Ръководство по Perl

Версия ***5.0.0 – 19.06.2006***

Димитър Пламенов Михайлов

Предговор

Нямам филологическо образование и затова в текста може да срещнете стилистични или други грешки за което моля да бъда предварително извинен. Това ръководство съм го създал с цел хората интересуващи се от **Perl** да получат безплатна и качествена информация. Ще приема всякакви препоръки за развитието на това ръководство. Надявам се да ми изпращате текстове, които смятате, че могат да обогатят ръководството. Моля ако забележите грешки, пишете ми за тях за да ги отстраня.

Специални благодарности за намирането на такива грешки на ***Станислав Захариев***.

e-mail: mitko_ddt@yahoo.com

СЪДЪРЖАНИЕ

Начални сведения

Какво трябва да знаете и какво е нужно да имате?

Не Ви е нужен опит в програмирането за да започнете изучаването на **Perl.** Нужно е да разбирате основните компютърни операции**.** Ако не знаете какво са директории или файлове изучаването на езика ще Ви се стори трудно**.** За да започнете да програмирате на **Perl** Ви е нужно **PC,** което може да стартира **Win32** операционна система**, Linux, Unix, OS/2** и др**. (**ръководството е написано за потребители на **Win32). Win32** са **Windows NT 3.5, 3.51, 4.0** или по**-**голяма версия**, Windows 95, Windows 98, Windows 2000** или **Windows XP.** Сигурно няма да имате проблеми ако използвате **Windows 3.1 (**но не ми се вярва да работите с тази версия**).** Нужно е да си набавите копие на **Perl,** и затова може би Ви е нужен интернет**.** На сайта **http://www.activestate.com/** може да намерите и свалите последната версия на **Perl.** Компанията **ActiveState** е основният дистрибутор на **Perl** за **Win32.** Там ще намерите файл с име подобно на **ActivePerl-5.8.0.802-MSWin32-x86.msi.** Става въпрос за операционна система **Win32** и **Intel** процесор**,** не **Alpha. 5.8.0** в случая е версията на **Perl.** Инсталацията е подобна на всяка друга под **Windows.** Може да свалите и сорскодовете на езика**,** който да ги компилирате на машината си**.**

*На сайта **http://bgperl.tripod.com**, Ви предлагам диск със всичко необходимо за да започнете да програмирате на **Perl -** софтуер**,** редактори**,** достатъчно количество литература. Също така ще намерите и други езици и книги за тях. На диска се съдържат всички книги**,** който можете да намерите на диска със **Perl** който се продава на пл. Славейков **+** много друга събирана от мен в продължение на **2** година. Също така всички модули от сайта **ActiveState (**около **1500)** и около **1300** модула от **CPAN.***

*Диска струва **15** лв. За да видите пълното съдържание посетете този линк*
http://bgperl.tripod.com/Other/disc.html

След инсталирането**,** трябва да се направи асоциация с файловете на **Perl (**за да се отварят автоматично при натискането им**),** ако при инсталирането **Perl** не го е направил**.** Операциите са следните**:**
Start -> Settings -> Folder Options ->File Types ->New Type
Попълнете полетата**:**
Descriptions of type: PL file
Associated extension: .pl
Content Type (MIME) : text/plain

Actions: Open
Aplication used to perform action:
c:\perl\bin\perl.exe "%1" %* - или посочете там където сте инсталирали интерпретатора на **Perl**

Малко повече информация за Perl

Езика е създаден от Лари Уол**. Perl** е най**-**популярния метод за създаване на динамични **Web** страници**.** Простото обяснение е**,** че до последните няколко години на практика всеки съществуващ **Web** сървър работеше върху **UNIX** платформа**, a Perl** е сред най**-**полезните инструменти на **UNIX.** Това е една малка част от възможностите за използване на **Perl.** Силата му **е** основно върху вградените възможности за обработване

на текст, чрез създаване на шаблони за търсене и заместване на низове във файлове или цели групи от файлове. Друго основно предимство, е че програмите на **Perl** са със значително по-малък код, отколкото ако ги напишете на друг език. **Perl** е интерпретаторен език, а не компилаторен като **C** и **C++**. Основната разлика е, че при компилирането цялата програма се транслира на машинен език на компютъра, на който ще се изпълнява, от друга програма наречена компилатор. Компилираните файлове се изпълняват самостоятелно. От друга страна интерпретираните програми се транслират в процеса на изпълнението си от програма наречена интерпретатор. **Perl** програмите не се компилират, поради което ги наричаме скриптове. Когато говорим за интерпретатора на **Perl**, ще го записваме с малко **'p'** ето така **- perl**.

Как да използваме ръководството ...

Просто го прочетете от началото до края. Основно, кода следва обясненията. Преди да прочетете обясненията опитайте се да разберете какво прави кода и после проверете дали сте били прави. По този начин ще получите максимални знания, а и ще напрегнете сивите си клетки. Когато свършите изпратете ми мнението си за дизайна, съдържанието и ако откриете грешки в ръководство. Ще отговоря на всички писма! Моля не ми изпращайте въпроси за проблемите си с **Perl -** не съм техническа поддръжка **(**ако някой иска трябва да плаща**)**. Ако търсите решения на проблемите си пишете на **Usenet** или на **ActiveState mailing lists**. В ръководство кодът ще е е със син цвят, а командите от **DOS-prompt** със зелен цвят.

```
print "You must teach hard Perl";
```

```
C:>perl myfirst.pl
```

Ако прочетете цялото ръководство написано от мен **(**заедно с предстоящите допълнения**)** за езика **Perl**, Вие ще добиете добра представа за него. Следващите интернет адреси, ще Ви изведат на едно по-високо ниво в работата с **Perl**. Препоръчвам първо да прочетете информацията в това ръководство и след това да търсите отговори на възникналите въпроси.

Usenet информационни групи

Usenet е **Internet** услуга, която разпределя съобщения между сървъри. Всека стая си има специфична група. Нуждаете се от програма която чете новини и да ги запазвате на компютъра си. Най-лесно е да използвате **google.com**

Нюз група	Обяснение
comp.lang.perl.misc	Покрива общите въпроси свързани с **Perl**.
comp.lang.perl.announce	Покрива съобщения свързани с **Perl**.
comp.lang.perl.modules	Тази информационни група е много полезна, давайки отговори какви модули са налични, как да ги използваме и ако има някакви проблеми с използването им, винаги може да си зададете въпроса
comp.lang.perl.tk	**Perl/Tk** интеграцията и използването им. Това е форум където се водят дискусии за **Tk** и **Perl**. **Tk** е интерфейс разработен от **Sun**, основно за да се използва с **Tcl**, впоследствие займстван от **Perl** за разработването на **GUI** приложения.
comp.infosystems.www.announce	Не е свързан с **Perl**, но е много полезен за да научавате новите разработки в **web**.
comp.internet.net-happenings	Иформационна група даваща добра представа за

интернет разработките.

Най-използваната информационна група свързана с **Perl** е **comp.lang.perl.misc.** Когато имате въпроси или проблеми пишете на нея. Не задавайте въпросите си в повече от една нюз група, защото хората, които отговарят проверяват няколко нюз групи и е неприятно когато видят един и същи въпрос зададен няколко пъти. Преди да зададете въпросите си прочетете **Perl FAQ (**най-често задаваните въпроси**).** Запомнете, че хората които отговарят не са длъжни да го правя. Ако Вие се изразявате неясно и не описвате добре проблема си или се правите на многознайковци, в замяна няма да получите нищо.

Web Сайтове и Internet Relay Chat или IRC

Следващите сайтове са добро място за посещение за да си създадете собствена **Perl** и **CGI** скриптова библиотека. **Те** ще Ви дадат представа какво вече съществува, да намерите информация за това как другите разрешават проблемите си, начина на писане на скриптове, **Perl** идиоми и др. Много от скриптовете на посочените сайтове са **freeware** или **shareware.**

http://www.activestate.com/ - Сайтът е за **Perl, Rython** и **TCL** за **Win32.** Всеки който възнамерява да се занимава с **Perl** на **Windows** операционна система, трябва да посети този адрес.

ftp://convex.com/pub/perl/info/lwall-quotes - сайт за **Larry Wall.** Почитателите му са го направили за да покажат някои от коментарите написани за него.

http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Perl/

http://www-genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.cfml - Сайт за **CGI.pm.** `CGI.pm` е модул, който дава много възможности за писане на **HTML** и **CGI** скриптове.

http://www.stars.com - Сайтът съдържа много ръководства за **HTML, CGI, HTTP, Databases, CSS** и др.. Съдържа много богата колекция от линкове свързани с уеб програмирането.

http://www.teleport.com/~merlyn/ - Randal L. Schwartz е един от гурутата в **Perl** програмирането. Сайтът съдържа интересна и полезна информация.

http://www.engr.iupui.edu/~dbewley/perl/ - Perl информация. Съдържа ръководства, книги, скриптове.

http://www.worldwidemart.com/scripts/ Matt Wright's scripts е много добър сайт с добра организация, интуитивен, съдържащ добре документирани скриптове. Препоръчвам го ако искате да си изтеглите **CGI** скриптове, които да разгледате и прочетете упътванията за работата с тях.

http://www.hermetica.com/technologia/DBI/index.cfml - Сайта е за **Perl** и **DBI. DBI** е интерфейс за работа с бази данни като **– MySQL, mSQL, Oracle, Sybase, Informix** и др.

http://www.atmos.washington.edu/perl/perl.cfml - предлага ръководство по **Perl**

http://www0.cise.ufl.edu/perl/ - Страницата за **Perl** на университета във Флорида.

Internet Relay Chat усуга е много добро средство за намиране на информация. Ако имате късмет може да се свържете с хора, които имат големи познания по въпросите, които Ви интересуват и могат да Ви отговорят. Предимството на **IRC** е, че контакта става в реално време. Задавате въпрос и Ви се отговарят веднага. Има няколко мрежи поддържащи **IRC: EfNet, Undernet, DALnet** и **UniBG. Perl** гурутата са в **EfNet. #perl IRC** канала е добро място за задаване на въпроси. Ако имате по-прости въпроси, за вас е канала **#perl-basics. CGI** въпросите се задават в **#cgi.**

Да започнем изучаването на Perl

Вашият първи скрипт

Създайте нова директория, където ще съхранявате вашите **perl** скриптове например **c:\scripts**. Стартирайте текстовия редактор[1], с който ще пишете скриптовете се. Използвам **EditPlus**, който ще намерите на **www.editplus.com**. Напишете:

```
#!perl
print "I am a future Perl guru\n";
```

Запишете файла като **myfirst.pl**. Внимавайте с какво разширение ще запишете скрипта. Не затваряйте редактора си - дръжте го отворен за да правите промени по скрипта в движение. Стартирайте вашия **command prompt** (за тези, които съвсем нищо не знаят това е старият добър **DOS**. Стартирането става като натиснете **Start->Programs->MS-DOS Prompt** или **Start->Run->**напишете **command.com**). Напишете **doskey**, което стартира досовската програма **doskey**, служеща да запомня командите които пишете. Например написвате **perl myfirst.pl**, за да стартирате скрипта. Скрипта се изпълнява. След неговото изпълнение искате пак да го изпълните, не е нужно отново да пишете **perl myfirst.pl**, а просто със стрелките нагоре и надолу може да извиквате всичко, което сте писали в конзолата (промпта, **DOS** режима). Стигнете до директорията съдържаща скрипта. Напишете следващото за да стартирате скрипта:

```
C:>perl myfirst.pl
```

В тази кратка програма използвахме само една функция на **Perl**, която е **print**. Тук **print** получава низ (заграден с кавички текст), като свой аргумент. В случая казваме на **print**, че искаме да отпечатаме на екрана '**I am a future Perl guru**'.

Shebang

Почти всяка книга за **Perl** е написана за **UN*X**, което е проблем за тези които работят на **Win32**. Това води до скриптове като:

```
#!user/bin/perl
print "I'm a cool Perl hacker\n";
```

Функцията на **'shebang'** линията (първия ред на програмата започващ със символите **#!**) е да каже как да се изпълни файла. Под **UNIX**, това води до резултат. Под **Win32**, системата трябва вече да знае как да изпълни файла преди файла да бъде изпълнен, така че линията не е нужна. Но тя не се обезмисля напълно, тъй като могат да бъдат зададени първоначални опции за изпълнение на този ред (за пример **-w** е флаг, който указва на интерпретатора на **Perl** да изпълни скрипта в предупредителен режим. Вие може да напишете линията, така че скриптът да може да се изпълнява директно под **UNIX** без модификации. Под **win32** всеки **Perl** скрипт трябва да започва така:

```
#!perl
```

Коментари

Знака # в кода на скрипта указва, че следва коментар. Всичко след # се игнорира при изпълнението на скрипта. Вие не може да продължавате коментара от един ред на друг, затова ако искате да продължите коментара на следващия ред поставете # в началото му. Може да документирате програмата си, така че другите хора (или вие, след като месеци не сте я докосвали) да могат да разберат какво се постига с дадения код.

```
#! perl
# Този скрипт ще изведе кратко съобщение
print "I'm a cool Perl hacker\n";
```

Escaping

Символът ****, 'премахва' специалното значение на символите **$**,**@**,когато се използва заграден в кавички и др . Символът **** има и по-широко използване - поставен пред който и да е специален знак в **Perl** премахва значението му и го третира като обикновен символ. Също така, прави някои обикновенни символи специални. Прибавянето на **** ,прави от простия символ **'n'** указание за нов ред. Той може да премахва и собственото си значение, например **** . Така че ако искате да изпечатате единична **** пробвайте:

```
print "The main path is c:\\perl\\";
```

'\' се използва и за създаване на референции. **Perl** използва **DOS** символи които не са еднозначни по своето значение в различни случай. В някои случай един символ има две или повече значения в зависимост от контекста в които се употребяват. Коректорния знак **^** има различно значение в **[^abc]** и **[a^bc]** . Така и **'\'** има различно интерпретиране от **perl** в зависимост от средата в която е поставен.

Типове данни

Променливи

Всеки програмен език разполага с типове данни, върху които извършва операции. В **Perl** те могат да бъдат от една страна число или низ, константа или променлива.

Число: едно- или многоцифрено бройно число.

Низ: сбор от символи (числа, букви, всъщност всеки **ASCII** код от **0** до **255**). Единичните или двойните кавички оказват начало и край на низа. Двойните кавички имат по специално предназначение. Чрез тях се включват някой специални символи. Низът се заграден в единични или двойни кавички според ефекта, който търсим.

Константа: това е число или низ, което предварително е известно и зададено в скрипта, не се изменя с изпълнението на скрипта.

Променлива: това е място в паметта, където може временно да се съхранява информация, по време на изпълнение на скрипта тази информация може да се променя. В **Perl** всяка променлива може да бъде с "произволна" дължина. т.е променливата освен, че може да променя съдържанието си, може да променя и размера си - всичко това става автоматично. За разлика от други програмни езици, променливите не е необходимо да бъдат декларирани, те могат да се обявяват направо като се използват.

Променливите се означават чрез поставянето на специален символ, последван от име на променливата. Специалните символи могат да бъдат: **$**, **@**, **%**, ***** - определящи типът на променливата. Името може да бъде произволно, с много малко ограничения, едно от които е, че ако името започва с число, то в него не могад да се съдържат букви, само числа са разрешени.

Скалари

А сега да преминем към нещо по интересно от принтването на екрана. Скаларът е единична променлива. В **$var=10**, скалар е **$var**, а равното задава на променливата с името **$var** стойността **10**. Означаването на скалари става посредством знакът: **$**, който много прилича на **"S"(Scalar)**. По-късно,ние ще се запознаем с масивите и хешовете, където **@var** се обръща към повече от една стойност.

```
$string="perl";
$num1=20;
$num2=10.75;
print "The string is $string, number 1 is $num1 and number 2 is $num2\n";
```

$ % @ са основни понятия в **Perl**. Ако Вие сте запознат с други програмни езици ще се изненадате от кода **$var=10**. С повечето езици, ако искате да определите стойността **10** на променливата с име **var** Вие трябва предварително да определите какъв тип ще е **var**, **integer**, **boolean** или др.. Не е така в **Perl**, това е отличителна черта. Всички променливи имат за представка символ като **$ @ %** . Които ги определят какъв тип променливи са. За

разлика от други езици, променливата може да се зададе и в самата операция, в която се използва.

Интерполация на променливи

Ние все още не сме приключили с кода. Забележете начина на използване на променливите в стринга. Когато искаме променливата да се интерполира - да покажем стойността, използваме **" "**, а когато не искаме интерполация - да покажем името на променливата - използваме **' '**.

```
$string="perl";
$num=20;
print "Doubles: The string is $string and the number is $num\n";
print 'Singles: The string is $string and the number is $num\n';
```

Свързване

Когато имате данни, които трябва да се свържат, например в скалар използвайте **'.' .** Например:

```
$x="Hello";
$y=" World";
$z="\n";

print "$x\n";
$prt=$x.$y.$z;
print $prt;
$x.=$y." again ".$z;
print $x;
```

По-долу е показан пример, с който да разберете именните пространства и как променливите се държат в тях. С **package** се декларира ново именно пространство. Пълното име на една променлива **(**както го вижда **perl) е $**пакет**::**име. Пакета може да е главния **(main)** или да е някой, който е деклариран или зареден преди това. В примера е пакета **foo.** Ако искаме да извикаме променлива от един пакет в друг трябва да укажем пълното име на променливата. Когато започваме работата си ние работим в **main**, за да променим именното пространство **(**символната таблица**)** ние трябва да укажем това чрез **package.**

```
$name="mitko\n";
print "main $name";
print "main $main::name";

package foo;

$name="Dimitar\n";
print "foo $name";
print "foo $foo::name";
print "foo $main::name";

package main;

print "main $name";
print "main $main::name";
print "main $foo::name";
```

Сега нека пробваме друго:

```
$car ="Porsche 911";
$aircraft="G-BBNX";

print "Car :$car    - Aircraft:$aircraft:\n";
print "Aircraft exists!\n" if $aircraft;
print "Car exists!\n" if $car;

$car="";

print "Car :$car    - Aircraft:$aircraft:\n";
print "Aircraft exists!\n" if $aircraft;
print "Car exists!\n" if $car;
```

Изглежда че сме изтрили променливата **$car**. Но не точно. Ние сме изтрили стойността, която тя има, но не и самата променлива. В момента тя има стойност **null** и затова теста с **if** пропада. Ако нещо дава като отговор грешка, то това не означава, че не съществува. Перуката е фалшива коса, но перуката съществува. Вашата променлива е все още там. **Perl** има функция, която да тества дали нещо съществува. Съществувам, в превод на **Perl** означава дефиниран **(defined)**:

```
print "Car is defined !\n" if defined $car;
```

Този код ще върне истина. Примерите дотук поставят въпроса, всъщност как да премахнем невъзвратимо една променлива. Много просто, ето как:

```
$car ="";
$aircraft="G-BBNX";

print "Car :$car: Aircraft:$aircraft:\n";
print "Aircraft exists!\n" if $aircraft;
print "Car exists!\n" if defined $car;

undef $car;

print "Car :$car: Aircraft:$aircraft:\n";
print "Aircraft exists!\n" if $aircraft;
print "Car is not defined and not exists!\n" if not defined $car;
```

Променливата **$car** е унищожена, тя липсва.

Масиви

В **Perl** има два типа масиви, асоциативни масиви **(**хешове**)** и масиви. И двата масива са списъци. Списъкът е множество от стойности. Масивът е подреден списък от скалари, достъпни чрез номера им. Не е задължително те да бъдат от един и същи тип. Може да имате три скалара, два масива и три хеша в един масив. Дефиницията на масив в другите езици е: съвкупност от еднотипни елементи. В **Perl** обаче всеки елемент може да е различен и с произволна дължина, следователно масивът е съставен от разнородни по състав елементи. За това често масивът е наричан списък. Масивите се означават с **@ -** която наподобява **"a"(**A**rray-**масив**).**

Основи на работата с масиви

Може да се обръщаме към целия списък или към отделни елементи от него. Скрипта по долу задава масива наречен **@names .** Зададени са **5** стойности в масива.

```
@names=("Nie","Vsichki","zaedno","chte","pobedim");

print "The elements of \@names are @names\n";
print "The first element is $names[0] \n";
print "The third element is $names[2] \n";
print 'There are ',scalar(@names)," elements in the array\n";
```

Първо, забележете как се декларира масива **- @names .** Всяка стойност е заградена в кавички **"",**а запетайката е подразбиращия се разделител за стойности в масива**.** След това вижте как се извежда**.** Първо се обръща към себе си като цяло в контекста на списък**.** Това означава **,** че се обръща към повече от една стойност**.** Вижте следващия код**:**

```
@names=("Muriel","Gavin","Susanne","Sarah","Anna","Paul","Trish","Simon");

print @names, "\n";
print "@names";
```

Когато списъка е между **""** стойностите в него се извеждат с интервал между тях**.** Ако искаме да се обърнем към повече от една стойност от масива използваме **@ .** Когато се обръщате към повече от една стойност от масива**,** но не към целия масив**,** това се нарича дял**,** резен **(slice) .** Аналогията с кекс е подходяща**.** Парче от кекс е подходящо сравнение за дял**.**

```
print @names[1..3];
```

Има и по лесен начин за задаване на масив**:**

```
@names=qw/one two tree four five/ # задава масив от пет елемента
```

При този начин за задаване на масив**,** за разделител на елементите се използва интервал **(**редовна грешка е да се използва **‘,’ ,** което предизвиква грешка при компилиране**).**

Елементи на масив

Масивите не са много полезни**,** ако не можем да се обръщаме към стойностите в него**.** Първо ако извикваме единичен елемент от масив не трябва да използваме префикса **@,** който се използва за обръщение към няколко стойности**.** Ако ни трябва единична стойност**,** то това ще е скалар**,** а префикса за скалар е **$.** Второ ние трябва да укажем кой точно елемент искаме**.** Това е лесно **- $array[0]** за първия елемент**, $array[1]** за втория и т**.**н**..** Индексът на списъците започва от **0,** докато не направите нещо което да промени това **(**нещо**,** което е твърде нежелателно и противопоказно за добро и пълноценно програмиране**).** В примера по горе използвахме масива в скаларен контекст за да видим колко елемента има**,** чрез **scalar(@names).** Ето как да се обръщаме към елементите на масив

```
$myvar="scalar variable";
@myvar=("one","element","of","an","array","called","myvar");

print $myvar;     # обръща се към скалар с името myvar
print $myvar[1];  # обръща се към втория елемент на масива myvar
print @myvar;     # обръща се към всички елементи на масива myvar
```

Двете променливи **$myvar** и **@myvar** нямат нищо общо помежду си**.** Ако направим аналогия с животните**,** това е като да си имаш куче с име **'Myvar'** и златна рибка **'Myvar'.** Вие никога няма да ги сбъркате**,** нали**?** Номерът на елемента от масив**,** който извикваме може да бъде променлива**.**

```
print "Enter a number :";
chomp ($x=<STDIN>);

@names=("one","element","of","an","array");
print "You requested element $x who is $names[$x]\n";
print "The index number of the last element is $#names \n";
```

Забележете последния ред на примера. Връща индекса на последния елемент на масива. Същото може да го направите и с **$last=scalar(@names)-1;** но това не е толкова ефективно. Има по-лесен начин да получите последния елемент на масива, както ще видите:

```
print "Enter the number of the element you wish to view :";
chomp ($x=<STDIN>);

@names=("one","element","of","an","array");

print "The first two elements are @names[0,1]\n";
print "The first three elements are @names[0..2]\n";
print "The first, second, third and fifth elements are @names[0..2,4]\n";

print "a) The last element is $names[$#names]\n"; # първи начин
print "b) The last element is @names[-1]\n";      # втори начин
```

На **5** ред има две стойности разделени от многоточие. Това дава всички стойности от първата до последната между двете числа. **[0..3]** означава **0, 1, 2, 3** .Когато за индекс се използва отрицателно число, стойностите на масива се взимат отзад напред, **-1** е за последната стойност, **-2** за предпоследната и т.н.

