My name is $0"

echo "My first parameter is $1"

echo "My second parameter is $2"

echo "All parameters are $@"

Нека да разгледаме изпълнението на този код и да видим изхода:

$ /home/steve/var3.sh

I was called with 0 parameters

My name is /home/steve/var3.sh

My first parameter is

My second parameter is

All parameters are

$

$ ./var3.sh hello world earth

I was called with 3 parameters

My name is ./var3.sh

My first parameter is hello

My second parameter is world

All parameters are hello world earth

Обърнете внимание, че стойността на **$0**промените зависи от начина на извикване на скрипта. Външната програма **basename**може да помогне за това:

echo "My name is `basename $0`"

**$#**и **$1 .. $9**се задават автоматично от черупката.   
Можем да вземем повече от 9 параметъра, като използваме **shift** командата; погледнете скрипта по-долу:

[var4.sh](https://www.shellscript.sh/eg/var4.sh.txt)

#!/bin/sh

while [ "$#" -gt "0" ]

do

echo "\$1 is $1"

shift

done

Този скрипт продължава да се използва, **shift**докато **$#**не достигне нула, след което списъкът е празен.

Друга специална променлива е **$?**. Това съдържа стойността на изхода на последната команда за изпълнение. Така че кодът:

#!/bin/sh

/usr/local/bin/my-command

if [ "$?" -ne "0" ]; then

echo "Sorry, we had a problem there!"

fi

ще се опита да тече, **/usr/local/bin/my-command**което трябва да излезе със стойност нула, ако всичко върви добре, или ненулева стойност при неуспех. След това можем да се справим с това като проверим стойността на **$?**след извикване на командата. Това помага на скриптовете да бъдат стабилни и по-интелигентни.   
Добре възприетите приложения трябва да връщат нула при успех. Оттук и цитат:

Една от основните причини за падането на Римската империя е, че без нула те нямат начин да посочат успешното прекратяване на техните програми. (Робърт Фирт)

Другите две основни променливи, определени за вас от околната среда, са **$$** и **$!**. Това са и двата вида процеси.   
В **$$** променливата е PID (Процес идентификатор) от момента работи обвивка. Това може да бъде полезно за създаването на временни файлове, като например това, **/tmp/my-script.$$**което е полезно, ако много копия на скрипта могат да бъдат стартирани едновременно и всички те се нуждаят от свои временни файлове.

Най- **$!**променлива е PID на процеса на фона последния цикъл. Това е полезно, за да следите процеса, докато той работи с работата си.

Друга интересна променлива е **IFS**. Това е *вътрешният сепаратор на полето* . Стойността по подразбиране е **SPACE TAB NEWLINE**, но ако я променяте, е по-лесно да вземете копие, както е показано:

[var5.sh](https://www.shellscript.sh/eg/var5.sh.txt)

#!/bin/sh

old\_IFS="$IFS"

IFS=:

echo "Please input some data separated by colons ..."

read x y z

IFS=$old\_IFS

echo "x is $x y is $y z is $z"

This script runs like this:

$ ./ifs.sh

Please input some data separated by colons ...

hello:how are you:today

x is hello y is how are you z is today

Обърнете внимание, че ако въведете: "Здравейте: как сте: днес: моят: приятел", изходът ще бъде:

$ ./ifs.sh

Please input some data separated by colons ...

hello:how are you:today:my:friend

x is hello y is how are you z is today:my:friend

Важно е, когато става дума за IFS в частност (но всяка променлива, която не е изцяло под вашия контрол), за да осъзнаете, че тя може да съдържа интервали, нови линии и други "неконтролируеми" символи. Ето защо е много добра идея да използвате двойни кавички около нея, т.е.: **old\_IFS="$IFS"**вместо **old\_IFS=$IFS**.

