END OF FILE – EBOOK PERL

Fórum: http://eofcommunity.com/forum

Autor: kodo no kami (http://www.facebook.com/hacker.fts315) skype: hackerfts315

Greetz: susp3it0@virtual , mmxm, sir.rafiki, lend4pop, gjuniioor, nass, neofito, 0racle

Data: 06/11/2015 – 25/11/2015 (sem revisão)

E ae galera aqui é o kõdo no kami e esse é meu terceiro ebook de programação e nele vou ensinar a linguagem de programação perl que é uma das linguagens que eu mais utilizo devido a facilidade em desenvolvimento e execução, diferente das linguagens nos ebooks anteriores (c/c++ e pascal) essa é uma linguagem interpretada e multiplataforma com isso você pode criar scripts para diversa plataformas sem precisa se preocupar em recompilar para aquela plataforma especifica, como é uma linguagem que tem a sintaxe muito parecida com diversas outras linguagem você consegue aprender outra a partir dessa com muito mais facilidade ou vice versa devido ela ter a sintaxe parecida com varias outras como por exemplo a linguagem C ou PHP, muitas funções da linguagem perl são comandos do próprio unix/linux porem nativa já da linguagem que facilita na criação de scripts para manipulação e automação em servidores linux e algumas dessas funções pode ser usada em outras plataformas também, outro ponto forte dessa linguagem é programação web que é tao boa quanto o php se não melhor em alguns requisitos (talvez meu próximo ebook seja sobre php kkk), como os ebooks anteriores de minha autoria ele pode ser compartilhado livremente porem sem fins lucrativos ou seja a venda desse ebook é proibida sem autorização do autor

Indice

1.0 – Introdução a linguagem perl

A linguagem perl foi criada por Larry Wall em 1987 antes mesmo do sistema linux e ela também influenciou outras linguagens como o próprio PHP e Python, ela é uma linguagem interpretada e de alto nível diferente das linguagens compiladas que gera o executável final como por exemplo a linguagem C, no caso a linguagem perl usa códigos puros que a gente chama de scripts eles são executados linha por linha ou trecho por trecho por um interpretador especifico da linguagem, esse interpretador ler o código linha por linha e executa as funções com base nele, na maior parte das vezes esse interpretador que é o executável na linguagem da maquina para uma plataforma especifica, com isso você não precisa gerar o executável para aquela plataforma porem você precisa de um interpretador dessa linguagem naquela plataforma, a linguagem perl permite uso de regex nativa como operadores da própria linguagem diferente de muitas outras linguagens que tem que usar módulos externos para usar regex, perl também tem muitas funções que são comandos do linux/unix isso já nativo da própria linguagem e formas de manipular textos e arquivos com muita facilidade usando uma sintaxe muito compacta permitindo assim criar scripts muito rápido, perl também pode ser usada para programar para web como CGI ela também tem muitos módulos que vamos ver mais para frente para diversos fins, ela é uma linguagem fracamente tipada e dinamicamente tipada ou seja você não é obrigado declarar tipos de variáveis e as variáveis pode mudar o tipo ao decorrer do programa além do mais os tipos dela são genéricos, a orientação a objeto da linguagem perl é um pouco mais complicada que algumas linguagens devido a pessoa ter

que especificar algumas coisas a mais como criar o construtor e especificar exatamente as entradas para a classe e como manipular ela pela classe (seria como pegar uma class da linguagem java e ter que especificar o ponteiro this para cada um dos os atributos e as entradas referente a ele kkk), já existe o perl vesão 6 (codename rakudo '-') porem nesse ebook não vamos abordar ele vamos mexer com o perl versão 5, talvez eu atualize esse ebook futuramente para falar um pouco sobre o perl 6 e outras coisas