Промяна на елементите на масив

И така имаме масива **@names.** Искаме да го променим. Пробвайте следващия код:

```
print "Enter a name :";
$x=<STDIN>;


@names=("one","element","of","an","array");

print "@names\n";
push (@names, $x);
print "@names\n";
```

Push функцията прибавя нова стойност към края на масива. Разбира се може да не е само една стойност:

```
print "Enter a name :";
chop ($x=<STDIN>);

@names=("one","element","of","an","array");
@jus=("cola","coca","cola","pepsi");

print "@names\n";

push (@names, $x, 10, @jus[1..3]);

print "@names\n";
```

Ето още функции за работа с масив:

```
@names=("one","element","of","an","array");
@jus=("cola","coca","cola","pepsi");

print "Names : @names\n";
print "jus: @jus\n";

$last=pop(@names);
unshift (@jus, $last);

print "Names : @names\n";
print "jus: @jus\n";
```

pop функцията премахва последния елемент на масив и го връща, което означава че може да извършите някакви действия с върнатата стойност. Функцията **unshift** прибавя стойност в началото на масив. В таблицата по долу са изборени функциите, които се използват за работа с масив.

push	Прибавя стойност към края на масив.
pop	Премахва и връща последната стойност на масив.
shift	Премахва и връща първата стойност на масив.
unshift	Прибавя стойност към началото на масива.

Сега за достъпа до другите елементи на масива. Ще Ви представя функцията **splice**.

```
@names=(“one”,”element”,”of”,”an”,”array”);

print "Names : @names\n";
print "The elements of massive are: @middle\n";

@middle=splice (@names, 1, 2);

print "Names : @names\n";
print "The elements of massive are: @middle\n";
```

Първия аргумент на функцията **splice** е масив. Втората стойност е индекс (число) показващо, коя стойност от списъка (масива) да се използва за начало. В този случай тя е **1** (ще се започне от втория елемент). После идва елемента, който показва колко елементи да се премахнат от масива и да се прибавят към втория.
Ако използвате резултата от **splice** като скалар:

```
@names=(“one”,”element”,”of”,”an”,”array”);

print "Names : @names\n";
print "The Splice Girls are: $middle\n";

$middle=splice (@names, 1, 2);

print "Names : @names\n";
print "The Splice Girls are: $middle\n";
```

скалара взема като стойност последната му подадена стойност.

Триене на елемент на масив

Искаме да изтрием **Hamburg** от масива. Как да го направим? Може би ето така:

```
@cities=("Brussels","Hamburg","London","Breda");
```

```
print "Cities: ",scalar(@cities), ": @cities\n";

$cities[1]="";

print "Cities: ",scalar(@cities), ": @cities\n";
```

Разбира се **Hamburg** е премахнат, но забележете, броят на елементите остана един и същ. Все още има **4** елемента в масива. Резултата от горния код е, че **$cities[1]**, съществува но няма стойност. И така ние трябва да използваме **splice** функцията за да премахнем елемента изцяло.

```
splice (@cities, 1, 1);
```

Хешове

Хешовете са множество от скалари достъпни, чрез ключа свързан със всеки от тях. При хеша няма начало и край, те не са подредени. Всяка двойка в хеша се състой от ключ и стойност. Стойността е достъпна чрез ключа си, а не като при масивите по ред на номера. Ето как се задава хеш:

```
%hash1=(“key”,value”,“number”,123,“az”, “Vie ste ot drugata strana”);
%hash2=(
“key”=>”value”,
“number”=>123,
“az”=> “Vie ste ot drugata strana”,
)
```

Цялото съдържание на хешът не може да се извежда, както при масива. За да изведем дадена стойност от хеш,трябва да посочим ключ:

```
print $hash{“number”};
```

Което ще изведе **123** като резултат. И тук всяка отделна стойност започва с **$**, като ключа се указва в **{}**, за разлика от масива където се използват квадратни скоби **[]**.За да изведем всички ключове, а после всички стойност на един хеш, се използват двете функции **keys**, **values**.

```
print keys %hash,”\n”;
print values %hash,”\n”;
```

За да изведем ключ**/**стойност наведнъж се използва функцията **each**:

```
foreach (($key,$val)=each %hash) {
       print “$key - $val\n”;
}
```

Подпрограми

Общо представяне

Нека погледнем по друг начин на кода, който вече използвахме. Забелязваме, че много пъти ни се налага да пишем едно и също. Повтарящият се код може да се сложи в блок (блок е всичко което е записано между **{})** и да се изпълнява многократно при повикване, ако има име. Подпрограмата е блок с име. Чрез извикване на името на подпрограмата, ние ще изпълним блока и. Чрез подпрограмите се намалява писането на кода, увеличава се прегледността на програмата, намалява големината на файла и т.н.. Ето и пример:

```
$num=10;
&print_results;

$num++;
&print_results;

$num*=3;
&print_results;

$num/=3;
&print_results;

sub print_results {
print "\$num is $num\n";
}
```

Подпрограмите могат да се слагат навсякъде в скрипта, в началото, края, по средата ... няма значение. Добра практика е всички подпрограми да се слагат в края на файла. По-прегледно и удобно е. Подпрограмите са код, който искате да се изпълни повече от веднъж в скрипта. Задават се чрез **sub** и след него името на подпрограмата. След което лява голяма скоба **{** , кода на подпрограмата и затваряща дясна голяма скоба **}**. Подпрограмите обикновенно се извикват чрез **'&'** и името на подпрограмата, като **&print_results**; . Може да се пропусне импеданса, но тогава възниква възможност за грешки със файловите манипулатори (за това по-късно), затова по-добре не го правете. Ето различни начини за извикване на подпрограма:

```
&print_results();
&print_results;
print_results();
print_results;
```

Всеки от тези начини, за извикване на подпрограма има различно значение (дали ще се проверяват прототипите или не – за това по-долу).

прототипи

```
#!perl

use warnings;
use strict;

my @s=qw/Just another Perl hacker/;
my $f="Yes! That's right";

sub prototipes (\@$) {
my $a=shift;
my $b=shift;
print "\@a - @$a\n";
print "\$b - $b\n";
}

prototipes(@s,$f);
```

Прототипи се използват за да се укажат броя и вида на аргументите който трябва да се предадат на подпрограмата. Без прототипи в стандартно извикване на по горната подпрограма нещата ще са коренно различни. Няма да може да предадете масива отделно от другите данни, трябва да използвате референция. Ако искаме да извикаме подпрограма, която използва прототипи, но ние искаме да ги заобиколим, то подпрограмата трябва да се извикаа като пред нея поставим **&**. Когато искаме да декларираме като прототип масив или хеш пред тях поставяме ****, например **podsub1 (\@\%).** Задължителните от незадължителните аргументи се разделят с точка и запетая **;** , например **podsub1 (\@\%*;$$).**Тук извиквамме подпрограма изискваща масив, хеш, манипулатор и два скалара който са незадължителни.

Ако в подпрограмата няма зададено какво да се върне, то тя ще върне последното пресметнато. Например може да създадете функция която да проверява дадено условие и да връща истина или неистина **(** в контекста на **perl).**

for (1..20) { print “ Числото е това което ми трябва**” if rez($_);}**

sub rez { $_>10 && $_<=15;}

Оператори

Оператора изпълнява функция или операция над някакви данни. Например оператора **+** събира две числа. Ако искате да прибавите **1** към променливата, Вие може да направите следното; **$num=$num+1** . Има и кратък начин да направите това, който е **$num++**. Това е автоматично прибавяне на единица към стойността на скалара. Познаите какво е това: **$num--** .

```
$num=22;
print "\$num is $num\n";
$num++;
print "\$num is $num\n";
$num--;
print "\$num is $num\n";
$num+=33;
print "\$num is $num\n";
```

В предпосредният ред операцията е аналогична на **$num=$num+33.**

Променливи, нарастването и принтирането им

```
$num=20;
$num2=3;
print "The number is $num\n";
$num*=$num2;
print "The number is $num\n";
```

=** означава 'умножи **$num** по **$num2** или, **$num=$num*$num2** . Разбира се **Perl** поддържа обикновенните оператори **+**, **-**, **, **/**, ******, **%** . Последните две са степенуване и делене, връщащо остатъка. **print** е функция за работа със списъци. Това означава, че приема списък от стойности, разделени със запетайки, както във следващия пример:

```
print "a doublequoted string ", $var, 'that was a variable called var', $num,"
and a newline \n";
```

Разбира се, Вие може да сложите всичко това между **""**

```
print "a doublequoted string $var that was a variable called var $num and a
newline \n";
```

за да постигнете същия резултат. Предимството от използването на **print** в списъчен контекст, е че изразите се изчисляват преди да бъдат принтирани. Например пробвайте това:

```
$var="Perl";
$num=10;
print "Two \$nums are $num * 2 and adding one to \$var makes $var++\n";
print "Two \$nums are ", $num * 2," and adding one to \$var makes ",
$var++,"\n";
```

Ако искаме към **$var** да се прибави единица преди да се принтира то тогава трябва да използваме следния записване **++$var.**

Инструкции

Сравнения – if , elsif, else

Работата с функцията **if** е проста. Ако деня е **Monday**, тогава ще отивам на работа. Прост тест, с два изхода:

```
if ($day eq "Monday") {
       &work;
}
```

Вие вече знаете, че **&work** извиква подпрограма със същото име. Допускаме, че променливата **$day** е зададена по рано в скрипта. Ако **$day** няма стойност **' Monday'**, **&work** няма да се изпълни. Пробвайте това:

```
$day="Monday";
if ($day eq "Monday") {
       print "work, work and fucking work..\n";
}
```

Забележете синтаксиса. **if** се нуждае от нещо, което да се тества и да върне като резултат истина **(1)** или лъжа **(0)**. Този израз трябва да е в **()** - скоби. следвани от **{}** - където трябва да се постави блока за изпълнение, ако израза в **()** върне като отговор истина. Пробвайте този код:

```
$age=25;
$max=30;

if ($age > $max) {
      print "Too old !\n";
} else {
     print "Young person !\n";
}
```

Лесно е да видите какво прави той. Ако израза е грешен, тогава все едно какво има в **else** - блока се изпълнява. Просто е. Но ако искате още един тест?

```
$age=25;
$max=30;
$min=18;

if ($age > $max) {
     print "Too old !\n";
}
elsif ($age < $min) {
     print "Too young !\n";
}
else { print "Just right !\n"; }
```

Ако първия тест пропадне, вторият се изчислява. Това се извършва докато има **elsif**, или **else** е достигната. Има голяма разлика между горния код и следващия по-долу:

```
if ($age > $max) {
     print "Too old !\n";
}
if ($age < $min) {
     print "Too young !\n";
}
```

Ако го стартирате ще Ви върне същия резултат - в този случай. Както и да е, но това е Лоша Практика в Програмирането. В този случай ние тестваме число, но представете си, че тестваме стринг за да видим дали съдържат **R** или **S.** Възможно е стринга да съдържа и двете **R** и **S.** Така ще се преминат и двата **'if'** теста. С използването на **elsif** ще избегнете този недостатък. При първото срещане на истина в **elsif** изявление то се изпълнява, като следващите не се изпълняват.

Истина според Perl

Има много функции в **Perl** които тестват за истина. Някои от тях са **if, while, unless.** Така че, е важно да знаете какво е истина, като дефиниция в **Perl.** Има три основни правила:

Всеки стринг е истина с изключение на **""** и **"0".**
Всяко число е истина с изключение на **0.** Истина са и отрицателните числа.
Всяка недифинирана **(not defined)** стойност е лъжа. Недефинираната стойност е тази, която няма стойност.
Код, който илюстрира по-горните твърдения:

```
&isit; # $test1 в този момент е недефинирана - тя не съществува

$test1="hello"; # стринг който е различен от "" и "0"
&isit;

$test1=0.0; # $test1 е число, но реално е 0
&isit;

$test1="0.0"; # $test1 е стринг, но НЕ е '' или 0 !
&isit;

sub isit {
if ($test1) { # тества $test1 за истина или не
       print "$test1 is true\n";
}
else { # else ако не е истина
      print "$test1 is false\n";
}
}
```

Първия тест пропада, защото **$test1** е недефинирана. Това означава, че не е била създадена за да приеме някаква стойност. Последните два теста са интересни. Разбира се **0.0** е едно и също с **0** в числен контекст. Но **0** в контекста на стринг е истина. Дотук тествахме единични променливи. Сега ще тестваме резултатите от изрази.

```
$x=5;
$y=5;

if ($x - $y) {
       print 'if - $x - $y is ',$x-$y," which is true\n";
}
else { print 'else - $x - $y is ',$x-$y," which is false\n";}
```

Теста пропада защото **5-5** е **0,** което е грешка. **print** може да изглежда малко странно. Имаме списък, в който първата стойност е в **''.** Това е защото не искаме да интерполираме стойностите. Следващата стойност е израза който трябва да се пресметне,

затова е извън всякакви кавички. И накрая стойността е в **""**, защото искаме след изпълнение на кода интервала да отиде на нов ред - **\n**, а без **""** това няма да стане

Друг тест е да сравним две стойности за истина.

```
$lucky=15;
$drawnum=15;

if ($lucky == $drawnum) {
        print "Congratulations!\n";
}
else {
        print "Guess who hasn't won!\n";
}
```

За сравнение в горния пример е оператора за равенство **==** .

Сравненията и Perl

Сега трябва да внимавате във следващите обяснения, защото после ще се връщате тук отново. Символът **=** е оператор за присвояване на стойност, не е оператор за сравнение. Следователно:
if ($x = 10) е винаги истина, защото на **$x** успешно се присвоява стойността **10**.
if ($x == 10) сравнява две стойности, които може да не са равни.
И така ние сравнявахме до сега числа, но ние можем и сигурно ще се налага да сравняваме променливи.

```
$name = 'Mark';
$goodguy = 'Tony';
if ($name == $goodguy) {
        print "Hello, Sir.\n";
}
else {
        print "Begone, evil peon!\n";
}
```

Нещо грешно има в кода. Нещо изглежда не е както трябва. Очевидно **Mark** е различно от **Tony**, тогава защо **perl** ги намира за равни? Те са равни **--** числено. Ние трябва да ги тестваме като стрингове, а не като числа. За да направим това, просто заместете **==** със **eq** и горния код ще работи както трябва. Има два типа оператори за сравнение - за числа и за стрингове. Видяхте вече два такива, **==** и **eq**. Пробвайте този код:

```
$foo=291;
$bar=30;
if ($foo < $bar) {
        print "$foo is less than $bar (numeric)\n";
}
if ($foo lt $bar) {
        print "$foo is less than $bar (string)\n";
}
```

Операторът **lt** сравнява в контекста на стринг, и разбира се **<** сравнява в контекста на числа.
Азбучно, това е в контекста на стринг, **291** е след **30**. Това е по кодовата таблица **ASCII**.
Мотото на **Perl** е "Има повече от един начин да се направи" или **TIMTOWTDI**. Произнася се **'Tim-Toady'**. Това ръководство не се опитва да Ви покаже всички начини за решаването на даден проблем. При писане на вашите програми ще ги откривате.
В следващата таблица са показани операторите за сравнение в **Perl**:

	Числен контекст	В контекста на стринг
Равно	**==**	**eq**
Не равно	**!=**	**ne**
По-голямо от	**>**	**gt**
По малко от	**<**	**lt**
По-голямо или равно	**>=**	**ge**
По-малко или равно	**<=**	**le**

За циклите

За да изпълним многократно един цикъл върху даден масив или друго, то трябва да използваме операторите за цикли **for, foreach, while, until.** Например как да изведем всички елементи на масив един след друг**?** Решението е в следващия пример**:**

```
@names=qw/edno dve tri chetiri pet chest/;

for ($x=0; $x <= $#names; $x++) {
          print "$names[$x]\n";
}
```

което задава на **$x** стойност **0,** прави един цикъл**,** прибавя единица към **$x ,** проверява дали е по-малка или равна на **$#names ,** и т.н**..** Това е пример за цикъл**.** За да бъде по-детайлно**,** цикъла има три части**:**

- Инициализация
- Проверка на условието
- Модификация

В този случай**,** променливата **$x** е инициализирана на **0 ($x=0).** След това веднага е проверена дали е по-малка или равна на **$#names ($x <= $#names) .** Ако е истина**,** блокът се изпълнява веднъж**.** Ако условието не е изпълнено и блока не се изпълнява**.** Един път след като блока се изпълни**,** променливата се модифицира **($x++).** След това теста за изпълнение на условието се изпълнява и т.н**..**

В **Perl** няма цикъла **SWITCH (C,C++,PHP),** но може да се постигне по няколко начина**.** Един от най-удачните е**:**

```
for ($animal) {
       /camel/     and do {Humps(2); last};
       /dromedary/ and do {Humps(1); last};
}
```

Има и друг начин за написването на по-горния код**:**

```
for $x (0 .. $#names) {
        print "$names[$x]\n";
}
```

който използва оператора за поредица **..** (две точки една до друга**).** Това просто задава на **$x** стойност **0,** после увеличава **$x** с **1** докато стане равно на **$#names .** През цялото това време изпълнява блока по веднъж на всяко увеличаване с единица**.**

foreach

За по-добър код използваите **foreach .**

```
foreach $person (@names) {
        print "$person";
}
```

Цикъла минава през всеки елемент **('**многократно повторение**',** друго техническо описание на процеса**)** на масива **@names ,** и всеки елемент се връща като стойност на променливата **$person .** След това може да извършвате всякакви действия с променливата**.** Вие може да използвате и**:**

```
for $person (@names) {
        print "$person";
}
```

Няма разлика като цяло**,** но кода изглежда малко по-неясен**.**

Невероятното $_

За да намалим кода **,** ще Ви преставя **$_ ,** които е променлива по подразбиране или шаблонна променлива**.**

```
foreach (@names) {
        print "$_";
}
```

Ако Вие не зададете променлива в която да се присвояват като стойности**,** елементите на масива **,** те се присвояват в **$_. $_** се използва по подразбиране за тази и много други операции в **Perl.** Още един пример с функцията **print:**

```
foreach (@names) {
         print;
}
```

Ние не определяме кой елемент да се принтира**,** затова **$_** се изпозва по подразбиране**.** По дефиниция ако на **print** не е зададена променлива**,** тя използва стойността**,** която се намира в специалната променлива **$_.**

Преждевременен край на повторения

Процесът на повторенията завършва при постигането на някакво условие зададено първоначално или въобще не завършва**.** Следващия скрипт е пример за непрекъсваем процес**,** защото **1** е истина и условието е винаги истана**.**

```
while (1) {
      $x++;
      print "$x\n”
# чрез натискане на CTRL-C може да прекъснете изпълнението на perl скрипт
}
```

Друг начин да излезете от процеса е**,** ако при изреждането на елементите на масива намерим този**,** който ни трябва и продължаването на процеса се обезмисля**.**

```
@names=('Mrs Smith','Mr Jones','Ms Samuel','Dr Jansen','Sir Philip');

foreach $person (@names) {
        print "$person\n";
```

```
        last if $person=~/Dr /;
}
```

Оператора **last** изпълнява функцията прекъсване на процеса и излизане от цикъла. Не се тревожете за **/Dr /** това е регулярен израз, който ще бъде обяснен по-късно. Всичко, което трябва да знаете за сега е, че връща истина ако намери съвпадение между това в скобите и подадената променлива. В случая когато открие съвпадение подава истина и функцията **last** се изпълнява. Контрол върху функцията **last** се извършва чрез етикети.
Засега е просто, но почакайте**!** Ние се нуждаем от лекар, който го има и в списъка с имена, а не просто само лекар. Следващия пример е за това:

```
@medics =('Dr Black','Dr Waymour','Dr Jansen','Dr Pettle');

foreach $person (@names) {
        print "$person\n";
        if ($person=~/Dr /) {
               foreach $doc (@medics) {
                      print "\t$doc\n";
                      last if $doc eq $person;
               }
        }
}
```

Първо се взима първия елемент на масива **@names**, изписва се на екрана, и след това се проверява за съвпадение с **Dr**, ако се открие съвпадение се изпълнява следващия блок. В него се изреждат всички елементи на масива **@medics**, и ако се открие, че съдържанието на променливата **$doc** съвпада със съдържанието на променливата **$person** се прекъсва изпълнението на този блок и се преминава към външния цикъл с нова променлива **$person.**

Но има малък проблем, ние искаме при намиране на съвпадение да прекъснем изцяло цикъла, а не само изпълнението на вътрешния цикъл. Трябва да укажем на функцията **last**, кое точно искаме да прекъснем. Това става чрез етикети, като във следващия пример:

```
LABEL: foreach $person (@names) {
        print "$person\n";
        if ($person=~/Dr /) {
               foreach $doc (@medics) {
                      print "\t$doc\n";
                      last LABEL if $doc eq $person;
               }
        }
}
```

Има само две промени тук. Дефинирали сме етикет с името **LBL.** Когато укажем на **last** даден етикет, той прекъсва функцията, на която е зададен първоначално този етикет.