# Променливи - част III

Както споменахме в [променливите - част I](https://www.shellscript.sh/variables1.html) , къдрави скоби около променлива избягват объркване:

foo=sun

echo $fooshine # $fooshine is undefined

echo ${foo}shine # displays the word "sunshine"

Това не е всичко - тези фантастични скоби имат друга, много по-мощна употреба. Можем да се справим с въпросите на променливите, които са неопределени или нула (в черупката няма голяма разлика между undefined и null).

#### **Използване на стойностите по подразбиране**

Помислете за следния кодов фрагмент, който подканва потребителя за въвеждане, но приема стандартните стойности:

#!/bin/sh

echo -en "What is your name [ `whoami` ] "

read myname

if [ -z "$myname" ]; then

myname=`whoami`

fi

echo "Your name is : $myname"

Предаването на " **-en**" на ехото му казва да не добавя разделителна линия (за bash и csh). За Dash, Bourne и други съвместими черупки **\c**, вместо това използвате " " в края на реда. Ksh разбира и двете форми. (Забележка: вижте [/echo.html](https://www.shellscript.sh/echo.html) за бележка за различни реализации - особено Dash / Bourne vs Bash)   
Този скрипт се изпълнява по следния начин, ако приемете по подразбиране, като натиснете "RETURN":

steve$ ./name.sh

What is your name [ steve ]

Your name is : steve

... or, with user input:

steve$ ./name.sh

What is your name [ steve ] **foo**

Your name is : foo

Това може да бъде направено по-добре с помощта на функция за променлива на обвивката. Чрез използването на фигурни скоби и специалната употреба ": -" можете да посочите стойност по подразбиране, която да се използва, ако променливата е изключена:

echo -en "What is your name [ `whoami` ] "

read myname

echo "Your name is : ${myname:-`whoami`}"

Това може да се счита за специален случай - използваме изхода на командата whoami, който отпечатва вашето потребителско име (UID). По-каноничният пример е да се използва фиксиран текст, като този:

echo "Your name is : ${myname:-John Doe}"

Както и при другото използване на backticks, **`whoami`**изпълнява се в подсечка, така че всички **cd**команди или настройка на други променливи, в рамките на backticks, няма да засегнат текущо изпълняваната обвивка.

#### **Използване и задаване на стойности по подразбиране**

Има още един синтаксис, ": =", който задава променливата по подразбиране, ако е неопределена:

echo "Your name is : ${myname:=John Doe}"

Тази техника означава, че всеки последващ достъп до **$myname**променливата винаги ще получи стойност, или въведена от потребителя, или "John Doe" в противен случай.

# 12. External Programs

Външните програми често се използват в скриптове; има няколко вградени команди ( **echo**, **which**и **test** обикновено са вградени), но много полезни команди са всъщност Unix комунални услуги, като например **tr**, **grep**, **expr**и **cut**.

Backtick (`) също често се свързва с външни команди. Поради това първо ще обсъдим обратното.   
Backtick се използва, за да укаже, че приложеният текст трябва да се изпълни като команда. Това е доста лесно за разбиране. Първо използвайте интерактивна обвивка, за да прочетете пълното си име от **/etc/passwd**:

$ grep "^${USER}:" /etc/passwd | cut -d: -f5

Steve Parker

Сега ще извлечем този изход в променлива, която можем да манипулираме по-лесно:

$ MYNAME=`grep "^${USER}:" /etc/passwd | cut -d: -f5`

$ echo $MYNAME

Steve Parker

Така че ние виждаме, че backtick просто хваща стандартния изход от всяка команда или набор от команди, които избираме да изпълняваме. Тя може също така да подобри производителността, ако искате да стартирате бавна команда или набор от команди и да анализирате различни битове на неговия изход:

#!/bin/sh

find / -name "\*.html" -print | grep "/index.html$"

find / -name "\*.html" -print | grep "/contents.html$"

Този код може да отнеме много време за изпълнение и ние го правим два пъти!   
По-добро решение е:

#!/bin/sh

HTML\_FILES=`find / -name "\*.html" -print`

echo "$HTML\_FILES" | grep "/index.html$"

echo "$HTML\_FILES" | grep "/contents.html$"

Забележка: кавичките **$HTML\_FILES**са от съществено значение за запазване на новите редове между всеки изброени файлове. В противен случай **grep**ще видите един огромен дълъг ред текст, а не един ред на файл.