1.1 – Interpretadores e IDE

para programar em perl precisamos de um interpretador da linguagem perl para sua plataforma (windows, linux, mac etc), alguns deles são

https://www.perl.org/get.html (site oficial)

http://www.activestate.com/activeperl/downloads (active state ~ recomendo esse)

http://strawberryperl.com/ (strawberry)

http://dwimperl.com/ (dwin perl)

se você usa linux pode baixar pelo próprio repositório da sua distro caso não tenha ele já instalado (maioria das distro linux costuma vim com perl já instalado)

distro baseada em debian (ubunto, kali):

```
# apt-get install perl
```

distro baseada em redhat (centos)

```
# yum install perl
```

além dos interpretadores tem os ambientes de programação que são chamados de IDE que facilita na hora de programar que é um editor de texto que destaca a sintaxe da linguagem, as vezes ate mostrando as funções ou executando os scripts sem precisar abrir o terminal, algumas IDEs para a linguagem perl são

http://padre.perlide.org/ (padre)
http://www.geany.org/ (geany)
http://www.epic-ide.org/ (perl editor ide for eclipse)
https://notepad-plus-plus.org/ (notepad++)
http://www.nano-editor.org/ (nano)
http://www.vim.org/ (vim) ← só coloquei se não rafiki e gjuniioor ia falar kkkk

uso de IDE é opcional você poderia programar usando qualquer outro editor de texto de sua preferencia bastando salvar o arquivo com extensão .pl, por exemplo com o nome “kodo.pl”

1.2 – Executando um script perl

para executar um script perl primeiramente deve ter o interpretador instalado, para conferir se ele esta instalado abra o prompt de comando (terminal) e digite “perl -v” se aparecer a versão do perl então ele esta instalado porem se quando digitar o comando aparecer a mensagem “comando não encontrado” ou é invalido quer dizer que não esta instalado ou o diretório do perl não é uma variável de ambiente

```
perl -v
```

se ele estiver instalado basta apontar o prompt ou terminal para o diretorio onde esta o script, para isso usamos o comando “cd” seguido do diretorio (esse comando é tanto para o windows quanto para o linux), no meu exemplo ele esta no desktop do meu linux (/home/kodo/Desktop)

```
cd /home/kodo/Desktop
```

depois basta digitar perl seguido do nome do script para executar ele, meu exemplo o script se chama fts.pl

```
perl fts.pl
```

se o nome do script tivesse espaço eu teria que colocar o nome dele entre aspas, nesse meu exemplo o script se chama kodo no kami.pl