Има още две функции за работа с цикли **redo** и **goto**. **Redo** връща на току-що изпълнилото се **$_**, като условието в проверката не се изпълнява. **Goto** изпраща към друг цикъл, подпрограма или израз за изчисляване. Не се препоръчва използването му.

Въвеждане и работа с данни

STDIN, манипулатори

Понякога Вие трябва да контактувате с потребителя за да получите информация и да вършите действие с нея.

```
print "Your age: ";
$name=<STDIN>;
print "Your age is $name !\n";
```

Има нови елементи, които трябва да научите тук. Първо **<STDIN> . STDIN** е стандартният вход на една програма **(**първоначално това е клавиатурата, докато не промените това**).** Манипулаторите служат за да се взаймодейства с файлове, сокети, входни данни и др. В този случай **STDIN** чете от клавиатурата **(**входните данни въведени от потребителя**).** Този тип скоби **- <>** чете данни от подаденият и манипулатор **–** той може да е файл или др.**.** И така ние прочитаме въведените данни от **STDIN.** Зададената стойност се установява на провенливата **$name** и принтира. Някаква идея защо има още един ред. Като натиснете **Enter**, Вие включвате нов ред във вашите данни. Лесния начин да премахнете новия ред е чрез **chop:**

chop

chop VARIABLE
chop LIST
chop

```
print "Please tell me your name: ";
$name=<STDIN>;
chop $name
print "Thanks for making me happy, $name !\n"
```

Този код пропада поради синтактична грешка. Можете ли да познаете защо**?** Погледнете изведената информация, вижте номера на реда, където е открита грешката и вижте къде в синтаксиса сте сбъркали. Отговорът е в липсващите точка и запетая **(;)** в края на последните два реда. Ако Вие добавите **;** в края на трети ред, но не и на последния ред, тогава програмата ще работи както трябва. Това е така защото **Perl** не се нуждае от **;** на последния ред от блок. Препоръчително е винаги да се слага, най-малкото, защото е едно натискане на клавиш, а и след дадения код винаги може да добавите нещо, а ако сте пропуснали **;** да търсите къде има синтактична грешка. Функцията **chop** премахва последния символ от всичко каквото и е дадено. В този случай премахва символа за нов ред. Кодът може да се съкрати, ето как**:**

```
print "Please tell me your name: ";
chop ($name=<STDIN>);
print "Thanks for making me happy, $name !";
```

Скобите **( )** принуждават **chop** да действа на резултата от това което е вътре в скобите. И така **$name=<STDIN>** се изпълнява първо, после на резултата от това се премахва последния символ. Пробвайте без него. Може да прочетете от **STDIN** колкото пъти поискате. Вижте този код.

```
print "Please tell me your nation: ";
chop ($nation =<STDIN>);

print "Please tell me your name: ";
chop ($name=<STDIN>);

if ($nation eq "BG" or $nation eq "Bulgaria") {
       print "Zdrasti $name, da izpiem po bira v Punka,a?\n";

} elsif ($nation eq "Ru" or $nation eq "Russian") {
       print "Zdrastvui bratushka $name!\n";

} else {
       print "Hoi men $name\n";
}
```

Премахва последния символ от стринг и го връща. Може да се използва и за премахване на нов ред, но е много по-ефективен от **s/\n//**, защото не сканира стринга. Ако променливата не е зададена, **chop** действа на **$_**.

```
$text="Long text";
while ($text) { # докато има текст ще се изпечатва буква по буква
chop; # премахва последния символ
print $_,"\n" # изпечатва го;
}
```

Премахването е опасно, в период когато най-важното нещо е сигурността при програмирането.

Безопасно премахване чрез chomp

chomp VARIABLE
chomp LIST
chomp

Ние искаме да премахваме само символа за нов ред, а не безогледно, който и да е последен символ. Това може да стане с **chomp** - който премахва последния символ само, ако той е за нов ред.

```
chomp ($name=<STDIN>);
```

И тук **Perl** гурутата извикват "Открих грешка!". Е добре, **chomp** не винаги премахва последния символ, ако той е за нов ред, но ако не го премахне, Вие имате специалната променлива **- $/** .Като зададете нов символ на тази променлива, тази стойност става символа за нов ред, на която стойността по подразбиране е **\n**.

Тази безопасна форма на **chop** премахва само стойността зададена от специалната променлива **$/** (още позната като **$INPUT_RECORD_SEPARATOR** в модула **English**), съдържаща символа, който се приема за знак означаващ последен ред, който по подразбиране е **'\n'**. Връща броя на всички премахнати **$/** . Използва се за премахване на новия ред от края на въведена информация. Ако променливата не е зададена, **chomp** действа на **$_**. Пример:

```
while (<>) {
chomp; # премахва \n от всеки ред
@array = split(/:/);
# ...
}
```

Може да премахвате **$/** от всяка въведена стойност**:**

```
chomp($cwd = `pwd`);
chomp($answer = );
```

Ако прилагате **chomp** върху списък**,** ще бъде върнат като отговор общия брой на премахнатите **$/ .**

Работа с файлове и директории

Отваряне на файл, четене,затваряне и запис

```
$stuff="c:/scripts/stuff.txt";

open (STUFF, $stuff) or die $!;

while (<STUFF>) {
       print "Line number $. is : $_";
}
```

Задаваме във един скалар пътя до даден файл. Функцията **open** има две стойности, който трябва да се зададат. Първата е името, което ще се даде на отворения файл или тъй нареченият файлов манипулатор, което ще се използва за достъп до него. Втората променлива е името на файла, в случая скалара **$stuff** го съдържа. Функцията **die** прекъсва програмата и изкарва последната системна грешка **($!)**, ако отварянето на файла не е било успешно.

Предполагам вече се досещате какво е това **- while (<STUFF>). <> -** този оператор наричан диамант служи за четене от манипулатор. Функцията **while** прочита един ред от файла изпълнява нещо с него и после пак докато се стигне до последния ред. Специалната променлива **$. -** съдържа номера на, реда от който се чете в момента.

```
$out="c:/scripts/out.txt";

open OUT, ">$out" or die "Cannot open $out for write :$!";
for $i (1..10) {
print OUT "$i : The time is now : ",scalar(localtime),"\n";
}
```

В този скрипт първо отваряме файл и го означаваме с файловият манипулатор **OUT.** Функцията **for** се изпълнява **10** пъти, като всеки път записва във файла **out.txt** текущата дата и час. Когато на **print** му се посочи даден файлов манипулатор, в случая **OUT,** знае че записите трябва да се запишат в него. Ако не е зададен файловият манипулатор, използва по подразбиране **STDOUT,** което е монитора на компютъра.

Ако отваряме файл за четене то операцията е **open (STUFF, “file.txt”)**
Ако отваряме файл за запис, като изтрива съдържанието на файла или той се създава ако не съществува, операцията е **open (STUFF, “>file.txt”)**
Ако отваряме файл за запис, който допълва вече съществуващият файл, без да изтрива съдържанието му, операцията е **open (STUFF, “>>file.txt”)**

```
open(FHANDL, "filename");          # прочита от съществуващ файл
open(FHANDL, "<filename");         # същото
open(FHANDL, ">filename");         # създава файл и записва в него
open(FHANDL, ">>filename");        # допълва съществуващ файл
open(FHANDL, "| output-pipe-command"); # изпраща информация
open(FHANDL, "input-pipe-command |");  # приема информация
```

seek (ФАЙЛОВ_МАНИПУЛАТОР,позиция,начална_точка**)**

Командата **seek** е почти същата като във **C. seek** променя мястото от което се чете или записва във файл**.** начална_точка може да е **0,1,2 ,** което отговаря на началото на файла**,** текущата позиция във файла или края на файла**.** Позицията се изчислява спрямо посочената начална точка**.**

Директории - отваряне, четене и затваряне на директория

За да отворим една директория командата е подобна на тази за отварянето на файл**.**

```
opendir DIR, $dir or die "Can't open directory $dir: $!\n";
```

За да прочетем съдържанието на директорията използваме **readdir.**

```
@files=readdir(DIR);
```

И накрая трябва да затворим манипулатора на директорията с**:**

```
closedir DIR;
```

Примера по**-**горе**,** може да го направим по следният начин като съчетаем знанията си досега**:**

```
$dir= shift || '.';

opendir DIR, $dir or die "Can't open directory $dir: $!\n";

while ($file= readdir DIR) {
       print "Found a file: $file\n";
}
```

Първата разлика с горния пример**,** е че ако няма стойност в **@ARGV (**зададените стойности след името на скрипта от командния ред се съдържат в **@ARGV),** то **$dir = .,** което ще е текущата директория**.** След това отварихме директорията задаваме възможност**,** ако директорията не може да бъде отворена то програмата да прекъсне и да ни се изведе съобщение защо приключва**.** Функцията **readdir** прочита съдържание на един ред и го подава на скалара **$file.**

```
@files=readdir(DIR);
```

По този начин се прочита цялото съдържание на директорията и попълваме с нея масива **@files.**

glob

```
@fa=<C:/*.*>;          # Масива се попълва със всички файлове в директорията C:/
@fa=<C:/*.html>;       # Попълва се само от файлове с разширение html
@ga=glob ("C:/*.com"); # Прави същото, но тук може да се задава и със скалар,
допуска интерполация
@ga=glob ("$ааа/*.com"); # например в $ааа е зададен пътя
```

Функцията **chdir** променя текущата директория**,** от която може да се чете**,** на друга подадена на **chdir,** в случая**,** която е зададена от скалара **$dir,** който пък си взима

стойността от командния ред. След това **while** прочита всички файлове в текущата директория, преминава ги през теста **-f,** и ако са файлове се изпечатват на екрана.

```
$dir =shift; # shifts @ARGV
$type='txt';

chdir $dir or die "Can't chdir to $dir:$!\n" if $dir;

while (<*.$type>) {
       print "Found a file: $_\n";
}
```

В директорията ще се търсят файлове само с разширение зададено от **$type,** което в случая е **txt.**

Функциите telldir и seekdir

По-долу е представен работещ скрипт, който ще Ви даде представа за използването и значението на двете функции **telldir** и **seekdir.**

```
# Променяме местоположението и вече всички файлове който ще отваряме
# ще се отварят от директорията D:/
chdir "D:/";

# Отваряме директорията . - което означава, че ще отворим D:/ (ако сме там разбира се)
opendir (GLAV,".");

# Прочитаме всеки файл или директория който се намират в D:/ (не трябва да Ви
# смущава, че това е дял на диска, просто така ми е по удобно за писане - разглеждайте
# го като директория все едно, че е C:/my documents/mitko/
while ($a=readdir GLAV) {

# Подаваме на $b къде се намира манипулатора на директорията в момента
# Това е число което отговаря на файл или директория която е прочетена последна
$b=telldir GLAV;

# Изпечатваме дадения файл или директория и позицията му
print "$a\n$b\n";

# Създаваме масива @dirtell който ще съдържа всички позиции на файловете и
директориите в D:/
# за да може да ги използваме после с функцията seekdir
push @dirtell,$b; }

# По-долу на мястото на XXX в $dirtell[XXX] въведете някаква стойност валидна за
дължината на масива
# и ще видите, че ще бъде прочетена файла или директорията който отговаря на
посочената стойност
seekdir GLAV,$dirtell[XXX];
$v=readdir GLAV;
print "$v\n"
```

seekdir връща манипулатора на директорията в посочената му от **telldir** позиция. Ако се направи пак един цикъл с **while,** ще видим, че четеното на директорията ще започне от позицията която е посочена на **seekdir.**

rewindir, mkdir и rmdir

rewindir се използва за да се върне манипулатора на директорията в началната позиция и ако четем от манипулатора**,** то ще четем от началото му**,** независимо докаде сме били стигнали преди използването на **rewindir.**

```
rewindir GLAV;
```

Чрез **mkdir** създаваме директория**,** а чрез **rmdir** изтриваме директория**.** Изтриването на директорията ще бъде успешно само ако директорията е празна**.**

Логически оператори

Логически оператори са **OR, NOT, AND** и др.. Те всичките оценяват изрази. Оценените изрази връщат **true (**истина **- 1)** или **false (**лъжа **- 0),** в зависимост какъв критерии се използва за оценка от операторите.

or

or оператора работи както следва:

```
open (STUFF, $stuff) or die "Cannot open $stuff for read :$!";
```

Този код означава **-** ако операцията по отварянето на файла **STUFF** пропадне, тогава се спира програмата. Друг пример:

```
$_=shift;
/^R/ or print "Doesn't start with R\n";
```

Ако регулярния израз върне лъжа **(**не е намерил ред започващ с **R),** тогава се изпълнява каквото е от лявата страна на **or .** Както знаете**, shift** изпълнвява действие върху **@ARGV,** ако не е зададена стойност**,** или върху @_ във подпрограма**. Perl** има два **OR** оператора. Единия ни е известния **or,** а другия е **||.**

Приоритет: Кое се изпълнява първо

За да разберете разликата между двата оператора**,** трябва да поговорим за приоритета при изпълнението. Добър пример е следния:

```
C:>perl -e"print 2+8
```

което както знаем ще изпечата **10.** Но ако ние направим така:

```
C:>perl -e"print 2+8/2
```

Сега**,** това **2+8 == 10,** разделено на **2** ли е или може би **8/2 == 4,** плюс **2 == 6?**

Приоритет е кое ще се изпълни първо. В примера горе може да видите**,** че делението се извършва преди събирането. Следователно**,** делението има предимство пред събирането.

Може да използвате скоби за да е по-чисто и ясно**,** коя след коя команда да се изпълнява:

```
C:>perl -e"print ((2+8)/2)
```

И така основната разлика между **or** и **||** е приоритета на изпълнение. В примера по-долу**,** ние се опитваме да присвоим на две променливи стойностите на два несъществуващи елемента на масив. Това ще пропадне:

```
@list=qw(a b c);

$name1 = $list[4] or "1-Unknown";
$name2 = $list[4] || "2-Unknown";

print "Name1 is $name1, Name2 is $name2\n";
print "Name1 exists\n" if defined $name1;
print "Name2 exists\n" if defined $name2;
```

Изходът е интересен. Променливата **$name2** е създадена с фалшива стойност. Обаче, **$name1** не съществува. Причината е в приоритетите на операциите. **or** оператор има по нисък приоритет от **||** .

Това означава, че **or** поглежда на входа от лявата страна. В този случай, това е **$name1 = $list[4]** . Ако той е истина, всичко е наред. Ако е лъжа, дясната страна се пресмята, и лявата страна се игнорира, все едно никога не е съществувала. В примера горе, когато лявата страна се открива че е лъжа, дясната се пресмята, която е **"1-Unknown"** която може да е истина, но води до някакво действие **(**изход**).**

В случая на **||** , който има по-висок приоритет, израза от лявата страна веднага се пресмята. В случая е **$list[4].** Той е лъжа, така че веднага се пресмята кода от дясната страна на оператора. Но кода от лявата страна които не е изчислен, **$name2 =** не е забравен. Следователно израза се изчислява не **$name2 = "2-Unknown".**

Примера по-долу ще Ви помогне да си доизясните нещата:

```
$ele1 = $list[4] or print "1 Failed\n";
$ele2 = $list[4] || print "2 Failed\n";

print <<PRT;
ele1 :$ele1:
ele2 :$ele2:
PRT
```

Друг пример:

```
$name1 = $list[4] or "1-Unknown";
($name2 = $list[4]) || "2-Unknown";

print "Name1 is $name1, Name2 is $name2\n";
print "Name1 exists\n" if defined $name1;
print "Name2 exists\n" if defined $name2;
```

Сега, **($name2 = $list[4])** е самостоятелен израз който се изчислява, а не само **$list[4],** така че ние получаваме същия резултат, ако бяхме използвали **or** .

Трети пример:

```
# $a е равно на $b, ако $b е истина, иначе $c
$a = $b || $c;

# $x е равно на $y, ако $x не е истина
$x ||= $y

# $a е равно на $b, ако $b е дефинирана, иначе $c
$a = defined($b) ? $b : $c;
```

And

Логическия оператор **AND** оценява два израза, и връща **true** само ако и двата са истина. Контраста е с **OR,** който връща истина само ако един или повече от два израза са истина. **Perl** има няколко **AND** оператора.

Първия вид **AND** е **&& :**

```
@list=qw(a b c);

print "List is:@list\n";

if ($list[0] eq 'x' && $list[2]++ eq 'd') {
print "True\n";
} else {
print "False\n";
}

print "List is:@list\n";
```

Изхода тук е лъжа. Ясно е, че **$list[0]** не съдържа **x . AND** оператора може да върне истина само ако и двата израза са истина, и така след като първия е лъжа **perl** решава, че няма смисъл да пресмята втория.

Втория вид **AND** е **& .** Подобен е на **&& .** Помъчете се да разберете разликата в кода по-долу:

```
@list=qw(a b c);

print "List is:@list\n";

if ($list[0] eq 'x' & $list[2]++ eq 'd') {
print "True\n";
} else {
print "False\n";
}

print "List is:@list\n";
```

Разликата е, че се пресмята дясната част на израза независимо от резултата на лявата част. Въпреки факта, че **AND** оператора не може да върне истина, **perl** продължева напред и пресмята втория израз при всички положения.

Третия вид **AND** е познатото ни **and .** Има същата чуствителност, като **&&,** но е с по-малък приоритет. Всички указания за **||** и **or** могат да се приложат и тук.

Други логически оператори

Perl има **not,** който работи като **!** като изключим, че има малко предимство. Ако се чудите къде сте виждали **!** преди, ето два пример:

```
$x !~/match/;

if ($t != 5) {
       print “no no no";
}
```

Има и едно особенно **OR,** или **XOR.** Ако единия израз е верен, **XOR** връща истина
Ако и двата израза са лъжа, **XOR** връща лъжа **(**неистина**)**
Ако и двата израз са истина, **XOR** връща лъжа **(**основната разлика със **OR)**
Това се нуждае от примери. Имате две момичета **Jane** и **Sonia,** които не искате да дойдат

заедно на вашето парти. Вие ще направите малък скрипт като този по-долу, който ще изпълнява ролята на фейс контрол.

```
($name1,$name2)=@ARGV;

if ($name1 eq 'Jane' xor $name2 eq 'Sonia') {
print "OK, allowed\n";
} else {
print "Sorry, not allowed\n";
}
```

Предлагам да изпробвате скрипта със следните стойности:

perl script.pl Jane Karen
(една истина**,** една лъжа**)**

perl script.pl Jim Sonia
(една истина**,** една лъжа**)**

perl script.pl Jane Sonia
(и двете са истина**)**

perl script.pl Jim Sam
(и двете са лъжа**)**

Външни команди

exec

Exec спира работата на скрипта Ви и стартира каквото сте му посочили, като не връща резултат. Ако не може да стартира външния процес, връща код за грешка. Не работи както трябва под **Perl** за **Win32.**

system

Стартира външна команда, която се изпълнява заедно със скрипта. Винаги връща изходният и статус, но не и изходният поток. Това означава, че Вие може да тествате, дали програмата Ви работи. Всъщност Вие тествате да видите дали тя може да бъде стартирана, какво прави, когато е стартирана и каквото прави е извън ваш контрол, ако използвате **system.**
С примера по-долу ще илюстрирам **system** в действие. Стартирайте **'vol'** от командния ред първо ако не сте запознат с нея за да видите резултата от нея. След това стартирайте **'vole'** командата.

```
system("vole");

print "\n\nResult: $?\n\n";

system("vol");

print "\n\nResult: $?\n\n";
```

Както виждате, успешно системно извикване връща като резултат **0.** Неуспешното връща число което е необходимо да разделите на **256** за да намерите истинската върната стойност. Също така забележете, че може да видите и изхода от командата.

Backticks

Между използването на `` и **system** и **exec** има разлики. Те също стартират външни процеси, но връщат изхода от процеса. След това Вие може да правите каквото си поизкате с изходната информация.

```
$volume=`vol`;

print "The contents of the variable \$volume are:\n\n";

print $volume;

print "\nWe shall regexise this variable thus :\n\n";

$volume=~m#Volume in drive \w is (.*)#;
```

```
print "$1\n";
```

Както виждате, командата **vol** се изпълнява. След това извеждаме на екрана изхода на командата. След това с малък регулярен израз извеждаме името на устройството, на което те.

Кога да се използват системни извиквания

Преди да почнете да използвате външни команди в скрипта си за извикване на **net** команди, обърнете внимание, че вече има написани отлични модули, които вършат отлична работа, и че всяко стартиране на външна команда забавя скрипта ви. Също така по-добре използвайте **readdir**, а не **`dir`** - по-бързо и ефикасно е, това се отнася за всички команди които са вътрешни за **Perl**. Също така се гарантира, че при използването само на вътрешни команди, скрипта ще се изпълнява на **99,99%** от всички съществуващи ОС, докато ако използвате външни команди това няма да е така.

Стартиране на процес

Проблема с **backticks** е че трябва да се изчака целия процес да завърши и тогава да анализирате и обработвате информацията. Това е голям проблем, ако кодът, който се връща е голям или процеса е бавен. можем да стартираме процес и през канал **(pipe)** да обработваме информацията, също както го правим с файл. Кодат по-долу е същия като отварянето на файл с две разлики:

- Използваме външна команда, а не име на файл.
- Канала е **|**, който следва името на командата.

```
open TRIN, "dir |";
while (<TRIN>) {
         print "$. $_";
}
```

Забележете, че **|** означава че информацията ще бъде получена от външен процес. Също така Вие може да подавате информация на външни команди, ако **|** е първия символ.