По този начин ще изпълняваме само бавно **find**веднъж, приблизително наполовина по време на изпълнението на скрипта.

# 13. Функции

Една често пренебрегвана функция на програмирането на скриптовете на Борн е, че можете лесно да пишете функции за използване в скрипта. Това обикновено се прави по един от двата начина; с прост скрипт, функцията просто се декларира в същия файл, както се нарича.   
Обаче, когато пишете набор от скриптове, често е по-лесно да напишете "библиотека" от полезни функции и да изворите този файл в началото на другите скриптове, които използват функциите. Това ще бъде показано [по-късно](https://www.shellscript.sh/functions.html#libraries) .   
Методът е същият, какъвто е направен; първо ще използваме първия път тук. Вторият (библиотечен) метод е принципно същият, с изключение на командата

, ./library.sh

отива в началото на скрипта.

Може да има известно объркване относно това дали да се наричат *процедури* или *функции на функциите на* черупката ; дефиницията на функция е традиционно, че тя връща единична стойност и не извежда нищо. Процедурата, от друга страна, не връща стойност, но може да генерира изход. Функцията на черупката не може да прави нито едно, нито едното или и двете. Общоприето е, че в shell скриптовете те се наричат ​​функции.

Функцията може да върне стойност по един от следните четири начина:

* Промяна на състоянието на променлива или променливи
* Използвайте **exit**командата, за да прекратите скрипта
* Използвайте **return**командата, за да прекратите функцията, и да върнете доставената стойност към разговорната част на скрипта
* ехо изход към stdout, който ще бъде заловен от повикващия точно както е хванат c = `expr $ a + $ b`

Това е подобно на C, в което **exit**спира програмата и **return**връща контрола на повикващия. Разликата е, че черупката не може да променя своите параметри, въпреки че може да променя глобалните параметри.

Един прост скрипт с функция ще изглежда така:

[function.sh](https://www.shellscript.sh/eg/function.sh.txt)

#!/bin/sh

# A simple script with a function...# A simple script with a function...

add\_a\_user()()

{{

USER=$1=$1

PASSWORD=$2=$2

shift; shift;; shift;

# Having shifted twice, the rest is now comments ...# Having shifted twice, the rest is now comments ...

COMMENTS=$@=$@

echo "Adding user $USER ...""Adding user $USER ..."

echo useradd -c "$COMMENTS" $USER-c "$COMMENTS" $USER

echo passwd $USER $PASSWORD

echo "Added user $USER ($COMMENTS) with pass $PASSWORD""Added user $USER ($COMMENTS) with pass $PASSWORD"

}}

######

# Main body of script starts here# Main body of script starts here

######

echo "Start of script...""Start of script..."

add\_a\_user bob letmein Bob Holness the presenterBob Holness the presenter

add\_a\_user fred badpassword Fred Durst the singerFred Durst the singer

add\_a\_user bilko worsepassword Sgt. Bilko the role modelSgt. Bilko the role model

echo "End of script...""End of script..."

Ред 4 се идентифицира като декларация на функция, като завършва с (). Това е последвано от **{**, и всичко, следващо съчетаването, **}**се приема като код на тази функция.   
Този код *не се изпълнява* до извикването на функцията. Функциите се четат, но основно се игнорират, докато не се наричат.

Имайте предвид, че за този пример **useradd**и **passwd** командите са с представка **echo**- това е полезна техника за отстраняване на грешки, за да се провери дали правилните команди ще бъдат изпълнени. Това също така означава, че можете да стартирате скрипта, без да сте root или да добавяте измамни потребителски акаунти към вашата система!