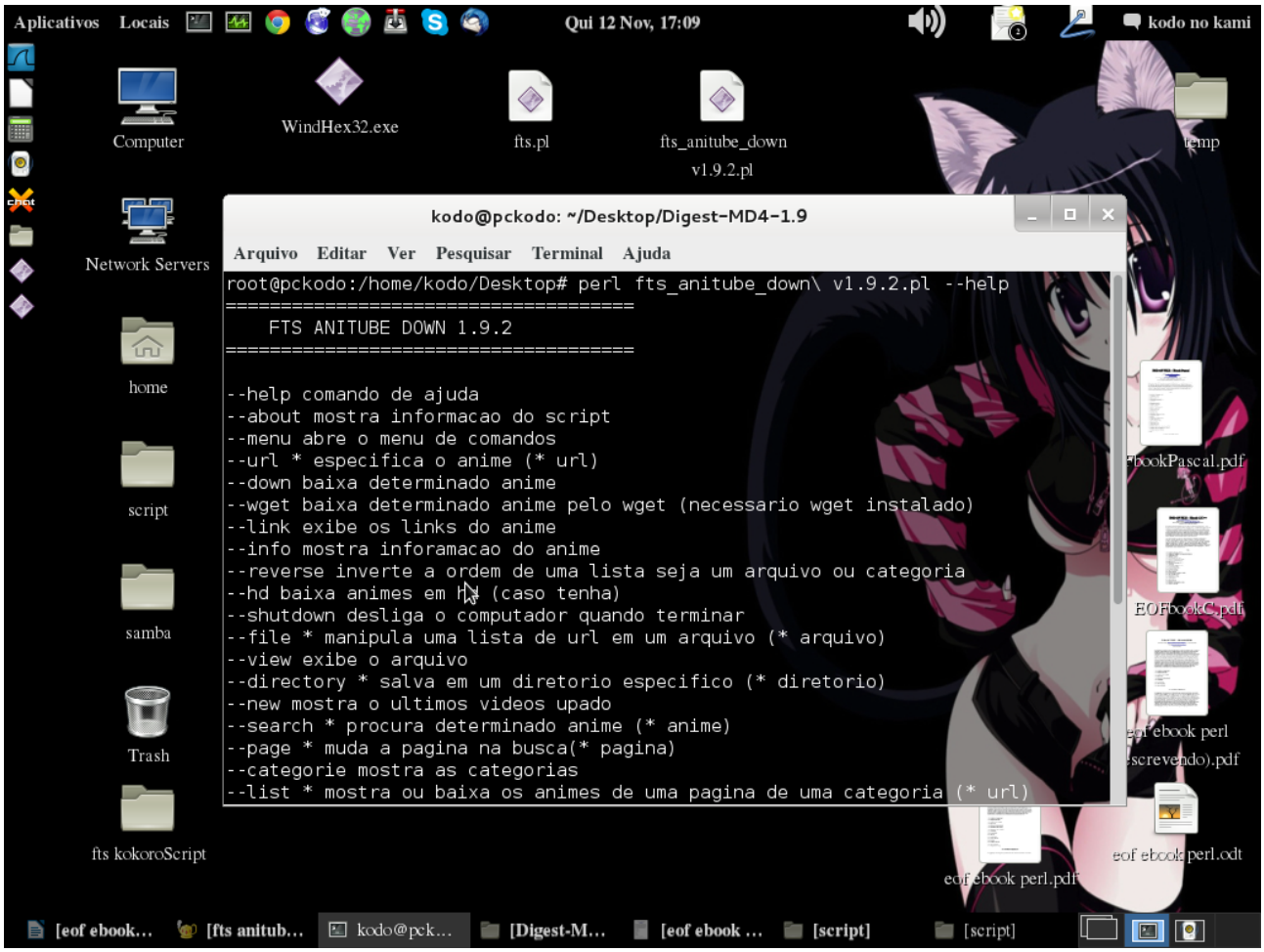
```
perl “kodo no kami.pl”
```

dependendo do script podemos passar argumentos diretamente pelo terminal para ser manipulado no script e esse valores deve ser passado depois do nome do script, nesse exemplo eu passo dois valores o numero 300 e o numero 15 para o script flavio.pl

```
perl flavio.pl 300 15
```

um dos meus scripts perl que eu tenho mais orgulho e o que eu mais uso é o “fts anitube down”, eu tenho brincado com ele deis de 2013 ele já esta na versão 1.9 (1.9.2 embora ainda não divulguei essa versão ainda XD), ele serve para baixar animes no site anitube como também verifica os últimos animes upados, busca por nomes ou ate categoria, mostra informações dos animes, baixa um animes ou uma lista salva em um txt ou uma categoria completa, e ainda é possível usar argumentos ou digitar os comandos em um menu, site para baixar ele caso tenha interesse

```
http://fts315.xp3.biz/ftsanitubedown/ site oficial
https://gist.github.com/hackerfts315/b0aa32f9ce19d1398297 github
```



1.3 – Sintaxe Perl

diferente das linguagens como C, Pascal ou Java que temos que declarar uma função principal da onde é possível chamar as outras funções do nosso programa, no perl não existe uma função principal e sim um pacote que é o main embora o uso dele é opcional, mesmo não declarando ele estamos escrevendo nosso código nele

```
package main

SEU CODIGO
SEU CODIGO
SEU CODIGO
```

também podemos usar a estrutura INIT que especifica os primeiros códigos que sera executados mesmo que essa estrutura seja criada depois do nosso código ela sempre sera executada antes de tudo

```
INIT {
```

```
SEU CODIGO
SEU CODIGO
SEU CODIGO
}
```

além da estrutura INIT exite o END que é o oposto só é executado esse trecho depois que tudo for terminado no script

```
END{
SEU CODIGO
SEU CODIGO
SEU CODIGO
}
```

para que um script perl ou outro seja possível ser usado como CGI deve ser especificado dentro dele na primeira linha onde esta o interpretador da linguagem, para isso usamos #! seguido do endereço do interpretador isso é chamado de shebang, o interpretador no linux na maioria das vezes fica no diretório /usr/bin/, e com isso ficando assim na primeira linha do script

```
#!/usr/bin/perl
```

no windows quando você instala o active state costuma ir para o diretório c:\perl\bin

```
#!c:\perl\bin\perl.exe
```