Регулярни изрази

Общо представяне

Регулярните изрази ни позволяват да търсим шаблони в данните си. Повечето букви и символи просто ще съвпадат със самите себе си. Например, регулярният израз **"test"** просто и точно ще съвпада със символния низ **"test"**. Можете да включите режим, нечувствителен към разликата между малки и големи букви, който ще позволи да съвпадне също така и с **"Test"** или **"TEST"**. Има изключения от това правило, някои символи са особени и не съвпадат със самите себе си. Вместо това те сигнализират, че имат специално значение и трябва да се разглеждат със различно значение от това просто как изглеждат. Например **\w** не е ескейпвана **w**, а нещо друго **-** всяка една буква или цифра от **0** до **9**. Голяма част от този документ е посветена на обсъждането на различни метасимволи и тяхното действие. Ето един пълен списък на метасимволите:

\ | ( ) [] {} ^ $ * + ? .

Може да се изключат специалните значения, използвайки ескейп последователността **\Q** - след нея **14** специални знака по-горе автоматично приемат своите обикновенни буквални значения. Това е дотогава, докато **Perl** не види **\E** или края на шаблона. Например **/\Q$future\E/**, спира интерпретирането на променливите.
Първият метасимвол, на който ще обърнем внимание с **"[]"**. Използват се за определяне на клас от символи, представляващ набора от символи, които искате да използвате за съвпадение. Символите могат да бъдат изброявани индивидуално или един диапазон от символи може да бъде обозначен чрез два символа и разделител **"-"**. Например, **[abc]** ще пасне с всички символи **"a"**, **"b"**, или **"c"**; това е същото както **[a-c]**, където се използва диапазон, за да изрази същия набор от символи. Ако искате да пасне, с която и да е малка буква от латинската азбука, то би трябвало да бъде **[a-z]**. Метасимволите не са активни вътре в класовете. Например **[akm$]** ще пасне с всеки от символите **"a"**, **"k"**, **"m"**, или **"$"**; **"$"** обикновено е метасимвол, но вътре в класа от символи той е лишен от особената си природа. Допълвайки набора, можете да пасвате символи, които не са в дадения диапазон. Това се посочва чрез добавянето на **"^"** като първи символ от класа. Поставен където и да е другаде, **"^"** просто ще съвпада със символа **"^"**. Например, **[^5]** ще пасне с всеки символ, с изключение на **"5"**. Може би най-важният метасимвол е обратно наклонената черта, **"\"**. Обратно наклонената черта може да бъде последвана от различни символи, за да се обозначат различни специални последователности. Тя също се използва и за да се избегнат всички метасимволи, така че все пак да можете да ги използвате за съпоставка в образци. Например, ако искате да съвпаднете **"["** или **"\"**, те трябва бъдат предшествани от обратно наклонена черта за да се премахне специалното им значение: **\[** или ****. Някои от тези специални последователности представят предварително дефинирани набори от символи, които много често влизат в употреба, като например наборът от цифри, или наборът от латински букви, или наборът от всички символи, които не са празни **(whitespace)**. На разположение са следните предварително дефинирани специални последователности:

\d - Пасва с всяка десетична цифра; това е еквивалент на класа **[0-9]**.
\D - Пасва с всеки символ, който не е цифра; това е еквивалент на класа **[^0-9]**.
\s - Пасва с всеки празен символ; това е еквивалент на класа **[\t\n\r\f\v]**.

\S - Пасва с всеки не-празен символ; това е еквивалент на класа **[^ \t\n\r\f\v]**.
\w - Пасва с всеки буквеноцифров символ; това е еквивалент на класа **[a-zA-Z0-9_]**.
\W - Пасва с всеки не-буквеноцифров символ; това е еквивалент на класа **[^a-zA-Z0-9_]**.

Тези последователности могат от своя страна да бъдат включвани в класове от символи. Например, **[\s,.]** е клас от символи, който пасва с всеки празен символ, или **","** или **"."**.

Първият метасимвол за повторения, който ще разгледаме, е *****. ***** не съвпада буквално със символа **"*"**; вместо това, той указва, че предшестващият символ трябва да се среща нула или повече пъти, вместо точно веднъж. Например, **ca*t** ще пасне с **"ct" (0** символа **"a")**, **"cat"(1 "a")**, **"caaat" (3** символа **"a")**, и тъй нататък.
Да разгледаме израза **a[bcd]*b**. В началото той пасва с буквата **"a"**, после с нула или повече букви от класа **[bcd]**, и накрая завършва с **"b"**.
Друг метасимвол за повторение е **+**, който пасва един или повече пъти. Обърнете внимание на разликата между ***** и **+**; ***** пасва нула или повече пъти, така че, това което трябва да се повтаря може изобщо да не се среща, докато **+** изисква минимум едно срещане. За да използваме подобен пример, **ca+t** ще пасне с **"cat" (1** символ **"a")**, **"caaat" (3** символа **"a")**, но няма да пасне с **"ct"**.
Съществуват още два квалификатора за повторение. Въпросителният знак **?** пасва с едно или с нула повторения. Можете да мислите за него като за белег за нещо незадължително. Например, пиво-**?**варна пасва хем с "пивоварна", хем с "пиво-варна".
Друг квалификатор е **{m,n}**, където **m** и **n** са десетични числа. Той обозначава, че трябва да има поне **m** на брой повторения, но най-много **n**. Например, **a/{1,3}b** ще пасне с **"a/b"**, **"a//b"**, и **"a///b"**. Но няма да пасне с **"ab"**, защото не съдържа наклонени черти, или с **"a////b"**, защото съдържа четири.

Можете да изпуснете някоя от стойностите на **m** или **n** - например **{,4}** или **{2,}**. В такъв случай, на нейно място се приема някаква разумна стойност. Изпускането на **m** се тълкува като **0** за долна граница, докато изпускането на **n** води до установяването на безкрайността като горна граница.

Читателите от редукционалисткия лагер може би са забелязали, че останалите **3** квалификатора също могат да бъдат изразени чрез тази система за означаване. **{0,}** е същото като *****, **{1,}** е еквивалент на **+**, и **{0,1}** е същото като **?**. Но е по-добре да използвате *****, **+**, или **?**, просто защото те са по-кратки и лесни за четене.

Регулярният израз се задава така **$http=~m/^http://([^/]+)(.*)/**. В края на него може да се окажат модификатори, които повлияват на интерпретирането на самия рефулярен израз - променя неговото поведение. Значението на модификаторите е следното:

i	Игнорира големината на буквите, които се срещат в изследвания стринг.
m	Третира низ като съставен от много редове. Например **"one\ntwo"**.
s	Третира стринга като съставен от един ред. Позволява използването на **"."** за намирането на символа за нов ред.
x	Игнорира празните места и символа за нов ред в регулярния израз. Позволява писането на коментари.
o	Компилира регулярния израз само веднъж.
g	Открива всички съвпадения в дадения стринг (не спира при първото съвпадение). Ако се постави котвата **/G** в началото на регулярния израз ще го закотви в крайната точка на последното съвпадение.

Друг начин на писане на коментари в регулярните изрази е чрез **(?#)**. Например:

```
/Mitko (?# This is me) e pich/
```

Ако имате модификатор, от който искате да се оттървете временно меже да използвате **(?-x)**

```
/Mitko ((?-i) e pich)/i;
```

За да погледнем напред, за търсена дума и само тогава стринга да има съвпадение използваме:

```
/Mitko (?=e pich)/i;
```

За негативно използваме:

```
/Mitko (?!e pich)/i;
```

За поглед назад:

```
/Mitko (?<=e pich)/i;
```

Мета символи и количествени определители

. Съвпада с всеки символ с изключение на нов ред

[..] Съвпада с всеки символ в скобите.

[^...] Съвпада с всеки символ, освен с тези в скобите.

Служат за определение колко пъти може да съвпада един символ.

?	Съвпада с предишния елемент **0** или **1** пъти.	**0 < ? < 1**
*****	Съвпада с предишния елемент **0** или повече пъти.	**0 > * > max**
+	Съвпада с предишния елемент **1** или повече пъти.	**1 > + > max**
{число**}**	Съвпада точно определен брой пъти от числото.	
{min, max}	Съвпада от минимум пъти до максимум, зададени от числата.	

{min,} Съвпада с минимума числа или повече. **{,max}** Съвпада с максимума числа или по-малко. За пример:

```
/a.*e/
```

Означава, че се търси стринг започващ с **"a"** следван от всеки символ **0** или повече пъти следвани от **"e"**. Ще намери думи като **"alpine"** и **"apple"**. Ако искате да укажете колко цикъла да има след **"alpina"** :

```
/apple[0-9]{2}/
```

ще намери съвпадение с **"apple"** следвана от две числа от **0** до **9**, например с **"apple34"**. Ако изкате да укажете някакъв обхват, например да намери **"alpina"** следвана от три до пет числа от **0** до **9**, тогава :

```
/apple[0-9]{3,5}/
```

Съвпаденията ще са например: **"apple123"**, **"apple4321"**, **"apple15243"**, но не и **"apple21"**.

Котви

Определят позицията на съвпадението:

^ Съвпада с началото на реда **(**низа**)**.

$ Съвпада с края на реда **(**низа**).**

\< Съвпада с началото на дума**.**

\> Съвпада с края на дума**.**

\b Търси съвпадение с място между нещо**,** което не е знак за дума и нещо което е**.** Граница между знак **\w** и **\w.**

\B Съвпада с всеки символ които не е в началото или в края на думата**.**

Примери

Имената на месеците**.**
Този алгоритъм намира намира английските имена на дванадесетте месеца в пълната или абревиатурна форма**.**

```
$foo =~ m/
   (?:
      # Абревиатурна форма:
      (?:jan|feb|mar|apr|may|jun|
         jul|aug|sep|oct|nov|dec\.?
      )
      |  # Или...
      # Не абревиатурна:
      (?:january|february|march|april|may|june|july|
            august|september|october|november|december
        )
       )/xi;
```

```
$http = 'http://www.perl.org/index.html';
if ($http=~m#^http://([^/]+)(.*)#) {
       print "host = $1\n"; # host = www.perl.org
       print "path = $2\n"; # path = /index.html
}
```

$1, $2, $3, ... - скаларни променливи**,** които съдържат само цифри в името си се използват в регулярните изрази**,** при намиране на съвпадение**,** съвпадението се предава на такава скаларна променлива за да може да се обръщаме към нея и да я извикваме**.**

```
$ftp = 'ftp://ftp.uu.net/pub/systems';
if ($ftp=~m#^ftp://([^/]+)(/[^/]*)+#) {
          print "host = $1\n"; # host = ftp.uu.net
          print "fragment = $2\n"; # fragment = /systems
}
```

В горния пример **' (/[^/]*)+ ' -** съвпада с **/pub** и **/systems,** нищо**,** че има **+** и съвпада с **/pub/systems,** но в скобите за референция е само едно **/xxx** и се предава само на **$2** се предава последното съвпадение**.**

```
if ($ftp=~m#^ftp://([^/]+)((/[^/]*)+)#) {
        print "host = $1\n"; # host = ftp.uu.net
        print "path = $2\n"; # path = /pub/systems
        print "fragment = $3\n"; # fragment = /systems
}
```

В този случай второто съвпадение е целия път**,** а третото само последното съвпадение**.** Външната двойка **()** за **' (/[^/]*)+ '** е разликата с горния пример**,** която позволява да се маркира за обратна референция **'/pub/systems'.** Предаването на съвпаденията започват от външните скоби към вътрешните**.**

```
if ($ftp=~m#^((http)|(ftp)|(file)):#) {
         print "protocol = $1\n"; # protocol = ftp
         print "http = $2\n"; # http =
         print "ftp = $3\n"; # ftp = ftp
         print "file = $4\n"; # file =
         print "\$+ = $+\n"; # $+ = ftp
};
```

Намира съвпадението с зададения протокол на **URL.** Специалната променлива **$+** съдържа стойността на последното не празно съвпадение**.**

```
$a='I am sleepy....snore....DING ! Wake Up!';
if ($a=~/snore/) {
        print "Postmatch: $'\n";
        print "Prematch: $`\n";
        print "Match: $&\n";
}
```

А ето и един доста добър пример за използване на регулярен израз и оператора **?:**

```
$cipher=$options->{'cipher'};
$cipher='Crypt::DES' if !$options->{'cipher'};
$cipher=$cipher=~/^Crypt::/ ? $cipher : "Crypt::$cipher";
```

Референции

Основи

Референцията е скалар, който посочва мястото в паметта, което съдържа някакъв тип информация. Има **6** типа референции. Дадени са по-долу в таблицата.

Създаване на референция	Дереференция
$refScalar = \$scalar;	**${$refScalar}**
$refArray = \@array;	**@{$refArray}**
$refHash = \%hash;	**%{$refHash}**
$refFunction = \&function;	**&{$refFunction}**
$refGlob = *FILE;	**$refGlob**
$refRef = \$refScalar;	**${${$refScalar}}**

А сега и примери за използването им.

```
my @mass1=qw/we sdfsd gfs fger f s sgds gfdsg dsgf ds g dsg sd/;
my @mass2=qw/123 234 24 32 434 5 34 65 54 643 6 346 43 6/;
my $remas=\@mass1;
my $rema1=(\@mass1,\@mass2);
my $rema2=[@mass1,@mass2];

print "mass1 - @mass1\n";
print "mass2 - @mass2\n";

print "remas - ${$remas}[0]\n";
print "remas - $$remas[0]\n";
print "remas - $remas->[0]\n";
print "remas - ${$remas}[2]\n";
print "remas - @{$remas}[4..6]\n";
print "rema1 - $rema1\n";
print "rema2 - $rema2\n";

my @ma3=[@mass1,@mass2];
print "mas3 $ma3[0]->[0]\n";
print "mas3 @{$ma3[0]}[0]\n";
```

Сложни структури

Сега следва едно малко примерче свързано с използването на референции. Ако искаме да подадем масив на подпрограма, то това не може да стане по нормалния начин.

Трябва да подадем референция към масив. Ако се опитаме да подадем например три масива, то подпрограмата ще получи само един списък от всички стойности на масива. Затова в този случай се използва референция. Малко странно изглеждат всичките тези символи и стрелкички, но с повече практика се свиква. А ето и самия скрипт:

```
my @array1=(1,2,3,4,5);
my @array2=(11,12,13,14,15);

arrayref (\@array1,\@array2);
sub arrayref {
        ($array1_ref,$array2_ref)= @_;
        $length1=@$array1_ref;
        $length2=@$array2_ref;
        if (@$array1_ref=>@$array2_ref) {
                for (0..@$array1_ref) {
                        $array1_ref->[$_]+=$array2_ref->[$_];
                        print "$array1_ref->[$_]<br>";
                }
        }
        else {
                for (0..@$array2_ref-1) {
                        $array2_ref->[$_]+=$array1_ref->[$_];
                        print "$array2_ref->[$_]<br>";
}}}
```

Още един пример за референции, този път с използването на хеш, вложен в масив.

```
%me=(
        name=>"Dimitar",
        familia=>"Mihailov",
        age=>"24"
);
%girl1=(
        name=>"Isor",
        familia=>"not known",
        age=>"26"
);
%girl2=(
        name=>"Aniloap",
        familia=>"I don't say you!",
        age=>"29"
);

@mass_people=(\%me,\%girl1,\%girl2);
print "<br><b>$mass_people[2]{name}</b><br>";
```

За всеки, който иска да разбере референциите, а и всичко останало за **Perl**, най-добре е да прочете първо документацията идваща с **Perl**. Ето един пример от нея, която дава представа как да работим с референции, **my** и да се избегне една редовно допускана грешка.

```
#!perl

use warnings;

for $e (1..10) {
        @array=(&tt);
```

```
        $arr[$e]=\@array;
}

for  (1..10) {
        print join " ",@{$arr[$_]},"\n";
}

sub tt {
        $a=int rand 200;
        $b=int rand 200;
        $c=int rand 200;
        print "$a $b $c\n";
        return $a,$b,$c;
}
```

Скрипта по-горе няма да доведе до очакваният резултат - всеки елемент от масива **@arr** да сочи към различни **@array.** Това е така защото референцията сочи към масив който има постоянно място в паметта**.** Или референцията указва място в паметта**.** Така на ред **6** се указва едно и също място в паметта**,** което ще съдържа последното въведено в нея или цикъла с **$e=10** и стойностите получени от подпрограмата**.** Решението на проблема е с използването на **my** което създава абсолютно нова променлива с ново място в паметта**,** която няма да бъде изтрита докато към нея сочи референция**.** Вторият начин е да се създаде референция към чисто нов ненаименуван масив съдържащ стойността на **@array.**

```
1 #!perl

2 use warnings;
3 use strict;

my @arr;

4 for my $e (1..10) {
5       my @array=(&tt);
6       $arr[$e]=\@array;
7 }
```

```
1 #!perl

2 use warnings;
3 #use strict;

4 for $e (1..10) {
5       @array=(&tt);
6       $arr[$e]=[@array];
7 }
```

ref

Функцията **ref** връща типа на референцията**.** Ако не е референция**,** то тя връща **FALSE (0).**

```
print ref $array1_ref;
```

Функции

Функциите read и eof

read FILEHANDLE,SCALAR,LENGTH,OFFSET

функцията чете от подадения манипулатор, даден брой байтове **(length),** и да ги постави в дадената променлива**. OFFSET** задава от къде да почне да се чете от манипулатора**.** Ако **eof** върне единица значи е достигнат края на файла от който се чете**.**

```
open DD,"apmsqlmgr.exe";
binmode(DD);
read(DD,$ff,100000);
print $ff,"\n";
print "|".eof(DD)."|\n";
close DD;
```

printf

```
$e=123

printf "%x",$e;
```

Резултат **7b. printf** приема подадените му стойности и ги преобразува в това което е посочено от шаблона**.**

%x **-** ще преобразува всяко число което му е подадено в шестнадесетично

%o - преобразува в осмично**,** подаденото му число

%u - преобразува в десетично

%s - преобразува в стринг

%c - преобразува число от кодовата таблица в отговарящият му символ

%f - преобразува в число с десетична стойност **(**с плаваща запетая**)**

Числото между **%** и указателя**,** показва колко да бъдат полетата което да заеме стойността**.**
Ако има **-** между **%** и числото**,** казва да е ляво подравнено**,** това което ще заеме указаните полета**.**
.число **-** указва колко стойности след десетичната запетая да се покажат**,** на число с плаваща запетая**,** остатъка се запълва с **0.**
.число **-** указва максималният брой символи **(**ако е подаден стринг**),** който да се изведат от подаденият стринг

```
my $e=100;
```

```
my $xeks=sprintf "%x",$e;

print "\$e=100 in hex is $xeks\n";

printf "And now in dec %u",$xeks;

my $e=12313.324352352352;

printf "And now %-40f",$e;

printf "And now %-40.20f",$e;

my $e='Mitko e goliam pich';

printf "And now %.3s",$e;
```

grep

```
@files=grep !/^P/, readdir(DIR);
```

На функцията **grep** се задава един списък **(**в горния пример всички файлове в директорията**),** обработва се **-** търсят се всички файлове и директории в директорията**,** които не започват с **P** и така обработения списък се задава на **@files.** Ако търсите във списък и създавате друг списък с нещата**,** които сте открили**,** тогава **grep** е отличното средство**.** Всеки елемент от списъка идва като **$_,** почти идентично с **foreach.**

```
@stuff=qw(flying gliding skiing dancing parties racing);

@new = grep /ing/, @stuff;
```

Имаме масива **@stuff,** който се претърсва за елементи**,** които съдържат в себе си **ing** и списъка с намерените думи се задава на **@new.**

```
@new = grep s/ing//, @stuff;
```

Тук всички намерени думи съдържащи **ing,** се обработват като им се маха **ing** и се връщат на масива **@new.** Трябва да се обърне внимание**,** че в горният пример ще се промени и самият масив **@stuff.**

```
@new = grep { s/ing// if /^[gsp]/ } @stuff;
```

Тук **grep** изпълнява блок от команди задададени в **{}.** Ако думата започва **(^)** с **g,s** или **p ([gsp]),** то само тогава се премахва **ing.**

Малко**,** но много ефикасно примерче за **grep:**

```
@results = grep(/$searchstr/i,@mydata);
```

Map

Map работи по същия начин както **grep,** и двете функции обработват масив и връщат масив**.** Но има две основни разлики**:**

- **grep** връща стойностите на всичко което е обработено и е върнало стойност истина**;**

- **map** връща стойностите на всичко което е обработено независимо дали е извършена дадената операция или не

```
@stuff=qw(flying gliding skiing dancing parties racing);

@mapped = map /ing/, @stuff;
@grepped = grep /ing/, @stuff;

print "There are ",scalar(@stuff)," elements in \@stuff\n";
print join ":",@stuff,"\n";

print "There are ",scalar(@mapped)," elements in \@mapped\n";
print join ":",@mapped,"\n";

print "There are ",scalar(@grepped)," elements in \@grepped\n";
print join ":",@grepped,"\n";
```

От резултата виждате, че **@mapped** съдържа списък от единици. Забележете че са **5** за разлика от оригиналния масив който има **6** стойности. Това е, защото **@mapped** съдържа стойностите истина резултат от **map**.

```
@letters=(a,b,c,d,e);

@ords=map ord, @letters;
print join ":",@ords,"\n";

@chrs=map chr, @ords;
print join ":",@chrs,"\n";
```

Функцията **ord** променя всеки символ на **ASCII** еквивалента му, след това **chr** функцията променя **ASCII** цифрите на символи. Ако промените **map** на **grep** в примера по-горе, ще видите, че нищо няма да се случи. **grep** ще се опитва да обработи подадената стойност с израза, и ако успее ще върне подадената стойност, а не резултата от обработка. Като израз за проверка или изпълнение на **map** или **grep** може да му се зададе подпрограма.

join

За да се започне веднага със разделителят, а не след първата стойност, може да се използва трикът по-долу.

```
print join("name","",@mass_danni);
```

Аналогично, ако искаме да завършва с разделителя ще прибавим една празна стойност в края.

```
print join("name",@mass_danni,"");
```

substr

substr EXPR,POSITION,LENGHT,REPLACEMENT
substr EXPR,POSITION,LENGHT
substr EXPR,POSITION