Бяхме свикнали с идеята, че шел скрипт се изпълнява последователно. Това не е така с функциите.   
В този случай функцията **add\_a\_user**се чете и проверява за синтаксис, но не се изпълнява, докато не се извика изрично. Това е мястото, където се появява грешката на [Shellshock](http://steve-parker.org/articles/shellshock/) от 2014 година. Други команди след дефинирането на функцията бяха изпълнени, въпреки че те не бяха част от самата функция. Вижте <http://steve-parker.org/articles/shellshock/> за повече информация за това.   
Изпълнението започва с **echo**израза "Старт на скрипта ...". Следващият ред **add\_a\_user bob letmein Bob Holness**се разпознава като извикване на **add\_a\_user**функция, така че функцията се въвежда и започва да се изпълнява с определени допълнения към околната среда:

$1=bob

$2=letmein

$3=Bob

$4=Holness

$5=the

$6=presenter

Така че в рамките на тази функция, **$1**е настроено на **bob**, независимо от това, което **$1**може да бъде настроено извън функцията.   
Така че, ако искаме да се позовем на "оригиналния" $ 1 *вътре* във функцията, трябва да му присвоим име - като например: **A=$1**преди да извикаме функцията. Тогава, в рамките на функцията, можем да се позовем **$A**.   
Използваме **shift**командата отново, за да **$3**въведем и напред параметрите **$@**. След това функцията добавя потребителя и задава неговата парола. Това **echo**е коментар за този ефект и връща контрола на следващия ред на основния код.

## **Обхват на променливите**

Програмистите, използвани за други езици, могат да бъдат изненадани от правилата за обхвата на функциите на черупката. По принцип, няма обхвата, с изключение на параметрите ( **$1**, **$2**, **$@**, и т.н.).   
Вземане на следния прост сегмент код:

#!/bin/sh

myfunc()

{

echo "I was called as : $@"

x=2

}

### Main script starts here

echo "Script was called with $@"

x=1

echo "x is $x"

myfunc 1 2 3

echo "x is $x"

The script, when called as **scope.sh a b c**, gives the following output:

Script was called with a b c

x is 1

I was called as : 1 2 3

x is 2

На **$@**параметри се променят в рамките на функцията, за да отрази как е извикана функцията. Променливата **x**, обаче, е действително глобална променлива - **myfunc**променя я и тази промяна е все още ефективна, когато контролът се връща към основния скрипт.

Функцията ще бъде извикана в под-обвивката, ако нейният изход е свързан към друго място, т.е. " **myfunc 1 2 3 | tee out.log**" ще продължи да казва "x is 1" втори път. Това е така, защото се нарича нов процес на обвивка **myfunc()**. Това може да направи разтоварването много разочароващо; Астрид имаше скрипт, който внезапно се провали при **| tee**добавянето на " " и не е веднага очевидно защо това трябва да бъде. Трябва **tee**да се стартира преди функцията отляво на тръбата; с прост пример за " **ls | grep foo**", след това **grep**трябва да бъде стартирана на първо място, с **stdin**след това свързан с **stdout**от **ls**един път **ls**стартира. В обвивката на обвивката, обвивката вече е стартирана, преди дори да сме знаели, че ще прескочим **tee**, така че операционната система трябва да започне**tee**, след това стартирайте нова черупка, за да се обадите **myfunc()**. Това е разочароващо, но си струва да бъдем наясно.

Функциите също не могат да променят стойностите, с които са били извикани - това трябва да се направи, като се променят самите променливи, а не параметрите, предадени на скрипта.   
Един пример показва това по-ясно:

#!/bin/sh

myfunc()

{

echo "\$1 is $1"

echo "\$2 is $2"

# cannot change $1 - we'd have to say:

# 1="Goodbye Cruel"

# which is not a valid syntax. However, we can

# change $a:

a="Goodbye Cruel"

}

### Main script starts here

a=Hello

b=World

myfunc $a $b

echo "a is $a"

echo "b is $b"

Тази доста цинична функция променя $ a, така че съобщението "Hello World" става "Goodbye Cruel World".