maioria dos script em perl ou em outra linguagem estão especificado o local onde esta o interpretador mesmo não sendo usado para CGI, isso é opcional porem é uma boa pratica ate mesmo para diferenciar para as outras pessoas qual linguagem foi codado aquele script, então sempre use ele mesmo que seja opcional faz um diferença tremenda

```
#!/usr/bin/perl

SEU CODIGO
```

na linguagem perl e muitas outras não é especificado como quebra de linha o fim de uma função e sim como ponto e virgula que deve ser usado no final de uma função ou determinado procedimento indicando o final dele

```
#!/usr/bin/perl

SEU CODIGO;
SEU CODIGO;
SEU CODIGO;
```

algumas estruturas especificas pode conter uma sequencia de códigos internos como por exemplo as funções e estruturas condicionas, essa estrutura interna é separada por abre e fecha chaves que são chamadas de escopo ou bloco e o que estiver dentro do escopo pertence exclusivamente a ele

```
#!/usr/bin/perl

sub exemplo
{
SEU CODIGO;
}
```

as funções chamadas de funçoes na linguagem perl são o nome da função seguido de abre e fecha parenteses e entre ele os argumentos para dentro da função separados por virgula

```
#!/usr/bin/perl

exemplo(300,15);
```

a linguagem perl também não reconhece espaço e nem quebra de linha ou seja tanto faz você escrever seu código tudo junto e dando espaços grandes que para o interpretador daria no mesmo que esta um em baixo do outro organizado

```
#!/usr/bin/perl

SEU CODIGO; SEU CODIGO;                    SEU CODIGO;

SEU CODIGO;
```

jamais deixe seu código inlegível como no exemplo anterior, uma boa pratica é sempre quando determinada parte do código terminar quebre uma linha e sempre após cada escopo de alguns espaços para saber a qual escopo ele pertence (uma dica é usar a tecla TAB para da um espaço a cada escopo)

```
#!/usr/bin/perl

INIT
{
    SEU CODIGO;
    SEU CODIGO;
    SEU CODIGO;
}
```

1.4 – Biblioteca da linguagem perl

não é necessário criar todas funções já que muitas delas já estão prontas por terceiros e algumas delas já acompanha o interpretador como padrão outras pode ser baixada em repositórios tipo o CPAN ou do próprio linux, essas funções fica em arquivos externos chamados de bibliotecas ou módulos e esses módulos são arquivos que contem varias funções prontas para você usar, você também pode criar suas próprias biblioteca e distribuir para seus amigos ou para uma comunidade, para usar uma biblioteca externa na linguagem perl usamos a palavra “use” seguido do nome dela

```
#!/usr/bin/perl

use warnings;
use strict;
```

as bibliotecas deve esta dentro da mesma pasta do script ou em uma variável de ambiente dele, no linux costuma ficar na pasta “/usr/share/perl” seguido de uma pasta que é versão, já no windows costuma ficar na pasta “c:\perl\lib”, e os módulos também tem a extensão .pm diferente dos scripts que são .pl, quando o modulo esta dentro de um subdiretório basta usar dois pontos para separar cada diretório, exemplo Data/Dumper.pm

```
#!/usr/bin/perl

use Data::Dumper;
```

muitas vezes queremos usar uma função que esta em um modulo porem temos que especificar o caminho todo por exemplo a função md5_hex do modulo Digest::MD5, para usar essa função teríamos que usar Digest::MD5::md5_hex ou então podemos especificar a função entre aspas junto com o declaração do modulo

```
#!/usr/bin/perl

use Digest::MD5 "md5_hex";
```

se tivesse mais de uma função bastaria separar por virgula

```
#!/usr/bin/perl

use Digest::MD5 "md5_hex","md5_base64";
```

1.5 – Instalando modulos em perl

o perl tem vários tipos de repositórios de módulos que a gente poderia baixar ou ate enviar nossos módulos para eles, um desses é o cpan que contem milhares de módulos para linguagem perl, nesse repositório podemos baixar módulos novos com isso aumentando a nossa possibilidade de desenvolvimento ou agilizando ela já que não precisamos desenvolver aquilo, para instalar módulos nos repositórios cpan podemos ir no site de repositório cpan