Функцията **substr** се използва за работа с низове, когато искаме част от него.

```
$a=qq/mitko razbira ot Perl/;

print substr($a,0,5);
substr($a,0,5,"Dimitar");
```

substr приема като първи аргумент низа, вторият е откъде да се започне четенето и третия - колко символа да се прочетат. Има и четвърти, който ако е указан замества посоченият преди това низ. Ако не е указана дължината, то ще се върне всичко до края на низа. Погледнете следният доста интересен пример:

```
substr($a,0,10)=~s/raz/dva/;
```

index

index string, substring, position
index string, substring

Връща нулево-базираната позиция на подниза **substring** в низа **string,** която се намира след зададената в **position** позиция. Ако не е зададена се приема, че е равна на нула. Връща **-1,** ако не е открито съвпадение.

```
$text="That Perl is an easy";
# 012345678.....
$pos=index $text,"Perl";
print $pos
```

Резултат: **5**

```
$pos=index $text,"Perl",12;
print $pos
```

Резултат: **-1.** Започва търсенето от интервала пред **an** и затова не намира съвпадение

split

split преглежда подаденият му низ, и го разделя с разделителя на списък от поднизове, може да му се укаже лимит – до кое съвпадение да разделя и да спре.

```
$line = "c:\\;;c:\\windows\\;c:\\windows\\system;";

@fields = split(/;/,$line); # split $line, като използва ; за разделител

# сега @fields е ("c:\", "", "c:\windows","c:\windows\system")
```

Ако е възможно да има един до друг от символа указан за разделител, ще се попълни една стойност на ммасива с празна стойност, за да се избегне това трябва да се укаже **/;+/** за да съвпадне с един или повече сивмола.

```
@fields = split(/;+/, $line);
```

eval

Понякога има моменти когато няма грешка в кода, а използваме функция която не може да се използва на дадената система. Тогава скрипта ще се прекъсне от фатална грешка, а може би това не е желания резултат от нас в такива случаи. **eval ()** функцията приема израз и го изпълнява. Ако се генериара грешка то тя ще бъде изолирана и няма да засегне изпълнението на скрипта, ако изпълнението на израза е правилно от ще се изведе и резултата от него.

```
eval { alarm(15) };
warn() if $@;

eval { print("The print fuNCtion worked.\n"); };
warn() if $@;
```

Специалната променлива **$@** съдържа всички системни грешки **(**ако има такива**)** върнати от функцията **eval().**

```
do {
    print("> ");
   chop($_ = <>);
   eval($_);
   warn() if $@;
} while ($_ ne "exit");
```

С по-горния скрипт може да проверите кой функции са изпълними на системата ви.

length

length EXPR
length

Връща броя на символите в **EXPR.** Ако **EXPR** е изпуснат, връща броят на символите в **$_.** Тази функция не може да се използва за да се намери от колко елемента се състой даден масив или хеш. За това трябва да се ползва **scalar @array** и **scalar keys %hesh.**

```
print length "asdas";
```

Ще изведе като отговор **5,** защото в стринга има **5** символа.

reverse

reverse LIST

В списъчен контекст, връща елементите на списъка, но от отзад напред. В скаларен контекст, връща символите на стринга в обратен ред.

```
$a="perl"
print reverse $a; #изпечатва: lrep
```

Тази функция може да се използва и върху масив и хеш. При използването му върху хеш, специфичното е, че ако има две еднакви стойности, който при прилагането на функцията вече стават ключове, поради факта, че всеки ключ трябва да е уникален, едната стойност ще се загуби.

```
%by_name = reverse %by_address;
```

Функцията **reverse**, както всичко в **Perl** има значение как я използваме в скаларен или в списъчен контекст.Ако работи над скалар то тя ще обърне всички символи, ако е масив, първият индекс ще стане последен, вторият предпоследен и т.н.

```
$books=qq/Stiven King is a good author!/;
print "$books\n";
$e=reverse($books);
print "$e\n";
@author=qw/King Kunc Strob/;
print reverse (@author),"\n";
print reverse (split / /,$books);
```

rand и srand

rand EXPR
rand

Връща случайно дробно число по-голямо или равно на **0** и по-малко от стойността зададена с **EXPR. (EXPR** трябва да е положително число**).** Ако **EXPR** е пропуснат, използва се стойност **1.** Автоматично извиква функцията **srand(),** освен ако **srand()** не е била извикана вече.

```
$a=rand; # $a ще е със стойност от 0 до 1
$a=rand 100; # $a ще е със стойност от 0 до 100, за стойност може да се задава масив, скалар
```

Стартирайте долният скрипт няколко пъти

```
srand 15;

print rand;
```

Виждате, че резултата от него е едно и също число **(**случайно**).** Това се използва за криптиране и декриптиране **(**елементарни**). srand** с еднакво число, настройва на едно и също число **rand.** Подаденият аргумент на **srand** трябва да е чяло число. За да видите как са използвани за тази цел, разгледайте модула **Crypt::Beowulf** от **CPAN.**

int

int EXPR
int

Не се използва за закръгляване. Подадено дробно число го превръща в цяло. Ако му се подаде число **2.85464** ще върне **2.**

```
$a=int rand 100; # rand ще върне число от 0 до 100, например 43.966723532, а int ще върне 43
```

chr

chr NUMBER
chr

Връща символа представен от число в **ASCII** таблица. За пример **chr(65)** е **"A"** в **ASCII.** За

обратната операция, използваите функцията **ord.** Ако променливата не е зададена, **chr** действа на **$_.**

hex

hex EXPR
hex

Интерпретира **EXPR** като шестнадесетично число и връща кореспондиращата му стойност. За преобразуване на стрингове започващи с **0, 0x** или **0b,** вижте функцията **oct .** Ако **EXPR** не е зададена, **hex** действа на **$_.**

```
print hex '0xAf'; # показва на екрана '175'
print hex 'aF'; # същото
```

lc

lc EXPR
lc

Връща **EXPR,** но с малки букви. Тази функция е аналогична на **\L** ескейпа в стринг затворен с двойни кавички- **"\L string".**

```
chomp ($a=<STDIN>);
print lc $a;
```

При вход **Perl,** на екрана ще бъде изписано **perl.**

lcfirst

lcfirst EXPR
lcfirst

Връща **EXPR,** но с първа малка буква. Тази функция е аналогична на **\l** ескейпа в стринг затворен с двойни кавички- **"\l string".**

```
chomp ($a=<STDIN>);
print lcfirst $a;
```

При вход **PERL,** на екрана ще бъде изписано **pERL.**

uc

uc EXPR
uc

Връща **EXPR,** но с големи букви. Тази функция е аналогична на **\U** ескейпа в стринг затворен с двойни кавички- **"\U string".**

```
chomp ($a=<STDIN>);
print uc $a;
```

ucfirst

ucfirst EXPR
ucfirst

Връща **EXPR,** но с първа голяма буква**.** Тази функция е аналогична на **\u** ескейпа в стринг затворен с двойни кавички**- "\u string".**

```
chomp ($a=<STDIN>);
print ucfirst $a;
```

При вход **perl,** на екрана ще бъде изписано **Perl.**

sort

```
sort( <STDIN> )
```

Ще сортира всички редове на файла и ще върне съдържанието**,** разглежда входа в списъчен контекст**.**

Oneliners (Скриптове от командния ред)

Малък пример

Perl код може да се изпълнява директно и от командния ред. За пример:

```
perl -e"for (55..75) { print chr($_) }"
```

Флагът **-e** показва на **Perl,** че следват команди. Командите трябва да са в двойни кавички, не единични като в **Unix.** Командата в този пример извежда **ASCII** стойностите за номерата от **55** до **75.**

Достъп до файлове

Следващия пример е установена практика за писане на команда за търсене на стринг в даден файл.

```
perl -e"while (<>) {print if /^[bv]/i}" shop.txt
```

while (<>) конструкцията ще отвори, всичко което е в **@ARGV .** В този случай, имаме файла **shop.txt ,** който ще бъде отворен и ще бъдат принтирани редовете започващи с **'b'** или **'v'.** Този резултат може да се постигне и по по-кратък начин. Стартирайте **perl -h** и ще видите списък с опции за командния ред. Една от тях ще използваме сега, тя е **-n,** която слага **- while (<>) { } -** около целия код между **"",** посочен с опцията **-e.** Така че:

```
perl -ne"print if /^[bv]/i" shop.txt
```

прави абсолютно същото като по-горния пример.

@ARGV: Аргументи от командния ред

Perl притежава специален масив наречен **@ARGV.** Това е списък от аргументи следващи името на скрипта на командния ред. Стартирайте следния код:

```
perl myscript.pl hello world how are you
```

```
print "$_\n" foreach (@ARGV)
```

Друг начин да направите същото е :

```
print  while (<>)
```

или:

```
print <> ;
```

Флагове

Вие знаете, че може да пуснете режима с повече предупреждения и обяснения при грешки с **-w** от командния ред. Всяка опция на **perl**, която може да се зададе от командния ред, може да се зададе и в кода на скрипта.

perl script.pl hello

за да изпълни този код:

```
#!perl -w

@input=@ARGV;

$outfile='outfile.txt';
open OUT, ">$outfile" or die "Can't open $outfile for write:$!\n";

$input2++;
$delay=2 if $input[0] eq 'sleep';

sleep $delay;

print "The first element of \@input is $input[0]\n";
print OUY "Slept $delay!\n";
```

който ще се стартира по-същия начин , ако го стартирате така:

perl -w script.pl hello

Много по-удобно е да се слагат флагове вътре в скриптовете. Не е задължително да е **-w** , може да е всеки аргумент който **Perl** поддържа. Стартирайте

perl -h

за пълния списък на аргументите.

Други

DATA, END, LINE, FILE

Пример: Използване на *DATA*

DATA позволява да съхранявате информация само за четене в изпълним файл, която може да Ви потрябва и да имате достъп до нея. Позволява да не се отваря друг файл, а да се използва работния, като се използва оператора за четене . Следващият пример показва:

- Прочитане на всички линии които са след __END__
- Преминаване през всички елементи на масива @lines и принтирането им.
- Всичко над линията __END__ е код, всичко под нея е дата.

```
@lines = <DATA>;

foreach (@lines) {
    print("$_");
}

__END__

Line one
Line two
Line three
```

Тази програма извежда като резултат следното:

```
Line one
Line two
Line three
```

LINE и FILE

__FILE__ - съдържа името на файла от който се чете в момента.
__LINE__ - съдържа номера на реда до който е стигнало четенето в дадения момент на даден файл. Аналогична е и специалната променлива **$.**

Perl и математика

Perl има готови математически функции, които могат да се използват. Ето и примери:

```
print cos(3) . "\n"; # косинус 3
print sin(3) . "\n"; # синус 3
```

Perl няма готова функция тангенс, но ние може да я направим:

```
print (sin(1)/cos(1)) . "\n"; # тангенс 1
```

Много често при писане на скриптове се налага да се генерират случайни числа и символи. Това се постига доста лесно, както ще видите в следващия пример:

```
print int( rand(181) )+ 20;
```

Предполагам си мислите, че този пример връща случайно число в диапазона от **20** до **201**, но всъщност връща цяло число от **20** до **200**. **int()** се използва за закръгляне на числото. Макар, че **int()** не е най-подходящия начин, тук той върши поставената задача. **int()** закръглява винаги на по-малкото, например **3.9** ще бъде закръглено на **3**. **rand(181)** генерира случайно число в диапазона от **0** до **181** във вид **"41.1910741541703"**. След закръглянето прибавяме **20** за долна г**5**раница на случайното число което става ж виапазона **20 - 200**. Обикновенно се слага **srand**; преди генерирането на числото, за да се подобри функията за случайност. В новите версии на **Perl (**по-големи от **5.004)** не се налага използването на **srand**, той се включва автоматично без да бъде викан. При генерирането на символи нещата са малко по сложни. При генерирането на символи нещата са малко по сложни.

```
print chr(int(rand(26) + 65)) . "\n";
```

Горният код генерира случаен символ от A до **Z.**
Възниква проблем когато трябва да закръглявате едно число. **Perl** има функцията **(**по-горе**) int** която закръглява към основата, без да взима под внимание числата след запетаята **-** каквато е основата това ще Ви върне. Ако ви трябва закръгление до някаква точност използвайте **sprintf.**

```
my $num=2.345347257457;

$b=sprintf("%.4f",$num);
print "$num - delimeter 4 - $b\n"
```

Ако Ви се наложи да сравнявате десетични числа **-** то тогава също идва един неприятен момент **(**понякога може да се получат доста странни резултати**).** Проблема има елементарно решение **-** просто махнете **.** и сравнявайте цели числа **(**до **15** цифри**).**

Perl - Internet - CGI

По долу е представен прост скрипт извеждащ обикновенна **html** страница:

```
#!perl

print "Content-type:text/html\n\n";

print "<html><head><title>Test Page</title></head>\n";
print "<body>\n";
print "<h2>Hello, world!</h2>\n";
print "</body></html>\n";
```

Всяки отговор от **Perl** скрипт към **html** трябва да започва с реда **- print "Content-type:text/html\n\n";.** С него се указва как браузъра да обработи отговора и правилно да го покаже на потребителя.
По горния скрипт може да се запише и така:

```
print "Content-type:text/html\n\n";

print "<html><head><title>Test Page</title></head>
<body>
```

```
<h2>Hello, world!</h2>
</body></html>\n";
```

или така:

```
print "Content-type:text/html\n\n";

print <<EOF;
<html><head><title>Test Page</title></head>
<body>
<h2>Hello, world!</h2>
</body></html>
EOF
```

Извеждане на снимка от CGI скрипт

```
#!perl

use strict;

print "Content-type: image/jpeg\n\n";

my $pic="ha11.jpg"; my $lenght=4096; open A,$pic;
binmode A;
binmode STDOUT;
while (read(A,my $buff,$lenght)){
print $buff;
}
close A;
```

Основното което трябва да се запомни е, че има разлика от текстов и двоичен файл. За това и файла и **STDOUT** трябва да се укажат, че ще се чете и извежда двойна информация. Също така типът информация в **header**-ът трябва да указва че това е снимка, за да може сървърът правилно да обработи заявката **- Content-type: image/jpeg.**

Задаване и четене на бисквитки

```
 use CGI qw(:standard :html3);

# Задаване на възможните стандарти за сайта ни.

@colors=qw/aqua black blue fuschia gray green lime maroon navy olive
purple red silver teal white yellow/;
@sizes=("<default>",1..7)

 # прочитане на съществуваща бисквитка. Стойностите и се попълват в хеш

 %preferences = cookie('preferences');

# Ако потребителя си е променил интерсфейса, промените ще бъдат видени и
# подадената информация ще е в съответствие с желанието на потребителя

foreach ('text','background','name','size') {
     $preferences{$_} = param($_) || $preferences{$_};
 }

 # Задаване на стойности по подразбиране за всички.

 $preferences{'background'} = $preferences{'background'} || 'silver';
```

```
$preferences{'text'} = $preferences{'text'} || 'black';

# Изпращане на нова бисквитка

$the_cookie = cookie(-name=>'preferences',
                     -value=>\%preferences,
                     -path=>'/',
                     -expir package ONE;es=>'+30d');
print header(-cookie=>$the_cookie);
```

Основно правило при раабота със сървър е винаги да имате **index.html** файл във всяка директория**.** Друго важно нещо при писане на скриптове е да освобождавате буфера**.** Често има изчакване докато се върне някакъв резултат**,** а през това време заявката вече да е умряла затова започвайте скриптовете си чрез **$|=1 .**

my, local, our

```
sub visible {
print "var has value $var\n";
}

sub dynamic {
local $var = 'local';
visible();
}

sub lexical {
my $var = 'private';
visible();
}

$var = 'global';

visible(); # prints global
dynamic(); # prints local
lexical(); # prints global
```

Избор на променлива по подразбиране

```
$default_file='mitko.txt';
$file=(-f $ARGV[0]) || $default_file;
print "$file\n";
```

В примера ако имаме зададен файл от командният ред той се приема**,** ако няма то тогава използваме зададения в скрипта**.** Не трябва да се използва **or,** защото е с по**-**нисък приоритет от **||.** При **||,** се оценява лявата стойност в скаларен контекст и ако тя е **false ("",0,"0")** се оценява дясната страна и се подава на **$file.** Ако променливата **$ARGV[0]** може да приема стойност **false,** то трябва да използваме **defined** и тринарният оператор **?:.**

```
$default_file='mitko.txt';
$a='';
$b='another def.file';
$file=defined($a)?$b:$default_file;
print "$file\n";
```

В случая **defined** проверява дали скалара е дефиниран **-** дали съдържа някаква стойност**,** пък дори и **0** или **false.** Това се различава от булевата оценка на един скалар**.**

```
$a=''
if ($a) {
print "yes - 1 - $a\n";
}
if (defined $a) {
print "yes - 2 - $a\n";
}
```

Работа с ACSII таблица

Има няколко начина да направите това. За да преобразувате едно число в символ от **ACSII** таблицата се използва **chr,** и обратно символ в поредният му номер от **ACSII** таблицата **ord.** Двете функции приемат само по една стойност.

```
for (1..255) {
print chr;
}
for (a..z) {
print ord;
}
for (1..255) {
printf("%c",$_);
}
```

printf със символа **%c,** заставя всяко число да се преобразува в символ.Ако искаме да обработим цял низ, то трябва да използваме **pack** или **unpack**

```
print unpack("C*","Mitko E good boy"),"\n";
print unpack("U*","Mitko E good boy"),"\n";
print pack("C*","Mitko E good boy"),"\n";
```

Проблем със входните данни

Ето и проблема. Имате низ в кавички, който съдържа два скалара. Искате да се изпечатат стойностите му. За проблема има няколко решения в зависимост от това дали използвате прагмата **use strict.** Под нея не е разрешено да се използва символна препратка **(${$another_scalar}),** не може да използвате текста в един скалар като име на друг.

```
use strict;

my ($cols,$rows)=(10,20);
my $text=q/This is $cols and this is my $rows./; # q// == '' , ако сте
забравили
$text=~s/\$(\w*)/${$1}/ge; # Тук ще се генерира грешка при компилиране
```

За да избегнете този проблем може да премахнете влиянието на прагмата в даденият блок **(**момент**),** чрез **- no strict 'refs'.** Ако не искатее да го правите то може да използвате един малко по-сложен регулярен израз.

```
$text=~s/(\$\w*)/$1/gee;
print "$text\n";
```

При първото заместване **(**е**) $1** съдържа **$cols (**изходния момент**),** при второто заместване **(**второто е**)** се замества стойноста на **$cols** с **10.** Аналогично е и за **$ rows.**

Промяна на големината на буквите

За да промените големината на буквите в един стринг може, както винаги в **Perl** да се подходи по няколко начина. Чрез използването на функциите **up(),lc()** или чрез **\u, \U, \l, \L.**

```
use strict;

my $name=q/mitko/;
print "\u$name\n";
print "\U$name\n";
print "\l$name\n";
print "\L$name\n";
```

Разлика между printf и sprintf

Ако искаме да използваме форматирано извеждане на екрана използваме **printf.** Но ако искаме да модифицираме даден стринг и той да се подаде на скалар то използваме **sprintf.** Изхода отива не в **STDOUT (**или избрания от **select),** а в скалар който извиква функцията. Пример може да видите в частта **Perl** и математика.

За хакерството

Много хора употребяват думи като хакер, кракер и т.н., без да значт истинското им значение. Много хора се тупат в гърдите, след като са използвали някоя програма за разбиване на сайтове или друго незаконно действие. За тях е обяснението на тези прости думички, които все още много малко хора знаят какво означават:

leecher - човек който разменя софтуер и пароли за него. Извесни са и като **warez puppy, warez d00d.**
Script Kidder - това са всички онези който използват програми за незаконни цели без всъщност да знаят как става самото разбиване на сайт. Това са хора без опит, знание и образование в тази сфера. Те просто използват готов софтуер. Кракер който използва скриптове и програми написани от други. Тук попадат **90%** от хората който се мислят за хакери.
cracker - този който нарушава сигурността на една система. Хакерите смятат кракерите за низша форма на живот.
hacker - човек, който се радва при откриването на детайли от системата и как да увеличи възможностите и, за разлика от всички други на които им е достатъчно да изучат само необходимото им. Другото определение е за човек който може да програмира бързо.
phreak - (phone freak) - експерт в телекомуникациите. Хакери с интерес към телефоните и телефонните системи.
White Hat/ Gray Hat/ Black Hat - самото име обяснава нещата. Лошият каубоец е с черна шапка. **Black Hat** са хакери използващи знанията си с незаконни цели.

Съществуват много хакерски кодекси. Ето една точки която се среща в почти всеки от тях:
Хакерът не разрушава системата, той влиза в нея.

Този текст се намира тук, поради факта че **Perl** е едно от основните средства за писане на скриптове от хакерите и кракерите.

POD документация

Perl има много удобен начин за документиране **(**прост текстови формат за форматиране на текст**)** на по-голяма информация в самият скрипт или модул наречен **pod.** Ако започнете ред с **=xxxxx,** всичко след този ред ще се прескочи от интерпретатора на **Perl,** но ще бъде прочетено от транслаторите на **pod.** Четенето ще свърши до ред започващ с

=cut. Всичко след този ред па става работещ код. За да ви стане по-ясно отворете някои модул от разпространяваните с **Perl** и ще придобиете пълна представа. Удобството при **pod** е, че много лесно може да се трансформира в друг формаат. Общото за всички **pod** директиви започват с **=** и последвано с идентификатор.
Има означения които заменят **HTML** тагове, като **I<>,B<>,L<>,** които се тълкуват от **pod** четците като **HTML** тагове.
Специфично е:
=over число
=item символ
=back
Служи за създаване на списъци подобно на **
....<\ul>.** Числото пред **over** показва броя на интервалите преди началото на списъка. Символа указва какъв е символа за списъка.
=begin html,latex...
=end html,latex...
С помоща им се създават за спешифичните четци, цпецифични значения. Например ако е указано
=begin html
То в текста могат да се задават директно тагове, които да си излизат в **out** текста.