## **рекурсия**

Функциите могат да бъдат рекурсивни - ето един прост пример за факторна функция:

[factorial.sh](https://www.shellscript.sh/eg/factorial.sh.txt)

#!/bin/sh

factorial()

{

if [ "$1" -gt "1" ]; then

i=`expr $1 - 1`

j=`factorial $i`

k=`expr $1 \\* $j`

echo $k

else

echo 1

fi

}

while :

do

echo "Enter a number:"

read x

factorial $x

done

Както обещахме, сега ще обсъдим накратко използването на библиотеки между скриптовете на обвивката. Те също могат да се използват за определяне на общи променливи, както ще видим.

[common.lib](https://www.shellscript.sh/eg/common.lib.txt)

# common.lib

# Note no #!/bin/sh as this should not spawn

# an extra shell. It's not the end of the world

# to have one, but clearer not to.

#

STD\_MSG="About to rename some files..."

rename()

{

# expects to be called as: rename .txt .bak

FROM=$1

TO=$2

for i in \*$FROM

do

j=`basename $i $FROM`

mv $i ${j}$TO

done

}

[function2.sh](https://www.shellscript.sh/eg/function2.sh.txt)

#!/bin/sh

# function2.sh

. ./common.lib

echo $STD\_MSG

rename .txt .bak

[function3.sh](https://www.shellscript.sh/eg/function3.sh.txt)

#!/bin/sh

# function3.sh

. ./common.lib

echo $STD\_MSG

rename .html .html-bak

Тук виждаме два потребителски скрипта, **function2.sh**и **function3.sh**всеки *източник* в общия библиотечен файл **common.lib**и използвайки променливи и функции, декларирани в този файл.   
Това не е нищо прекалено разрушаващо земята, само пример за това как повторното използване на код може да се направи в черупковото програмиране.

## **Кодове за връщане**

За подробности относно кодовете за излизане вижте частта " [Изходни кодове"](https://www.shellscript.sh/exitcodes.html) в раздела " [Съвети и съвети](https://www.shellscript.sh/hints.html) " на урока. Засега, макар и накратко, да разгледаме **return**обаждането.

#!/bin/sh

adduser()

{

USER=$1

PASSWORD=$2

shift ; shift

COMMENTS=$@

useradd -c "${COMMENTS}" $USER

if [ "$?" -ne "0" ]; then

echo "Useradd failed"

return 1

fi

passwd $USER $PASSWORD

if [ "$?" -ne "0" ]; then

echo "Setting password failed"

return 2

fi

echo "Added user $USER ($COMMENTS) with pass $PASSWORD"

}

## Main script starts here

adduser bob letmein Bob Holness from Blockbusters

ADDUSER\_RETURN\_CODE=$?

if [ "$ADDUSER\_RETURN\_CODE" -eq "1" ]; then

echo "Something went wrong with useradd"

elif [ "$ADDUSER\_RETURN\_CODE" -eq "2" ]; then

echo "Something went wrong with passwd"

else

echo "Bob Holness added to the system."

fi

Този скрипт проверява двете външни повиквания, които прави ( **useradd**и **passwd**), и позволява на потребителя да знае, ако те се провалят. След това функцията дефинира код за връщане от 1, за да укаже на всеки проблем с **useradd**, и 2, за да покаже някакъв проблем с **passwd**. По този начин извикващият скрипт знае къде е проблемът.

Дълго време този урок е проверявал "$?" и двата пъти, а не настройката **ADDUSER\_RETURN\_CODE=$?**, и след това да се гледа стойността на **ADDUSER\_RETURN\_CODE**всеки път. Това е бъг (благодарение на Елиза за това, че го посочи). Трябва да запишете **$?**, защото веднага щом стартирате друга команда, като например **if**, нейната стойност ще бъде заменена. Ето защо запазваме **adduser**връщаната стойност в **$ADDUSER\_RETURN\_CODE**променливата, преди да действаме върху нейното съдържание. **$ADDUSER\_RETURN\_CODE**със сигурност ще остане същото; **$?**ще се промени с всяка изпълнена команда.