http://search.cpan.org/
https://metacpan.org/

para baixar um desses módulos basta escolher o modulo desejado e baixar ele no caso o escolhido aqui foi Digest::MD4 (lembrando que alguns desses módulos são específicos para determinada plataforma ou precisa de alguma lib ou ate programa instalado para funcionar e alguns desses módulos precisa de outros módulos anteriores que são chamados de dependência pois funciona por eles ou usa alguma função dela)

```
http://search.cpan.org/~mikem/Digest-MD4-1.9/MD4.pm
```

na lateral do site search cpan na pagina do modulo desejado tem um botao de download onde baixamos o tar.gz do modulo, la tambem mostra as dependencias do modulo e no próprio site tambem mostra informações do uso do modulo, no caso basta a gente baixar o modulo no link do tar.gz, no modulo Digest::MD4 seria o mesmo que ir nesse link

```
http://search.cpan.org/CPAN/authors/id/M/MI/MIKEM/DigestMD4/Digest-MD4-1.9.tar.gz
```

no meu caso eu salvei o modulo no desktop mesmo, depois de baixar abra o prompt ou terminal e vá para onde você salvo o modulo no meu caso desktop

```
cd /home/kodo/Desktop
```

depois extrai ele com o comando tar -xvzf seguido do nome dele (se você usa windows pode extrair com o winrar mesmo)

```
tar -xzvf Digest-MD4-1.9.tar.gz
```

entre na pasta extraída dele com terminal ou prompt

```
cd  Digest-MD4-1.9
```

Agora executamos o script MakeFile para gerar o arquivo make

```
perl MakeFile.pl
```

digitamos o comando make no linux para compilar os módulos, no sistema windows usamos dmake ao invés de make

```
make
```

e por fim instalamos o modulo já compilado com o make install (dmake install)

```
make install
```

antes da gente usar make install (dmake install), podemos também testar com make test (dmake test) para ver se retorno algum erro durante a compilação

```
make test
```

outra forma mais fácil de instalar módulos é com uma ferramenta automática chamada cpan ela além de instalar, ela busca, remove e atualiza os módulos, para usar ela basta abrir o prompt ou terminal e digitar cpan

```
cpan
```

outra forma de chamar essa ferramenta é pelo interpretador perl usamos o argumento -MCPAN -e “shell”

```
perl -MCPAN -e “shell”
```

pela ferramenta CPAN para buscar um mudulo especifico podemos apenas digita a letra i seguido do modulo e apertar enter para buscar com isso vai mostrar informações do modulo como criador, versão e local

```
cpan[1]> i Digest::MD6
```

podemos instalar os módulos usando install seguido do nome do modulo, viu muito mais rápido que baixar e compilar

```
cpan[2]> install Digest::MD6
```

para buscar um modulo com base em um nome usamos uma regex (relaxa vamos aprender mais sobre elas), para usar uma regex usamos a letra i seguindo da palavra porem ela tem que esta entre barras (fiz de zueira mais realmente acho um modulo do redtube, FlashVideo::Site::Redtube agora estou curioso para saber o que faz e como usar ele '-')

```
cpan[3]> i /redtube/
```

podemos instalar manualmente baixando o modulo com get, usamos make para compilar e por fim install para instalar

```
cpan[4]> get Digest::MD2
cpan[5]> make Digest::MD2
cpan[6]> install Digest::MD2
```

para sair digitamos exit

```
cpan[7]> exit
```

o active state ainda tem uma ferramenta para instalação de modulo chamada ppm que é grafica, para usar ela basta digitar ppm no terminal para abrir ela e depois escolher qual pacote que vai instalar

```
ppm
```

no linux muitas distros também tem em seus repositórios os módulos perl onde é possível baixar diretamente por eles usando o gerenciador de pacote da distro (apt-get, pkg, packman, yum, etc)

```
apt-get install libwww-curl-perl
```

1.6 – Informações sobre os modulos

as vezes vem junto com o interpretador perl uma ferramenta chamada perldoc ela permite ver informação de uso dos modulos, para usar ela basta digitar no terminal perldoc seguido do modulo

```
perldoc Data::Dumper
```

também podemos usar o perldoc para ver informações das funções principais da linguagem bastando usar o argumento -f seguido do nome dela

```
perldoc -f print
```

com argumento -v podemos ver as variáveis padrões da linguagem

```
perldoc -v ARGV
```

também podemos ver outras informações da linguagem usando -q

```
perldoc -q number
```

na ferramenta cpan também é possível ver informações de uso dos módulos usado o comando perldoc seguido do módulos

```
cpan[1]> perldoc Digest::MD4
```

as vezes ate pelo man do linux é possível ver algumas informações de uso do modulo perl

```
man Digest::MD5
```