За подробности може да разгледате всеки модул които се разпространява. **Perl** се разпространява и с **8-9** модули, специално разработени за работа с **pod Pod::Checker, Pod::Html, Pod::Parser, ...** Също така в **/perl/bin** има доста четци като **pod2text, pod2html, pod2man .....**

perlapp

За да се прочете файл който е прибавен с **perlapp**

```
C:> perlapp --bind name.file[text,0777] script.name
```

```
$e=PerlApp::get_bound_file("bert1.txt");
print $e;
```

Прагми

use strict;

Какво е това строгост **(strict)** и как да я използваме**?** Модулът **strict** ограничава **'**несигурните**,** опасните конструкции**',** съгласно **perl** документацията**.** Няма смисъл да се притеснявате за опасни конструкции**,** ако сте прекарали часове наред в дебъгване на кода си**.** Когато включите модула **strict ,** има три неща които **Perl** държи да са зададени точно**:**

- Променливите **'vars'**
- Референциите **'refs'**
- Подпрограмите **'subs'**

При използването на този модул трябва променливите да се декларират преди използването им**,** като всяка променлива се дефинира с **my .** Това е пример за програма**,** която не използва стриктен режим**:**

```
perl script.pl "Alain James Smith";
```

където **""** затваря стринга като единичен параметър**,** иначе той ще се разглежда като три параметъра**.**

```
#use strict; # махнете '#' след като стартирате кода няколко пъти

$name=shift; # дава първия аргумент от масива @ARGV

print "The name is $name\n";
$inis=&initials($name);

$luck=int(rand(10)) if $inis=~/^(?:[a-d]|[n-p]|[x-z])/i;

print "The initials are $inis, lucky number: $luck\n";

sub initials {
my $name=shift;
$initials.=$1 while $name=~/(\w)\w+\s?/g;
return $initials;
}
```

Вие вече трябва да разбирате какво върши по**-**горния код**.** Когато премахнете **'#'** пред **use strict;** и стартирате кода отново**,** Вие ще получите изход подобен на този**:**

Global symbol "$name" requires explicit package name at n1.pl line 3.
Global symbol "$inis" requires explicit package name at n1.pl line 6.
Global symbol "$luck" requires explicit package name at n1.pl line 8.
Global symbol "$initials" requires explicit package name at n1.pl line 14.
Execution of n1.pl aborted due to compilation errors.

Тези предупреждения означават**,** че на **Perl** не му е ясно какъв е обсега на действие на тези променливи**.** Това означава**,** че Вие трябва да декларирате всяка една от тях с **my,** за

да ограничите изпълнението им в дадения блок, в който се използват, или да им се зададе референцията с тяхното пълно име. В долния пример се използват и двата метода:

```
use strict;

$MAIN::name=shift; # shifts @ARGV if no arguments supplied

print "The name is ",$MAIN::name,"\n";
my ($inis, $luck );

$inis=&initials($MAIN::name);
$luck=int(rand(10)) if $inis=~/^(?:[a-d]|[n-p]|[x-z])/i;

print "The initials are $inis, lucky number: $luck\n";

sub initials {
my $name=shift;
my $initials;
$initials.=$1 while $name=~/(\w)\w+\s?/g;
return $initials;
}
```

Използването на **my** в подпрограма не е нищо ново за вас, а **my** извън подпрограма вече е. Ако се замислите цялата програма е един блок, така че може да се зададе променливите да се виждат само в този блок. Друга интересна част от кода е **$MAIN::name**. Това както може би очаквате е пълното име на променливата. Първата част е името на пакета, в случая **MAIN (**главния, основния**)**. Втората част е името на променливата.

Може да отменяте действието на дадена прагма във всеки един момент посредством **no**.

```
no integer;
no strict 'refs';
no warnings;
```

use warnings;
use diagnostics;

use warnings е задължителна прагма при писането на скриптове. Тя може да Ви помогне при разрешаването на много проблеми, като извежда малки съобщения къде какво се е сбъркало при изпълнението на скрипта ви. Неудовството е, че не винаги тези съобщения са много ясни и подробни и ако например имате цикъл с неинициализирани стойности, на всяко извикване той ще изведе много едни и същи съобщения. Този проблем е решен със прагмата **use diagnostics** която извежда доста по-подробни съобщения, каква е грешката.

use autouse;

Използва се за по късното зареждане на модул, ако бъде извикана някоя от неговите функции. Спомага за по-бързото зареждане на скрипта Ви. Трябва да се внимава как се използва, защото някои модули се нуждаят да бъдат заредени на фаза компилация за да инициализират някои стойности в тях, а **autouse** пречи на това, защото той ги зарежда на фаза интерпретация. Същото важи и за подпрограмите с атрибути използвани в извикваният модул.

Използване и създаване на модули

Има два вида модули традиционни и обектноориентирани или и двете.

Извикване на модули

Един модул се включва чрез:

```
use module;
```

Например:

```
use Tk;
use CGI::Carp;
use Benchmark;
```

Ако не се приложи някакъв лист от имена, то се импортират тези заявени във вътрешния масив **@EXPORT** на модула. Ако има име в листа което го няма в **@EXPORT** или **@EXPORT_OK** ще възникне грешка. Тъй като се импортират символните имена то може да се използват без квалификатора на пакета. Например:

```
use Animal;
camel(); # Animal::camel()
```

Всички модули завършват с **.pm**. Модулите се търсят от **Perl** в масива **@INC** който се зарежда на фаза компилация, затова всички промени в този масив трябва да се правят по време на компилацията. Има няколко възможности, чрез използването на **lib (**прагмата**)** или чрез **BEGIN** блокове и промяна в масива **@INC**. Всеки включен модул в скрипта Ви се включва и в хеша **%INC** като ключ**/**стойност. Ключът е името на файла, а стойността пълното име на пътя.

Прието е имената на модулите да започват с главни букви за да се различават от прагмите, който се записват с малки букви. Прагмите са модули които влияят на процеса на компилиране **(**всъщност те са директиви към компилатора**)**.

package

package се обработва на стадий компилиране, и задава префикса на всички глобални променливи за пакета.

```
package main;
$name='mitko';

package ONE;
$name='nora';
```

```
package TWO;
$name='cveti';

package main;
print "$name, $ONE::name, $TWO::name";
```

Трябва добре да се разбере, че това се отнася до глобални променливи.

GLOB

Едно от най-трудните неща за мен беше да разбера същността на символните таблици и работата с тях, поради това, че когато ги изучавах никъде не бяха обяснени като хората, затова ще се постара да ги обясня добре.

Съдържанието на пакета (скалари, масиви ...) се нарича символна таблица. Символната таблица на пакета се съдържа в хеш с име името ма пакета плюс две двоеточия.

```
package main;
package Finance::Rate;
%main:: ;
%Finance::Rate:: ;
```

Двата хеша по-горе ще съдържат символните таблици на пацкетите за които се отнасят. Всяка символна таблица съдържа референция от тип **GLOB** към хешове чийто ключ/стойност отговарят на името на променливата и стойността и. Когато укажем:

```
*data
```

Това означава, че ***data** сочи към скалар, масив, хеш, подпрограма със име **data**.

```
*OtherData=*data
```

OtherData става псевдоним на **data.** Това означава, че ако променим стойността на **$data** ще променим стойността и на **$OtherData** или ако променим съдържанието **@ OtherData** на ще променим и **@data.** По-голямо примерче да се разбере за какво става въпрос:

```
package AA;

$scal=1;
@scal=qw/mitko RFI/;

package main;

*glb=*AA::scal;

print $glb,"\n";
print "@glb\n";

$glb++;
push @AA::scal,'Iasen Petrov';

print $AA::scal,"\n";
print "@glb\n";
```

Може да правим псевдоним и отделна част от глобалният тип

```
*glb=\$AA::scal;
```

Така скалара е достъпен чрез **$glb**, но ако има масив **@AA::scal**, той няма да е достъпен чрез **@glb**, **$glb** и **$AA::scal** стават едно и също. Псевдонимите ни дават възможност да декларираме константи:

```
*name=\ 'whiteshadow..*.com';
print $name;
```

Понеже глобалният тип е всъщност хеш съдържащ различни типове променливи, то има и друг начин за постигането на присвочване на само една препратка от глобалният тип:

```
*glb=\@AA::scal;
```

а той е:

```
*glb=*AA::scal{'ARRAY'};
```

require и use

require и **use** се използват за включване на модул към скрипт. Разликата е, че:
1. require зарежда модула по време на изпълнение с проверка дали съществува и дали е зареден вече.
2. use зарежда модула по време на компилация. Ако липсва модула се генерира фатална грешка.
3. use импортира символните имена, които сме заявили.

```
use 5.6.0 # Изисква версия на Perl тази или по-голяма от посочената
```

Има два начина за достъп до интерфейса на даден модул - чрез експортиране на символни имена или чрез обръщения към методи.
При обектноориентираните модули не се експортира нищо.

Пример за създаване на модул:

```
package FutureSound;

require Exporter;

our @ISA=qw/Exporter/;
our @EXPORT=qw/music electro punk /; #Експортира по подразбиране
our @EXPORT_OK=qw/$body_music Front242 Sex_Pistols /; #Експортира по заявка
our %EXPORT_TAGS=(               # Подрограмите в хеша трябва задължително да са
our_music=>[qw(electro punk)],   # декларирани преди това в @EXPORT или
@EXPORT_OK
best_bands=>[qw(Front242 Sex_Pistols)],
);
our VERSION=1.01;

sub music {
print "Once was New Generation";
}

$body_music="120 bits";

1;
```

За да използваме този модул в скрипт, то трябва да декларираме:
use FutureSound;
и функцията **music** става достъпна или казано по друг начин всички имена декларирани в **@EXPORT** стават достъпни. Ако искаме да имаме достъп до променливата **$body_music**, то

трябва да го обявим:

```
use FutureSound qw/$body_music/;
```

но тогава подпрограмата **music** става недостъпн, ако искаме и двете то:

```
use FutureSound qw/music $body_music/;
```

Ако не искаме да се внасят никакви имена в скрипта ни а само да се компилира модула, то синтаксиса на това извикване е:

```
use FutureSound ();
use FutureSound qw/:DEFAULT $body_music / ;
#Импортира всичко от @EXPORT и $body_music
```

@ISA, UNIVERSAL, AUTOLOAD

В основата на ОО програмирането е наследяването (използва се и в традиционното също) ставащо чрез специалниат масив **@ISA**. Когато се извика дадена подпрограма която не е декларирана в пакета, то се проверяват модулите които се съдържат в **@ISA** за тази подпрограма, ако я намерят я стартират. Търсенето в масива се извършва в дълбочина – в **@ISA[0]** ако не намери подпрограмата то се търси след това в **@ISA** на претърсения токущо модул. В модулите които се проверяват ако има други **@ISA** и те се проверяват. Ако всичко това приключи неуспешно се търси програма с име **UNIVERSAL::**име_на_подпрограмата. Ако не успее пак претърсва всички пакети в **@ISA** за **AUTOLOAD::**име_на_подпрогра-мата. Най-накрая търси **UNIVERSAL::AUTOLOAD**.

@INC

Ако създаваме модули и искаме да ги държим в различна директория от тези които са включени в **@INC** имаме няколко възможности за действие. Ако например модулите Ви се намират в директорията **E:/PerlModules**, може да постъпите по няколко начина.

Промяна на **@INC** от командния ред:

```
C:>perl –I E:/PerlModules perlscript.pl
```

Попълване на **@INC** в скрипта, чрез използване на прагмата **lib**:

```
#!perl
use lib ‘E:/PerlModules’;
```

Може да Ви се наложи използването на модула **FindBin**, който може да Ви послужи чрез два скалара **$FindBin::Script** – съдържащ името на скрипта чрез който е извикан **Perl** и **$FindBin::Bin** показващ директорията от която е бил извикан скрипта. Това може да Ви послужи за указване на пътища до модули.

```
use FindBin;
use lib "$FindBin::Bin/../lib";

use FindBin qw($Bin);
use lib "$Bin/../lib";
```

Обектноориентирани модули

Обекта е референция принадлежаща към даден клас.
Класът е модул.

Метод е подпрограма от класът на извикващия обект.

```
$myfirst=DBI->connect();
```

$myfirst е обект.
DBI е класът (инвокант).
connect е подпрограма от модула **DBI** (метод).

Обекта е референция която знае към какво сочи. Определението се разбира след като се види как найстина става това.

```
sub new {
        $class=shift;
        $self={};
        bless $self,$class;
        return $self;
}
```

$self е референция към хеш в случая но може да бъде и всякаква друга референция.
$class е класът(модулът) към който знае че принадлежи обекта (референцията).

След като една референция се благослови (това става с **bless**) тя представа дасочи типа си ако я извиклате с **ref**, тя вече сочи към класа (пакета) който принадлежи.

```
$dve=[qw/edno dve tri/];
print ref($dve)," @$dve\n";
bless $dve,"MyClass";
print ref($dve)," @$dve\n";
```

Метод се извиква чрез **->**, а не чрез **=>**

```
$dbi=DBI->connect();
```

което е все едно с:

```
$my_prite=DBI::connect("DBI");
```

Винаги първата стойност, която се предава е името на извикващия клас или препратка към обект. Конструктора е длъжен да предаде класа като втори аргумент на **bless**.

Понякога не знаем предварително кой клас ни трябва,за които имаме едни и същи методи, а го разбираме след дадено условие. Пример за решаването на този въпрос е:

```
$class=$condition ? ”PACK_ONE->” : “PACK_TWO->”;
$main=${class}method(@arg);
```

Референция може да се извиква и, по по неестествени начини:

```
$methodName = "connect";
$dbi->$methodName;
```

Има различни видове методи, който може да създадем.

```
package Books;
use strict;

sub new {
       my ($class) = @_;
       bless {
       _name => $_[1],
```

```
        _author => $_[2],
        _publisher => $_[3],
        _rating => $_[4],
}, $class;
}

sub name { $_[0]->{_name} }
```

Метода **new** се нарича конструктор – създава обекта. Метода **name** може само да чете стойност от обекта, не може да я променя. Тези методи се създават, защото директният достъп до стойност на хеша от потребителя:

```
$book=Books->new('Black Home','King');
print $book->{'_name'};
```

е нежелателно. Нарушава се едно от правилата на ООПрограмиране – да не се използва директен достъп до капсулирана информация. Има много начини да предпазим потребителя на модула от грешката да мисли, че ако подаде стойност ще, промени стойността в хеша.

```
use Carp;
sub read_only
{
croak "Can't change value of read-only attribute " . (caller 1)[3]
if @_ > 1;
}
sub name { &read_only; $_[0]->{_name} }
sub author { &read_only; $_[0]->{_author} }
```

Ако потребителя се опита да подаде някаква стойност, то програмата ще приключи с грешка. Друг начин е чрез използване на по-добри имена на функциите ни, за да може потребителя да се ориентира какъв резултат да очаква. Ако искате стойност функцията ще започва с **get_**, а ако задаваме нова стойност **set_**.

```
sub get_name { $_[0]->{_name}}
sub get_author { $_[0]->{ author }}
sub set_name
{
my ($self, $name) = @_;
$self->{_name} = $name if $name;
}
```

Използването на прототипи за подрограмите, няма да помогне, защото **perl** не ги проверява когато са извикани като метод.

```
sub name();
sub author(;$);

$book->author;
```

Горното извикване няма да провери прототипите.

AUTOLOAD

Ако се извика недефинирана програма ще настъпи фатална грешка, но ако сме дефинирали подпрограма с име **AUTOLOAD**, то тя ще бъде извикана като ще и се предадат

същите аргументи. В глобалната променлива **$AUTOLOAD** ще се съдържа пълното квалифицирано име на извиканата подпрограма.
Ако имаме подобни изреждания за записване или четене на стойности, може да си облекчим работата чрез използването на **AUTOLOAD**.

```
package Books;
use strict;

our AUTOLOAD

sub AUTOLOAD {
       my $class=shift;
       $AUTOLOAD=~/.*::get(.*)/;
       return $class->{$1} if exists $class->{$1};
       print “Not that method, yet”;
}

sub new {
       my ($class) = @_;
       bless {
       _name => $_[1],
       _author => $_[2],
       _publisher => $_[3],
       _rating => $_[4],
}, $class;
}
```

Така ако се извика метод който не съществува във пакета, то ще се извика метода **AUTOLOAD**. Така **AUTOLOAD** може да замести всички **get**_ методи който иначе трябва да напишем. Този начин създава проблем, че потребител може да изиска поле което иначе ние не бихме искали той да има достъп до него.

Други

Ако искате метод да се извиква само за клас, в метода трябва да добавите:

```
sub new {
       my $class=shift;
       die “can’t use class method on object” if ref $class;
}
```

Ако искате метод само за обект:

```
sub new {
       my $class=shift;
       die “can’t use class method on object” unless ref $class;
}
```

Ако има извикване на метод който не съществува в дадения пакет, то се претърсва масива **@ISA**. Ако първия пакет указан в масива **@ISA**, не съдържа търсеният метод, то се претърсва **@ISA** на този пакет.Ако последователното претърсване на **@ISA** завърши безуспешно, то проверката започва наново но се търси метода **AUTOLOAD**.

Функциите в **Perl** които се записват с главни букви като **FETCH**,**STORE**,**BEGIN**,**END** се изпълняват от **Perl** автоматично. Ако в един модул имаме функцията **DESTROY**, тя автоматично ще унищожи дадения обект и ще освободи памет. Това принципно не е нужно да се прави от програмиста, дотолкова доколкото **Perl** го прави автоматично.

Ако имаме данни за даден клас, който трябва да се еднакви за всеки обект от този клас, и ако се променят за класа, то и всички обекти на класа да променят тази стойност, то решението е в следният способ:

```
package A;

use strict;
our $max;

sub new {
       my $class=shift;
       my $self={@_};
       $self->{'max'}=\$max;
       bless $self,$class;
       return $self;
}

sub Max { $max=$_[1]; }

sub printRef {
             my $self=$_[0];
             print "max is ${$self->{max}}\n";
       }
1;
```

Глобалната стойност трябва да е подадена като референция. А долу е скрипта извикващ модула:

```
#!perl

use A;

A->Max(1000);
$edno=A->new();
$dve=A->new();
$edno->printRef;
$dve->printRef;
A->Max(5000);
$edno->printRef;
$dve->printRef;
```

Има съществена разлика дали един метод ще се извика от класа или от референция:

```
$dbi=DBI->connect(); # класен метод, извиква се от класа
$dbi2=$dbi->set();   # екземплярен метод, извиква се от обект
```

Затова при създаване на нов обект трябва да се види дали се вика от класа или от обект, ако е от клас то първата подадена стойност ще е даден клас, а ако е от обект, то първата стойност ще е референция.

```
sub new {
       my $class=shift;
       $class=ref($class) || $class;
       my $self={};
       bless $self,$class;
       return $self;
}
```

Модули за различни OS

Ако създавате модул или скрипт, който ще се използва на различни операционни системи, трябва да се погрижите той да може да е съвместим с **OS.** Ако се нуждаете от даден модул под **Win32** за дадена работа, а има друг модул за същата работа под **FreeBSD**, като и двата модула са несъвместими за различните **OS**, трябва да разберете под каква **OS** се изпълнява модула или скрипта ви и да извикате нужният Ви.

```
BEGIN {
$| = 1;
$OS_win = ($^O =~ /win/i) ? 1 : 0;

if ($OS_win) {
eval "use Win32::SerialPort 0.19";
}
else {
eval "use Device::SerialPort 0.07";
}
die "$@\n" if ($@);
}
```

Същата система може да се използва когато не знаете какви модули ще има на другата машина и има няколко модула с които може да тръгне скрипта Ви.

pod2html

За да извадем **pod** текста на даден файл в **html** файл, командата е

```
C:> pod2html --infile d:/perl/bin/h2xs.bat --outfile
d:/perl/bin/h2xs.html
```

h2xs

Използва се за внасяне на хедър файлове на **C,** чрез помоща на протоезика **XS** в **Perl.** Създава всички необходими файлове с който върви един модул за да може да се изпрати на **CPAN.** Може да се използва и за обикновен модул, чрез извикване подобно на:

```
C:>h2xs -AX Mitko::Modul
```

-A – модула няма да използва **AutoLoad**

-X – модула няма да използва **XS**

-C – не създава файла **Changes.txt,** а вкарва **history-**то в **POD** на скрипта

-O **–** ако съществува вече подобна директория ще бъде записан новия модул върху нея

-P – не създава **POD** във модула

AutoSplit, AutoLoader,SelfLoader

За да заредим един модул по бързо имаме две възможности да използваме **AutoLoader** или **SelfLoader. AutoSplit** създава за всяка подпрограма след __**END**__ в модула отделен файл

с името на подпрограмата и разширение **.al.** Файловете се намират в директория с името на модула**,** която трябва да се намира в **lib/auto/** .

```
use AutoSplit;

autosplit("MyModule.pm", "E:/dir/to/save/filesAL", 1, 1, 1);
```

третата стойност ако е **0** премахва стари **.al** файлове**,** който не се създават от новия модул**,** ако преди това са съществували
четвъртата стойност указва да се провери дали модула използва **AutoLoader** и ако не използва да не прави нищо
петата указва да се провери дали модула е по**-**нов от този който го има **(**ако го има**)**

Когато използваме **AutoLoader,** трябва да декларираме функциите който искаме да се зареждат само при извикване след **__END__.** Тези който знаем**,** че винаги ще ни трябват няма нужда де ги слагате след **__END__.** Един пример с модул използващ **AutoLoader:**

```
package MyModule;

use strict;
use warnings;
use Exporter;
use AutoLoader ‘AUTOLOAD’;

our @ISA=qw/Exporter/;
our $VERSION=1.01;
our @EXPORT=qw/edno dve/;
our @EXPORT_OK=qw/tri/;
our %EXPORT_TAGS=(
      ALLSUBS=>[qw/edno dve tri/]
);

our $digit=1000;
1;
__END__
sub edno {
      my $handle=shift;
      while (<$handle>) {
            print ;
      }
}
sub dve {
      print caller;
}
sub tri {
      print “$digit\n";
}
```

Една важна подробност **–** променливи декларирами с **my** в главния блок**,** няма да са видими във подпрограмите зареждани с **AutoLoader,** затова трябва да се декларират с **our** или прагмата**,** която не трябва да се използва вече **- vars.**