1.7 - Comentários

como qualquer outra linguagem podemos comentar o nosso código para facilitar futuras modificações ou ate mesmo para orientar ou informar uma terceira pessoa sobre ele, para comentar na linguagem perl usamos o sharps e qualquer coisa depois dele vai ser o comentário

```
#!/usr/bin/perl

#isso e um comentário
#fim do codigo
```

podemos comentar depois de um código, isso é uma boa pratica para dizer o que o código faz

```
#!/usr/bin/perl

CODIGO; #esse é meu codigo
CODIGO; #esse é outro codigo
```

podemos usar o comentário para excluir um código temporariamente

```
#!/usr/bin/perl

#CODIGO;
```

também podemos usar comentários para especificar informações como quem é o autor do código, e-mail do autor, site etc

```
#!/usr/bin/perl
#coder: kodo no kami
#forum: eofcommunity.com/forum
```

na linguagem perl podemos usar comentário de varias linhas ou rótulos para isso basta colocar igual seguido de algum texto, e para finalizar usamos =cut com isso o que estiver entre eles sera comentários

```
#!/usr/bin/perl

=info
autor: kodo no kami
data: 06/11/2015
fórum: eofcommunity.com/forum
=cut
```

também podemos especificar o fim do codigo com __END__ e o que estiver depois dele sera comentário, ele é muito usado para documentar o código,

```
#!/usr/bin/perl

__END__
isso aqui é um comentário do
final do codigo
```

2.0 – Saida de dados

bom galera saída de dados ou output é qualquer coisa que sai de ponto para outro no caso ponto que estamos definindo aqui é o programa então saida de dados é nada mais nada menos que alguma coisa que sai do programa podendo ser uma saida do programa para o monitor, para outro computador, para um arquivo, a saída mais simples do perl é a função print que mostra algo na tela, para usar a função print basta usar ela seguido do que a gente quer mostrar na tela do terminal podendo ser uma string, uma variável ou qualquer outra coisa, no caso de uma string que é nada mais nada menos que um texto ele sempre deve esta entre aspas

```
#!/usr/bin/perl

print("ola mundo");
```

podemos o print quantas vezes a gente quiser

```
#!/usr/bin/perl

print("ola mundo");
print("by kodo no kami");
```

porem se a gente percebe no exemplo anterior saiu tudo na mesma linha, para da uma quebra de

linha podemos usar a formatação de escape \n que quebra uma linha, esse \n no linux não é nada mais nada menos que o código hexadecimal 0xa e no window ele é o 0xd e 0xa

```
#!/usr/bin/perl

print("ola mundo\n");
print("by kodo no kami\n");
```

podemos dar varias quebras de linhas em qualquer parte da string

```
#!/usr/bin/perl

print("ola\nmundo\n\nby\nkodo no kami\n");
```

além da formatação de escape \n existem outras como a \t que seria a tabulação

```
#!/usr/bin/perl

print("seu nome: \t flavio \n");
```

temos um sinal sonoro com a formatação de escape \a

```
#!/usr/bin/perl

print("beep \a \n");
```

podemos exibir aspas usando \"

```
#!/usr/bin/perl

print("programacao em \"perl\" e muito facil \n");
```

para usar contra barra sem interpretar como formatação de escape usamos \\

```
#!/usr/bin/perl

print("nunca remova o diretorio c:\\windows\\system32. ele nao e virus  \n");
```

strings também pode ser colocado com aspas simples, a diferença que o uso de aspas simples não reconhece determinadas formatação de escape

```
#!/usr/bin/perl

print('perl = camelo \npython = cobra \njava = cafe \ne o \'assembly\' e o que?');
```

também podemos juntar duas ou mais strings com o ponto isso é chamado de concatenação

```
#!/usr/bin/perl

print("forum: " . "eofcommunity.com/forum");
```

ou podemos usar o print para mostrar o conteudo de uma variável

```
#!/usr/bin/perl

$variavel = "eofcommunity\n";
print($variavel);
```