Модули

Benchmark

Както много други неща в **Perl,** един от най**-**добрите начини да решите как да направите скрипта Ви да работи по**-**бързо е да напишете няколко алтернативни кода и да видите кой от тях се изпълнява най**-**бързо с модула **Benchmark.**

```
#!perl
use Benchmark;

open (AA,"access.log");
@data = <AA>;

my $host;
timethese (100,
{ mem => q{
for (@data) {
($host) = m/(\w+(\.\w+)+)/; }
},

memfree => q{
for (@data) {
($host) = m/(\w+(?:\.\w+)+)/; }
}
}
);
```

И резултата**:**
Benchmark: timing 100 iterations of mem, memfree...
mem: 3 wallclock secs (3.24 usr 0.00 sys = 3.24 cpu)
memfree: 3 wallclock secs (2.63 usr 0.00 sys = 2.43 cpu)
Добър резултат**:** вторият код който не иска попълване за обратна референция е с близо **25 %** по**-**бърз от първия**,** в който съвпаденията се запомнят за обратна референция**.**

timethis - изпълнява даден код няколко пъти
timethis (брой изпълнение**,** "код"**)**
timethese - изпълнява няколко парчета код няколко пъти
cmpthese - изкарва резултата от изпълнението на **timethese** като сравнителна графика
timeit - изпълнява даден код и определя колко време продължава това
countit - дава колко пъти се е изпълнил даден код за зададено време

Win32::ODBC

По**-**долу съм дал напълно работещ пример за работа с бази данни на **MicroSoft Access** в **CGI** среда**.** Постарал съм се**,** кода по**-**долу да е ясно написан и описан**.** Предполагам**,** че имате познания по **SQL** или сте прочели ръководството ми за **SQL.** Тук ще намерите допълнение към него**.** Може да копирате кода и да го стартирате така като е**.**

```
#!perl

use warnings;
use CGI::Carp qw(fatalsToBrowser);
# Ако има грешки в скрипта, съобщенията за тях от командния ред се
# извеждат директно на браузъра, а не се записват в error.log на
# Apache, като се изпълняват запитвания на сървъра
use Win32::ODBC;
# Включване на модула за работа с бази данни на
# Win32 операционна система

print "Content-type: text/html\n\n";
print <<EOF;
<head><title>Perl from $HeadHunter_Sid</title>
        <META NAME="Author" CONTENT="$HeadHunter_Sid - Dimitar Mihailov">
        <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=windows-
1251"><style>
BODY {  BACKGROUND: #ffffff;FONT-FAMILY: Verdana, Arial;   FONT-WEIGHT:
normal;font-size : 12px;}
A:link {COLOR: #b82619; TEXT-DECORATION: underline}
A:visited {      COLOR: #80764f; TEXT-DECORATION: underline}
A:hover {        COLOR: #000000; TEXT-DECORATION: underline}
A:active {       COLOR: #b82619; TEXT-DECORATION:
underline}</style></head><body>
EOF

%drivers=Win32::ODBC::Drivers();
#Създава хеш с инсталираните драйвъри на машината

# 1. Има два начина на създаване на връзка към БД, Конфигурация без Perl (външно) -
1.1 и чрез Perl - 1.2
# 1.1. Създаване на нов обект за връзка с бази данни - new(bon) - в скобите трябва да се
попълни със създаден
# драйвер за база данни, това става от Settings -> Control Panel -> ODBC32
# $base=Win32::ODBC->new(bon);
#
# 1.2.|1.2.1|1.2.2|1.2.3|1.2.4|1.2.5|1.2.6|1.2.7|
# Win32::ODBC::ConfigDSN(ODBC_ADD_DSN,"Microsoft Access Driver
#(*.mdb)",("DSN=bon","Description=Tables","DBQ=C:/bons.mdb","UID=","PWD#="));
#
# $base=Win32::ODBC->new(bon); # Създаването на новия обект
#
# 1.2.1. За конфигуриране на DSN (името, под което ще се вика БД)
# 1.2.2. Двайвера за БД - Access; Oracle или др.
# 1.2.3. DSN  името се задава
# 1.2.4. Малко обяснение на БД
# 1.2.5. Локалния път до самата БД
# 1.2.6 и 1.2.7 ако има user name и password за достъп до БД
#

Win32::ODBC::ConfigDSN(ODBC_ADD_DSN,"Microsoft Access Driver
(*.mdb)",("DSN=bon","Description=Tables","DBQ=C:/db.mdb"));
$base=Win32::ODBC->new(bon);
if (!$base) {
        Win32::ODBC::DumpError();
# Задължително трябва да се добавя този блок за да съобщава ако има
# грешки (в случая ако няма създаден достъп до БД)
        die;
}
```

```
$base->Connection();                        # Свързване с БД

if ($base->Sql("Create table bonbions (edno char (22),dve char (123))")) {
# Между "" се пише Sql заявката която се обработва от БД и се връща
# резултата; В случая създавам таблица с две колони
        Win32::ODBC::DumpError();
# Ако заявката не се изпълни, показва се грешката и се прекратява скрипта
        die;
# Забележете, че няма ! пред $base, в случая грешното изпълнение е
# вярно и се изпълнява if конструкцията
}

if ($base->Sql("Create Table employees (firstname char, age integer not null
unique primary key)"))
        {       Win32::ODBC::DumpError(); die;}

if ($base->Sql("Delete from shipping where CarrierID>1 "))
        {       Win32::ODBC::DumpError(); die;}

if ($base->Sql("Update bonbons set [Chocolate Type]='Yellow' where [nut
type]='none'"))
        {       Win32::ODBC::DumpError(); die;}

if ($base->Sql("Select [bonbon name],[nut type],[filling type] from bonbons
order by [nut type]"))
        {       Win32::ODBC::DumpError(); die;}
print "<table>";
while ($base->FetchRow()) {
        ($a,$b,$c)=$base->Data('bonbon name','nut type','filling type');
        print "<tr><td>$a</td><td>$b</td><td>$c</td></tr>";
}
print "</table>";

while ($base->FetchRow()) {
        %col=$base->DataHash();
        foreach  (keys %col) {
                print "$_  -- $col{$_}<br>";
        }
        print "<hr>";
}

@mass_danni=qw/sad dsf ds fdg dfh g t trg df gh ty ht y hrsgsrt htr h df /;
foreach  (@mass_danni) {
        if ($base->Sql("Insert into bonbions (edno) Values ('$_')")) {
# Попълва полето edno на таблицата bonibons със стойностите на масива.
# Задължително подаденията стойности на sql заявката трябва да са в  кавички
        Win32::ODBC::DumpError();  die;            }
}

if ($base->Sql("drop table bb")) { # Изтрива таблицата bb
        Win32::ODBC::DumpError();
        die;}

@TableList=$base->TableList;
# Присвоява на @TableList списък с таблиците в БД

$base->Close();
# Задължително трябва накрая да се прекъсне  връзката с БД, иначе БД
# може да отиде на кино или някъде другаде
```

DBI

За да се свържете със **MySQL** сървъра:

```
$dbh = DBI->connect("DBI:mysql:$database:$hostname:$port", $username,
$password);
$dbh = DBI->connect('DBI:mysql:test;host=localhost',{'RaiseError'=>1}) or die
$!;
```

Ако не се използва **host** по подразбиране ще е **localhost.** Ако не се използва порт по подразбиране ще е **3306.**
Когато приключвате работата със сървъра трябва да прекратите връзката чрез:

```
$dbh->disconnect();
```

За да установите кои са наличните драйвъри на машината Ви използвайте метода **available_drivers.**

```
@drivers = DBI->available_drivers;
@drivers = DBI->available_drivers(1);
```

Този метод връща наличните драйвъри, като претърсва **DBD::*.**
available_drivers(1) - забранява сигнал, че има припокритие на драйвъри.

dump_results

```
$rows = DBI::dump_results($sth, maxlen, $lsep, $fsep, $fh);
```

Функцията прихваща всички редове от **$sth,** извиква функцията **DBI::neat_list** за всеки ред и принтира резултата в манипулатора **$fh. $fh** по подразбиране е **STDOUT,** символа за нов ред **$lsep** е **'\n',** разделителя за полетата **$fsep** е **','** и максималната дължина е **$maxlen 35.** Извежда на екрана всеки ред от получения отговор на **sql** заявката.

quote

```
$sql = $dbh->quote($string);
```

Метода премахва значението на специалните символи.

func

```
func('_ListTables') - създава списък с таблиците които са в дадената БД
@tables = $dbh->func('_ListTables') or die "Unable to list tables: $dbh-
>errstr\n";
foreach $table(@tables) {
print "$table
";
}
```

func('_ListDBs') - връща всички БД който се поддържат **(**които в момента се намират и управляват**)** от **mySQL**
func('_CreateDB') - позволява създаването на нова БД.
func('DropDB') - изтрива БД

```
$rc = $drh->func("createdb", $dbname, [host, user, password,], 'admin');
$rc = $drh->func("dropdb", $dbname, [host, user, password,], 'admin');
$rc = $drh->func("shutdown", [host, user, password,], 'admin');
$rc = $drh->func("reload", [host, user, password,], 'admin');
```

За да разберете най-добре как да използвате модула, разгледайте скрипта по долу:

```
use warnings;
use strict;
use DBI;

my $dbh = DBI->connect( 'dbi:mysql:myDB',    # Указва се да се използва # DBD
драйвъра DBI::MySQL за MySQL и се посочва за активна БД myDB
'my_user_name','my_password',{ RaiseError => 1}  # Ако има грешка то
# програмата ще се прекъсне и ще се изведе грешката, Ако искаме
# продължаване работата на скрипта при откриване на грешка, то трябва
# да се използва следното обръщение
# {RaiseError => 0,
                        PrintError=>1   })
                        || die "Database connection not made: $DBI::errstr";
# $DBI::errstr - съдържа текста на съобщението за грешка
# $DBI::err    - съдържа номера на грешката

my $sql = qq{ CREATE TABLE employees ( id INTEGER NOT NULL,
                                       name VARCHAR2(128),
                                       title VARCHAR2(128),
                                       phone CHAR(8)
                                     ) };
# Проста SQL заявка за създаване на таблица

$dbh->do( $sql );
# do() - изпълнява единични команди, който не са свързани със връщане
# на резултат под формата на масив. По своята същност е съчетание от
# prepare() и  execute(). Удобен за използване при заявки от тип:
# create, insert, delete, update, replace метода do подготвя и
# изпълнява заявката на един път. Използва се за заявки които не връщат
# редове с резултати.

$dbh->do("insert into table_name (first_name,last_name) values
('mitko','mitachki')");

my $sql = qq{ SELECT * FROM employees };
my $sth = $dbh->prepare( $sql );
$sth->execute();

my $sql = qq{ SELECT id, name, title, phone FROM employees };
my $sth = $dbh->prepare( $sql );
$sth->execute();
# Горе са показани как трябва да се изпълняват SQL заявки, който ще
# върнат резултат от тип масив

my( $id, $name, $title, $phone );
$sth->bind_columns( \$id, \$name, \$title, \$phone );

while( $sth->fetch() ) {
  print "$name, $title, $phone\n";
}
# За да видим резултата от нашите заявки има няколко начина, чрез
# fetch(), fetchrow_array(),fetchrow_arrayref(), fetchrow_heshref().
# По-горе е показано използването на bind_columns, който обвързва
# дадена колона със референция към скалар.

$sel2=$dbh->prepare('select * from president');
$sel2->execute();
$m=$sel2->fetchrow_arrayref();
for (1..@$m) {
        print "$m->[$_][0],$m->[$_][1],$m->[$_][2]\n";
```

```
}
$sel2->finish();

# Най-простия начин за получаване на резултат е чрез fetchrow_array.
# Всеки ред на заявката се връща под формата на масив

$sel3=$dbh->prepare('select * from president');
$sel3->execute();
while (@mass_data=$sel3->fetchrow_array()) {
        print "@mass_data\n";
}
$sel3->finish();

$sth->finish();
# Всяка заявка след изпълнението си трябва да я премахнем, чрез
# finish()

$dbh->disconnect();
# След като сме приключили работата си със БД, то трябва да затворим
# манипулатора към нея

__END__
```

По-долу са показани няколко по-сложни начина за работа с БД.
Използването на ? (заместващ маркер) спомага за по-бързото изпълнение на заявките, защото те се подават на сървъра само веднъж, а после се подават само стойностите които трябва да отидат на мястото на ?

```
use warnings;
use strict;
use DBI;

my $dbh = DBI->connect("DBI:mysql:$database:$hostname:$port", $username,
$password);
my @names = ( "Larry%", "Tim%", "Randal%", "Doug%" );

my $sql = qq{ SELECT id, name, title, phone FROM employees WHERE name LIKE ? };
my $sth = $dbh->prepare( $sql );

for( @names ) {
  $sth->bind_param( 1, $_, SQL_VARCHAR );
  $sth->execute();

  my( $id, $name, $title, $phone );
  $sth->bind_columns( \$id, \$name, \$title, \$phone );

  while( $sth->fetch() ) {
    print "$name, $title, $phone\n";
  }
}

$sth->finish();

# Показаното по-долу няма да работи на Mysql, но работи на други
# сървъри поддържащи commit().

my @records = (
                [ 0, "Larry Wall",      "Perl Author",  "555-0101" ],
                [ 1, "Tim Bunce",       "DBI Author",   "555-0202" ],
                [ 2, "Randal Schwartz", "Guy at Large", "555-0303" ],
```

```
              [ 3, "Doug MacEachern", "Apache Man",   "555-0404" ]
            );

my $sql = qq{ INSERT INTO employees VALUES ( ?, ?, ?, ? ) };
my $sth = $dbh->prepare( $sql );

for( @records ) {
  eval {
    $sth->bind_param( 1, @$_->[0], SQL_INTEGER );
    $sth->bind_param( 2, @$_->[1], SQL_VARCHAR );
    $sth->bind_param( 3, @$_->[2], SQL_VARCHAR );
    $sth->bind_param( 4, @$_->[3], SQL_VARCHAR );
    $sth->execute();
    $dbh->commit();
  };

  if( $@ ) {
    warn "Database error: $DBI::errstr\n";
    $dbh->rollback(); #just die if rollback is failing
  }
}

# Долу са показани примери за различните цикли за извличане на редове.
# fetchall_arrayref, връща референция, състояща се от масив от
# референции към масиви, който съдържат стойностите на даден ред

$sel=$dbh->prepare('select * from president');
$sel->execute();
while (@rez=$sel->fetchrow_array()) {
        print "@rez<br>";
}

while ($rez=$sel->fetchrow_arrayref()) {
        print "@$rez<br>";
}

$ref_mas=$sel->fetchall_arrayref();
for (1..@$ref_mas) {
        print "$ref_mas->[$_][0] ";
        print "$ref_mas->[$_][1] ";
        print "$ref_mas->[$_][2] ";
        print "$ref_mas->[$_][3] ";
        print "$ref_mas->[$_][4]<br>";

}

while ($ref_mas=$sel->fetchrow_hashref()) {
        print join " ", (values %$ref_mas),"<br>";
}
$sle->finish();

# когато очакваме да се върне само един ред като резултат може да
използваме selectrow_array()

@da=$dbh->selectrow_array('select * from president where first_name="John"');
print @da;

# Ако искате да запазите върнатите резултати за по нататъшна работа с
# тях има няколко метода с използването на цикъл или като се извлече
# съдържанието наведнъж

$sel=$dbh->prepare('select * from president');
```

```
$sel->execute();
while ($mas=$sel->fetchrow_arrayref()) {
        push @alldata,$mas;
}
$sel->finish();

$scl=scalar @alldata;
$scl2=scalar (@{$alldata[0]});
print $scl2;
foreach $edno (1..$scl) {
        foreach $dve (1..$scl2) {
                print "$alldata[$edno][$dve] ";
        }
        print "<br>";
}

# Метода selectall_arrayref е най-краткия и може би най бързия начин за
# обработка на една заявка. Съчетава в себе си prepare, execute, цикъла
# за извличане и finish

$sth=$dbi->selectall_arrayref(“select * from table”);
# $sth масив от референции сочещи към редове
foreach $row (@$sth){     # $row референция към ред от върнатия резултат
        print “@$row\n”; # ред от върнатия резултат
}

# В горният пример известен проблем е ако имате NULL стойности върнати
# от заявката Ви, те трябва да бъдат обработени отделно за да не Ви се
# извежда въобщението под use warnings – използване на недефинирани
# стойности.

# Обработката на NULL е малко по-различна от традиционната проверка на
# скалар дали е дефиниран или не. Когато дадено поле е недефинирано или
# казано по друг начин няма никаква стойност – NULL, то проверката за
# такива полета трябва да се извърши чрез defined

$sth=$dbi->selectall_arrayref("select * from table");
my $b;
foreach $row (@$sth){
        my $c=1;
        $b++;
        foreach $pole_null (@$row) {
                if (! Defined $pole_null) {
                        print "$b - $c\n"; # отпеатва реда и полето с NULL
                }
                $c++;
        }
}

# Полета с празен низ или със стойност 0, трябва да се търсят след
# горната обработка, защото иначе биха хва нали и полетата със NULL
# стойност

if (! Defined $pole_null)
      { print "$b - $c\n"; # отпеатва реда и полето с NULL }
elsif ($pole_null==’’){print “празен низ”;}
else  (print “поле със стойност 0”;)

# Методът trace(), служи за отстраняване на грешки. Има 9 нива на
# трасировка. Най полезни са 1 и 2 ниво. За да се изкара резултата от
# трасировката във файл, използвайте следното извикване
```

```
$dbh->trace(2,'trace.out');

# Метода може да се извиква за целият скрипт или само за даден
# манипулатор на заявка

$sth=$dbh->do('select * from president');
$sth->trace(1);

# Трасировката е на първо ниво и резултата отива в STDERR

$sth->execute();

# Има атрибути които дават допълнителна информация за заявката.
# Атрибутите трябва да се разделят на специфични за
# DBI и MySQL и другите БД

$dbh=DBI->connect("DBI:mysql:samp_db",user,pass,{RaiseError=>1});
        $sel=$dbh->prepare('select * from president');
$sel->execute();
while (@mas=$sel->fetchrow_array()) {
        print "@mas<br>";
}
$num1=$sel->{NUM_OF_FIELDS};

# Ще изведе броят на колоните съдържащи се в един ред отговор.

print "$num1<br>";
@num2=@{$sel->{mysql_max_length}};

# Връща максималната ширина на стойностите в колоните. MySQL

print "$num2<br>";
@num3=@{$sel->{NAME}};

# Връща имената на колоните. DBI

print "$num3<br>";
$sel->finish();
$dbh->disconnect();

# За правилно подаване на данни към заявка, когато съдържат данни със
# кавички или апострофи, трябва да се обработят предварително за да е
# правилна заявката

$real_name=”Mitko O’Koner”
$name=DBI->quote($real_name);

$sth=$dbi->prepare(“select * from table where=$name);
$dbi->do(“insert into table (name) value ($name)”);

# Забележете, че не се използват ‘’ при value ($name), заявката не е:

$dbi->do(“insert into table (name) value (‘$name’)”);
```

Carp

```
#!perl

  use warnings;
  use strict;
  use Carp;

  local $SIG{__WARN__}=\&Carp::cluck;

  correct();
  incorrect();

  sub correct{
    print_value("Perl");
  }

  sub incorrect{
    print_value();
  }

  sub print_value{
    my $var = shift;
    print "My value is $var\n";
  }
```

Съобщава от къде е била извикана подпрограмата **-** проследява пътя по който е била извикана функцията даваща грешка**.** Пътят е от вътрешното извикване към външното**.** Резултата от по горният код ще е следният**.**

Use of uninitialized value at ./warnings2.pl line 19.
main::print_value() called at ./warnings2.pl line 14
main::incorrect() called at ./warnings2.pl line 7

cluck - ще предупреди за грешката **(warn)** и ще проследи извикването**.**
confess - ще приключи скрипта **(die)** и ще проследи извикването**.**

Text::Tabs

```
#!perl

  use warnings;
  use strict;
  use Text::Tabs;

my $tabstop = 4;
my @lines_without_tabs = expand(@lines_with_tabs);
my @lines_with_tabs = unexpand(@lines_without_tabs);
```

Този модул служи за обработка на табулациите в даден файл**.** В **$tabstop** се задава колко интервала да заместят една табулация **(**по принцип **\t = 8** интервала**.**

Data::Dumper

Функцията **Dumper** взима подаденият и списък и връща подходящ за извеждане низ от **eval** или **do,** като създава копие на оригинала**.** Ако имате сложнаструктура от данни**,** масив от хешове от масиви или други подобни**,** чрез този модул може да разгледате

съдържанието на тази структура, да я запишете във файл за по нататъшно използване или да я предадете на друг модул като **MLDBM** или други модули записващи информация по даден начин. Записаното във файла може да се извика във файла чрез **eval** или **do** и да работите със старата си структура от данни. Като параметър не може да подавате масив или хеш, предавайте само референции към тях.

```
#!perl
use warnings;
use Data::Dumper;

@c = ('c');
$c = \@c;
$b = {};
$a = [1, $b, $c];
$b->{a} = $a;
$b->{b} = $a->[1];
$b->{c} = $a->[2];

open A,">a.txt";
$Data::Dumper::Deepcopy = 1;
print A Data::Dumper->Dump([$b, $a], [qw(*b a)]);
# print A Data::Dumper->Dump([$b, $a], [qw(*b a)]);
close A;
```

Може да извикате функцията **Dumper** и така:

```
print Dumper($foo, $bar);
```

Ако я извикате по този начин то няма да се запазят имената на променливите Ви, затова добавяйте и един масив с имената на променливите при извикването на функцията.

```
print Dumper([$b, $a], [qw(*b a)]);
```

Има различни вътрешни конфигурационни променливи като **Data::Dumper::Deepcopy**, който указват на функцията например до кое ниво да разглеждат референциите, как да представят данните и т.н. Ако има рекурсивни референции (референции към самите себе си) трябва да се използва флага **$Data::Dumper::Purity=1.**