podemos usar o print para mostrar um tipo numérico, no caso diferente das strings tipos numéricos não fica entre aspas embora para a linguagem perl isso não faz tanta diferença

```
#!/usr/bin/perl

print(315);
```

além do print temos o printf que parecido com anterior porem ele permite formata a saída com base na entrada dos argumentos, por exemplo podemos mostrar uma string formatada junto a um tipo numérico que usa como base decimal para hexadecimal isso dentro da string, o uso dele deve ser especificado o primeiro argumento de como deve ser formatada a string e os outros sera as entradas para string, ele também usa alguns escapes para formatar os argumentos que são %d para decimal ou números inteiros, %f para números quebrados com pontos, %x para números hexadecimais, %o para numeros octais, %b para números binários e %c para caracteres

```
#!/usr/bin/perl

$variavel = 97;
printf("o numero %d (decimal) \n",$variavel);
printf("o numero %f (float) \n",$variavel);
printf("o numero %x (hexadecimal) \n",$variavel);
printf("o numero %o (octal) \n",$variavel);
printf("o numero %b (binario) \n",$variavel);
printf("o numero %c (caracter) \n",$variavel);
```

outra função é o sprintf que é bem parecida com printf porem ao invés de sair para o terminal ele armazena a saida em uma variável

```
#!/usr/bin/perl

$variavel = 97;
$saida = sprintf("numero %d em hexadecimal e igual a %x \n",$variavel,$variavel);
print($saida);
```

nas versões mais novas do perl existe o say que é parecido com print porem da uma quebra de linha automático no final, para usar o say temos que especificar um modulo que seria uma das versões mais recente do perl que tenha ele, exemplo a versão 5.10.0

```
#!/usr/bin/perl

use 5.10.0;

say("eu estou programando em perl");
say("e não em python XD");
```

também poderia colocar a versao atual do meu intepretador ou uma anterior a ela que no meu caso é a versão atual é 5.14.2

```
#!/usr/bin/perl

use 5.14.2;

say("eu estou programando em perl");
say("e não em php XD");
```

ou também pode ser declarado dessa outra forma

```
#!/usr/bin/perl

use v5.14;

say("eu estou programando em perl");
say("e não em ruby XD");
```

2.1 – Variáveis

variáveis são alocações de memoria que permite a gente armazenar determinados tipos de dados que podemos acessá-los ao decorrer do programa e também modifica-los, as variáveis podem ter qualquer nome porem não deve começar com números e nem ter espaços ou caracteres especiais, na linguagem perl as variáveis são fracamente tipadas ou seja você não é obrigado a declarar uma variável antes de usa-la e ao decorrer do programa a mesma variável pode armazenar outro tipo dado sem precisar realocar espaço para tipos diferentes, as variáveis na linguagem perl também são

genéricas ou seja uma variável que armazena uma string pode ser usada como uma variável do tipo numérica ou vice versa, em perl também existe pelo menos 3 tipos diferentes de alocação ou de variáveis que é a scalar que é variável normal que armazena apenas um valor ou um tipo de dado por vez, as arrays que é uma variável com vários segmentos podendo guarda vários dados em varias posições na mesma variável e ao mesmo tempo onde pode ser acessado por um índice numérico especifico e por fim as hash que são tipo as arrays porem o índice dela é uma string ao invés de um numero, para a gente armazena um valor em uma variável do tipo scalar basta coloca o nome da variável porem ele deve começar com cifrão indicando uma variável scalar depois usamos o sinal de igualdade que indica que vamos atribuir um valor para ela e por fim o valor que vamos atribuir

```
#!/usr/bin/perl

$numero = 315;
$nome = "kodo no kami";
```

como dito antes variáveis não pode ter espaço e nem começar com numero então as variáveis abaixo estão errada e o interpretador vai retornar erro

```
#!/usr/bin/perl

$1numero = 315;
$nome ou nick = "kodo no kami";
```

quando precisar usar espaço no nome de uma variável ou função use underline no lugar do espaço

```
#!/usr/bin/perl

$nome_ou_nick = "kodo no kami";
```

para acessar os dados em uma variável scala basta especificar ela onde vamos usar

```
#!/usr/bin/perl

$nome = "kodo no kami";
print($nome);
```

podemos concatenar a nossa variável com uma string

```
#!/usr/bin/perl

$nome = "kodo no kami";
print("seu nome e " . $nome);
```