Archive::Zip

Този модул осигурява интерфейс към **ZIP** архив файлове. Модула позволява чрез **Perl** да четете, създавате, екстарквате и манипулирате със **ZIP** файлове. Може да използвате и различни нива на компресия.
Модулът експортира собствени кодове за грешки, подобно на **DBI - AZ_OK, AZ_STREAM_END, AZ_ERROR , AZ_FORMAT_ERROR, AZ_IO_ERROR ,** последните три съобщават за грешка.

```
  die "whoops!" unless $zip->read( 'myfile.zip' ) == AZ_OK;
```

Модула дава възможност всеки файл от архива да се компресира или не.

```
  my $member = $zip->memberNamed( 'xyz.txt' );
  $member->compressionMethod();    # Връща метода на архива
  # задава, че ще се чете некомпресиран файл
  $member->desiredCompressionMethod( COMPRESSION_STORED );
  # задава, че ще се чете компресиран файл
  $member->desiredCompressionMethod( COMPRESSION_DEFLATED );
```

COMPRESSION_STORED – файла не е компресиран
COMPRESSION_DEFLATED – файла е компресиран

Ако компресирате има няколко нива да направите това, но по този начин ще повлияете на скоростта на компресията и разархивирането и големината на файла.

```
$member->desiredCompressionLevel( 9 );
```

Нивата може да са:

COMPRESSION_LEVEL_NONE – не се използва компресия, същото като:

```
$member->desiredCompressionMethod( COMPRESSION_STORED );
```

1.. 9 – **1** е най-бързо, но най-малка компресия
COMPRESSION_LEVEL_FASTEST- синоним на **1** ниво
COMPRESSION_LEVEL_BEST_COMPRESSION - синоним на **9** ниво
COMPRESSION_LEVEL_DEFAULT – компромисен вариант, еквивалентно на **6** ниво, това ниво се използва ако не е зададено друго

Методи на модула:
Конструиране на нов обект:

```
use Archive::Zip qw( :ERROR_CODES :CONSTANTS );
my $zip = Archive::Zip->new();
```

members() - Връща списък на файловете в архива

```
my @members = $zip->members();
```

numberOfMembers() - Връща броят на файловете в архива
memberNames()- Връща имената на файловете в архива
membersMatching($regex) - Връща имената на файловете в архива, който съвпарат с дадения регулярен израз

```
my @textFileMembers = $zip->membersMatching( '.*\.txt' );
my $numberOfTextFiles = $zip->membersMatching( '.*\.txt' );
```

removeMember($memberOrName) – Премахва и връща дадения файл от архива
replaceMember($memberOrName, $newMember) – Премахва файл и го замества с нов

```
my $member2 = $zip->replaceMember( 'abc', $member1 );
```

extractMember($memberOrName [, $extractedName]) –** Изважда от архива дадения файл.Ако е указан път **$extractedName, файла се създава и файла се разархивира там
addMember($member) – Добава файл в архива

```
$zip->addMember( $member );
```

addFile($fileName [, $newName]) – добавя файл,опционално може да му се укаже ново име в архива
addDirectory($directoryName [, $fileName]) – файловете в посочената директория се добавят в архива
writeToFileNamed($fileName) – записва архива в указания файл. Връща както всички други методи на класа **AZ_OK** при успех.

```
my $status = $zip->writeToFileNamed( 'xx.zip' );
die "error somewhere" if $status != AZ_OK;
```

writeToFileHandle($fileHandle [, $seekable]) – записва архива в указания файлов манипулатор

```
my $fh = IO::File->new( 'someFile.zip', 'w' );
if ( $zip->writeToFileHandle( $fh ) != AZ_OK)
   { # error handling }
```

read($fileName) – прочита кои файлове се съдържат в архива.

Заедно с модула идват и **10**-тина скрипта които дават много ясна и нагледна представа как да се работи с него. Долу е даден скрипт който обхожда до **1** ниво директориите в директорията **whiteshadow** и разархивира всички архиви.

```
#!perl
```

```
use warnings;
use strict;
use Archive::Zip qw( :ERROR_CODES :CONSTANTS );

chdir "./whiteshadow";
opendir A,'.';
my @dirs=readdir A;
splice @dirs,0,2;
closedir A;
foreach my $dir (@dirs) {
        chdir $dir;
        my @files=<*.zip>;
        foreach my $zipName (@files) {
                print $zipName;
                my $zip = Archive::Zip->new();
                my $status = $zip->read($zipName);
                die "Read of $zipName failed\n" if $status != AZ_OK;
        my @r = $zip->memberNames();
        print " - @r\n";
                foreach my $file (@r) {
                        $zip->extractMember($file);
                }
        }
        chdir "../";
}
```

Tie::IxHash

Когато обработваме един хеш, понякога ни трябва последният въведен ключ или просто да го обработваме в реда в който сме го въвели хеша. Модулът предлага стандартните функции на **Perl split, push, pop, SortByValue** и други, който позволяват обработката на хеша да бъде подобна на масив.

Tie::RefHash

В **Perl** е наложено ограничението, че не могат да се използват референции за ключове на хеш, а само за негови стойности. Това ограничение може да се избегне като се използва стандартният модул **Tie::RefHash.**

За да повишим бързодействието с работа при големи хешове, удачно е първо да се укаже колко ключ/стойности ще има и после да се попълва. Указанаата стойност трябва да е степен на **2.**

```
keys %BigHash=1024;
```

XML::RSS

Модулът служи за създаване и преработка на **RSS** файлове. Осигурява основния скелет за създавне и манипулиране на **RDF Site Summary (RSS).** Разпространява се с много примери, показващи как да генерирате **HTML** файл от **RSS** файл, преработка на различните версии **0.9,0.91,1.0.** Този тип файлове са прост метод за описание на сайтове, така че други сайтове даобобщят съдържанието му и да сложат линкове към оригиналното съдържание. Идеята на модула е сваляне на **RSS** файлове, обръщането им в **HTML** и използването на част от тях на локалния Ви сървър. Други модули които се използват за обработка на **RSS** файлове са **XML::RSSLite** и **XML::RSS::Tools.** Работата с модулите е лесна и проста и няма смисъл от примери **(**много добри такива се разпространяват със самия него**).**

Chart

Модулът е направен с цел лесно да може да се модифицира самият той. Повечето от интерфейса е зает от **GIFgraph.** И двата модула използват **GD** модула за графичните операции. **GD, GIFgraph, Chart** не се разпространяват със стандартната дистрибуция на **Perl** и за да работите с тях трябва да ги инсталирате чрез **CPAN (**препоръчително**,** защото там винаги са по**-**нови версиите**)** или чрез **ppm** на **ActiveState.**

За да използвате модула за създаване на някаква диаграма трябва да укажете от какъв тип ще бъде. Ако искате линеина**,** то трябва да заредите модула така**:**

```
use Chart::Bars;
```

Ако искате точкова**:**

```
use Chart::Points;
```

и т.н. Засега диаграмите които може да се създават са **Points, Lines, Bars, LinesPoints, Composite, StackedBars, and Mountain**При създаването на новия обект**,** ако го извикаме без аргументи ще създаде графика с размер **400**x**300,** а може ние да укажем размера на графиката**:**

```
$obj = Chart::Bars->new (600,400);
```

Може да задаваме различни опции**,** стандартно като използваме двойки **–** ключ**/**стойности**.**

```
%hash = ('title' => 'Foo Bar');
$obj->set (%hash);

$obj->set ('title' => 'Foo Bar');
```

По**-**долу са обяснени някой от опциите**:**

```
# Дали да бъде прозрачна или не диаграмата, използва се когато има
# зададен цвят за страницата на която ще се изведе диаграмата, например
$obj->set('transparent'=>'true');
# Задава заглавие на диаграмата
$obj->set('title'=>'Bussines Graph');
#Позволява създаването на карта върху диаграмата
$obj->set('imagemap'=>1);
# Задава разстоянието между заглавието/етикетите и графиката
$obj->set('graph_border'=>100);
# Задава името на абцисата
$obj->set('x_label'=>'Months');
# Задава името на ординатата
$obj->set('y_label'=>'Dollars');
# Указва дали да има или няма ('none'), или от коя страна на диаграмата
# да се появи легендата
$obj->set('legend'=>'left');
# Указва етикетите за различните масиви информация
$obj->set ('legend_labels' => [qw/firma_ONE firma_TWO/]);
# Задава дължината на x и y отметки
$obj->set('tick_len'=>33);
# Задава максималната стойност за ординатата
$obj->set('max_val'=>330);
# Премахва сибия фон на графиката, който е по-подразбиране
```

```
$obj->set('grey_background'=>'none');
```

Извикването на **png** метода създава графиката и я запазва във файл**.**

```
$obj->png ("foo.png", \@data);
```

Ако не искате да запазвате всеки път диаграмата във файл Ви се префлага метода **cgi_png.** Метода ще изведе диаграмата**,** със задължителните **http** хедъри**,** на стандартният изход**,** позволявайки да извикате скрипта създаващ диаграмата директно от вашата страница**.**

```
$obj->cgi_png ( \@data );
```

Стойностите в масива създават графиката и се запазва във файла **foo.png.** Както в **GIFgraph, @data** трябва да съдържа референции към масиви с**,** като първият масив трябва да съдържа x**-**етикетите**.**

```
@data = ( [ 'foo', 'bar', 'junk' ],
            [ 30.2,  23.5,  92.1   ],
          [qw/23 34 45 45 56 56/ ] );
```

По**-**долу е даден напълно работещ скрипт**.** Инсталираите си модулите първо**!**

```
#!perl

use warnings;
use Chart::StackedBars;

$obj = Chart::StackedBars->new (500,400);
$obj->set('transparent'=>'true');
$obj->set('title'=>'Bussines Graph');
$obj->set('imagemap'=>1);
$obj->set('graph_border'=>20);
$obj->set('legend'=>'left');
$obj->set ('legend_labels' => [qw/firma_ONE firma_TWO/]);
$obj->set('grey_background'=>'none');


@data = ( [ 'april', 'may', 'jun',qw/july august september october/ ],
          [ 10,  34,  56 ,qw/12 32 34 66/  ],[qw/23 34 45 45 56 56/]);

$obj->png ("foo.png", \@data);
```

За съпоставка по долу съм дал друг скрипт който използва модула **Chart::Plot.**

```
use Chart::Plot;

my $img = Chart::Plot->new (500,400);
$img->setData ([qw/12 53 65 88 44/]);
$img->setGraphOptions ('horGraphOffset' => 75,
                       'vertGraphOffset' => 100,
                       'title' => 'My Graph Title',
                       'horAxisLabel' => 'my X label',
                       'vertAxisLabel' => 'my Y label' );
open A,">x.gif";
binmode A;
print A $img->draw();
close A;
```

Perl/Tk

Основи

За да излезем от дос промпта се нуждаем от начин да създаваме графични приложения за скриптовете ни. Това става възможно чрез използването на модула **Tk.** Той е заимстван от скриптовият език **TCL/Tk.** При създаването на всяко графично приложение под **Perl** има няколко основни стъпки през който трябва да се мине. По-долу е дадено, най-простото и задължително (за някой части), което трябвада се включи в скрипта.

```
#!perl

use warnings;
use strict
use Tk;  # Зареждане на модула

my $main=MainWindow->new(); # Създаване на основния уиджет
# Не задължителната конфигурация
$main->configure(-title=>'PassChecker v.2.0 from HeadHunter_Sid',-
background=>'#3399FF');  # Какво да е заглавието и цвета на задния фон
$main->minsize(qw/310 256/);  # Минималната му големина при стартирането
$main->geometry('+340+200');
# Къде да бъде разположен да десктопа при # появяването му
$main->resizable(0,0); # Дали може да се уголемява или намалява от
# потребителя. В случая не може
$main->setPalette("red");     # Задава цвета на всичко в $main

MainLoop; # Задължително за да може всичко това което сме написали до
# сега да бъде създадено и изведено на екрана
```

Този скрипт създаде един прозорец, в който ще се слагат всички други – радио-бутони, падащи менюта и т.н. За да създадем един прост етикет в който може да запишем нещо, командата е:

```
my $label=$main->Label(-text=>"This is my first label");
```

pack

Запишете това над **MainLoop;** и стартирайте скрипта. Няма да видите това което очаквате. Всеки обект който създаваме трябва да се пакетира и да му се укаже мястото което да заеме в главният прозорец. Това става чрез **pack;**

```
$label->pack();
```

Всъщност има три различни начина за това като **grid** например,но тук ще резглеждаме само **pack.** Принципа на задаване на опции е:

$widget->pack([опция**=>**'стойност'**,** опция**=>**'стойност'**, ...]);**

По-долу са всички възможни опции за **pack().**

```
-side => 'left' | 'right' | 'top' | 'bottom'
```

Поставя **widget** на указаната страна на прозореца или фрейма

```
-fill => 'none' | 'x' | 'y'| 'both'
```

Указва на **widget** да запълни указаната посокаа**.**

```
-expand => 1 | 0
-anchor => 'n' | 'ne' | 'e' | 'se' | 's' | 'sw' | 'w' | 'nw' | 'center'
-after => $otherwidget
-before => $otherwidget
-in => $otherwindow
-ipadx => размер
-pady => размер
-padx => размер
-pady => размер
```

bind

За да свържем натискането на даден бутон с дадено действие трябва да използваме **bind**

```
$main->bind('<Button-1>'=>sub{$label->configure(-text=>"This is rather than
your knowledge and ability")});
$main->bind('<Double-1>'=>\&subSMG);
$main->bind('<KeyPress-Return>', sub  {   print "dada";});
```

Бутона се указава първ**о,** '<Button-1>' отговаря на едно натискане на левия бутон на мишката**.** След това се посочва безименна подпрограма или референция към подпрограма**.** Във вторият случай при две натизкания на бутона на мишката**,** а в третия подпрограмата ще се изпълни при натискане на **Enter.** В долният пример и трите бутона които са указани се отнасят за едно и също **–** всеки натиснат бутон

```
$mw->bind( '<Any-KeyPress>'  => \&Press);
$mw->bind( '<ButtonPress>'   => \&Press);
$mw->bind( '<ButtonRelease>' => \&Press);

sub Press {
         my ($buttonpr) = @_;
         my $e = $buttonpr->XEvent;
         my ( $x, $y, $W, $K, $A ) = ( $e->x, $e->y, $e->K, $e->W, $e->A );
         print "  x coor  = $x\n";
         print "  y coor  = $y\n";
     }
```

font

Ако искаме да зададем даден шрифт за даден етикет или всичко което е свързано с букви**,** то в дефинирането му трябва да зададем шрифта му**:**

```
$font1=$main ->fontCreate(qw/C_small -family courier -size 10/);
$font2=$main ->fontCreate(qw/C_big -family courier -size 14 -weight bold/);
$font3=$main ->fontCreate(qw/C_vbig -family helvetica -size 24 -weight bold/);
$font4=$main ->fontCreate(qw/C_bold -family courier -size 12 -weight bold -
slant italic/);

my $labe2=$main->Label(-text=>"This is my first label", -font=>$font1);
```

```
my $labe3=$main->Label(-text=>"This is my first label", -font=>$font3);
```

Опцията **–wraplength=>1** извежда текста вертикално, а не хоризонтално. Може да използваме даден скалар за попълване на текст в етикета.

```
$tt='Mitko is pich';
$main->Label(-width=>10,-textvariable=>\$tt)->pack();
$main->bind('<Button-1>'=>sub  {  $tt="Yes Yes!";          });
```

Когато създаваме едно голямо приложение, трябва да разделим основният прозорец на по-малки част в които да влагаме свойте менюта за да има някакъв приличен вид приложението ни. Това става посредством **frame** командата.

```
my $commonFrame=$main->Frame(-background=>'#3399FF',-relief=>'groove',-
borderwidth=>3,-height=>500,-width=>200)->pack();
```

-height=>500 - фреймът ще е висок **-** у **- 500 ,** и **-width=>200** широк х **- 200**

-relief=>'groove',-borderwidth=>3 фреймът ще е очетан с граница **(**прави около него**),** дълбочината на правите се задава от **borderwidth**

-relief => 'raised' – ако зададем фрейм с тази опция за разлкика от горния които е вдлъбнат, този ще е изпъкнал **,** ще създава илюзия за триизмерност

-relief => 'sunken' - вдлъбнато, надолу триизмемрно

```
$commonFrame ->Label(-text => "This is Label in our frame")->pack(-padx=>100);
```

По принцип според големината на текста, уиджета ще има дадена големина, с **-padx** ние разширяваме големината му, като в горният случай текста остава в средата, а се разширява около него. С **-pady** е същото, но разширява по вертикала.

Scrolled и Listbox

За да създадем прозорец, отстрани с плъзгачи, ако текстът е по дълъг от прозореца да можем да го видим целия използваме **Scrolled** и **Listbox**

```
my $list = $main->Scrolled(qw/Listbox -width 20 -height 10 -setgrid 1
  -scrollbars osoe/)->pack(qw/-expand yes -fill y/);
```

Тук създадохме прозорец с определена ширина и големина **(-width 20 -height 10),** който ще има плъзгачи само ако има нужда от тях **(-scrollbars osoe),** ако махнем о пред **s** и **e** то те ще се създадат дори и да няма нужда от тях. **s** указва, че трябва да се създаде отдолу, а е от дясната страна на прозореца. Възможните стойности са **s,w,e,n.**

```
$list->insert(0,'And this is a text in a listbox',"second row','and another
row');
```

Тук добавихме текст във вече създаденият прозорец. Ето още един начин за попълване на стойности:

```
$list->insert(0, qw/Alabama Alaska Arizona Arkansas California Washington/,
'West Virginia', 'Wisconsin', 'Wyoming');
```

Три реда,който при натискането на някой от тези редове може да се вземе стойността му. В случая по-долу ще изпишем стойността избрана от нас в стандартния изход.

```
$list->bind('<Double-1>' =>  sub  {print $_[0]->get('active'),"\n"; },
```

Когато имаме повтарящи се стойности за много от обектите който създаваме, може да си спестим писането като използваме масиви:

```
  my(@scrolled_attributes) = qw/Entry -relief groove -scrollbars os/;
  my(@spacer_attributes) = qw/-width 33 -height 11/;
  my $e1 = $main->Scrolled(@scrolled_attributes)->pack(@pl);
```

Нека сега пак създадем един прозорец.

```
$wine_list = $top->Listbox("-width" => 20, "-height" => 5 )->pack();
$comp_list->insert('end', 'pentium','AMD','Celeron');
$comp_list->bind('<Double-1>', \&yourComp);
sub yourComp {
    my $comp = $comp_list->get('active');
    return if (!$comp);
    print "You must have buy Mac not $comp\n";
    $comp_list->delete('active');
}
```

Тук взимаме избраната от потребителя стойност (в зависимост от това какво искаме да направим с нея – правим) в случая ние я махаме от списъка.

```
@w=$main->children;
```
#Връща всички **frame toplevel ...** които са дъщерни за **main**

```
print $er->class;
```
връща какъв тип е обекта

```
if (Exists $frfin) {return;}
```
ако съществува такъв обект да не се създава.

```
@e=$main->geometry();
```

Връща положението на прозореца (**x** и **y**) на десктопа къде се намира

Пример

А сега един малко по-голям пример, в който ще създадем главният прозорец, етикети с текст и поле за попълване с данни.

```
#!perl
use warnings;
use strict;
use Tk;
$main=MainWindow->new();
$main->configure(-title=> HeadHunter_Sid',-background=>'#3399FF');
$main->minsize(qw/310 256/);
$main->geometry('+340+200');
$main->resizable(0,0);
my $font=$main->fontCreate(qw/C_small -family courier -size 10/);
my $dataFakturaFrame=$main ->Frame(-relief=>'groove',-borderwidth=>3)->pack(-
side=>'top');
$dataFakturaFrame->Label(-text=>"Дата:",-font=>$font)->pack(-side=>'left');
$dataFakturaFrame->Entry(-font=>$font,-width=>'10',-background=>"white",-
textvariable=>\my $date)->pack(-side=>'left');
$dataFakturaFrame->waitVariable(\$date); # няма да създаде долното
```

докато не се въведе стойност в **$date**

```
$dataFakturaFrame->Label(-text=>'Фактура N:',-font=>$font)->pack(-
side=>'left');
```

```
$dataFakturaFrame->Entry(-font=>$font,-width=>'15',-background=>"white",-
textvariable=>\my $faktura)->pack(-side=>'left');

MainLoop;
```

Въведените данни ще се попълнят в скаларите **$date** и **$faktura.**

Възможно е да създадете и стандартно **YesNo** или друго **Windows** съобщение**.**

```
my $button = $main->messageBox('-icon' => $iconvar, -type => $typevar,
        -title => 'Message', -message => "This is message");
```

Стойностите за **$iconvar** са **error info question warning.** Те отговарят на иконката която ще се появи на съобщението**.**

$typevar - AbortRetryIgnore OK OKCancel RetryCancel YesNo YesNoCancel – задава типа на бутоните**.**

Когато искаме при натискането на бутона да се изпълни дадена подпрограма има няколко начина да запишем това**.**

1. В масив се поставя препратка към подпрограма**,** и след нея параметрите който искаме да и се подадат

```
   $main->Button(-text => 'Do it',
                 -command => [ \&do_sum , $irate, $RATE ]
                 )->pack;
```

2. Директно да запишем подпрограмата

```
$main->Button(-text => 'EXIT', -command => sub {exit;}     )->pack;
```

Може да не задаваме променлива която да съдържа въведената стойност**,** а да използваме **get** или **Content** в зависимост от типа на обекта**.**

```
my $entry = $main  -> Entry()->pack;
my $text  = $main  -> Text(-width => '10', -height => '10')->pack;
my $ent   = $entry -> get();
my $tt    = $txt   -> Contents();
```

Приложение А. Разлики във версиите на ръководството

4.0.1
- добавено съдържание на ръководството, приложение А, поправка на правописни грешки, промяна в реда на главите за модули
- в глава Модули е добавено обяснение за GLOB, и са допълнени някои въпроси
- пример за разархивиращ скрипт в частта за Archive::Zip
- обяснен selectall_arrayref, обработка на NULL, DBI->quote в часта за DBI
- много други добавки

. . .

mitko_ddt@yahoo.com