no perl também é permitido usar uma variável diretamente dentro um string

```
#!/usr/bin/perl

$nome = "kodo no kami";
print("seu nome e $nome \n");
```

e como já disse antes uma variável pode mudar o valor ao decorrer do programa

```
#!/usr/bin/perl

$nome = "hfts315\n";
print($nome);

$nome = "kodo\n";
print($nome);
```

você também pode armazenar o conteúdo de uma variável em outra variável bastando atribuir ela para a outra

```
#!/usr/bin/perl

$nome = "kodo no kami\n";
$nome2 = $nome;

print($nome2);
```

ou ate atribuir uma variável nela mesma

```
#!/usr/bin/perl

$nome = "kodo no kami\n";
$nome = $nome;

print($nome);
```

é possível concatenar duas ou mais variáveis em uma única

```
#!/usr/bin/perl

$nome = "flavio\n";
$nick = "kodo\n";
$junto =  $nome . $nick;

print($junto);
```

é possível inserir vários dados em varias variáveis simultaneamente bastando colocá-los entre parênteses e separar eles por virgula, no caso só vai armazenar a quantidade de variáveis disponíveis caso tenha mais dados que variáveis

```
#!/usr/bin/perl

($nome, $nick, $idade) = ("flavio","kodo no kami",23);
```

2.2 – Arrays e Hashs

além das variáveis scalar temos as arrays ou list (lista) que é uma variável que armazena vários dados ao mesmo tempo em posições diferente na mesma variável, diferente da variável scalar o simbolo da array é o arroba porem isso quando vamos tratar ela como um todo no caso para acessar ou armazenar um valor dentro dela a gente usa o simbolo de cifrão e uma posição numérica entre colchetes e essa posição numérica começa no numero zero, e a manipulação dela é mesma coisa da scalar

```
#!/usr/bin/perl

$adm[0] = "kodo no kami";
$adm[1] = "51m0n";
$adm[2] = "sir.rafiki";
$adm[3] = "mmxm";

print("$adm[0], $adm[1], $adm[2], $adm[3] \n");
```

se a gente exibir em um print uma array sem a posição e usando o arroba ele vai mostrar todo conteúdo concatenado daquela array

```
#!/usr/bin/perl

$adm[0] = "kodo no kami";
$adm[1] = "51m0n";
$adm[2] = "sir.rafiki";
$adm[3] = "mmxm";

print(@adm);
```

se atribuir uma array para uma scalar ela armazena a quantidade de posições que tem aquela array

```
#!/usr/bin/perl

$adm[0] = "kodo no kami";
$adm[1] = "51m0n";
$adm[2] = "sir.rafiki";
```

```
$adm[3] = "mmxm";

$tamanho = @adm;

print($tamanho);
```

por outro lado se a gente atribui uma array para outra array ela simplesmente copia o conteúdo de uma para a outra

```
#!/usr/bin/perl

$adm[0] = "kodo no kami";
$adm[1] = "51m0n";
$adm[2] = "sir.rafiki";
$adm[3] = "mmxm";

@eofcommunity = @adm;

print(@eofcommunity);
```

da mesma forma que podemos usar vários dados para armazenar em varias scalar ao mesmo tempo podemos usar vários dados para armazenar em uma única array em varias posições diferente simultaneamente

```
#!/usr/bin/perl

@adm = ("kodo no kami", "51m0n", "sir.rafiki", "mmxm");
```

também podemos juntar duas ou mais arrays em uma única array (no lugar da array manos eu ia usar manada mais ai lembrei que era o coletivo de boi então deixa quieto kkkkk)

```
#!/usr/bin/perl

@adm = ("kodo no kami", "51m0n", "sir.rafiki", "mmxm");
@manos = ("lendapop","gjuniioor","nass","oracle","neofito");

@eof = (@adm, @manos);
```

da mesma forma podemos atribuir separadamente em varias scalar uma array

```
#!/usr/bin/perl

@adm = ("kodo no kami", "51m0n", "sir.rafiki", "mmxm");
($pri, $seg, $ter, $qua) = @adm;
```

se a gente usar variáveis scalar e uma array para atribuir para uma array ele vai armazenar alguns dados nas variaveis scalar e o restante na array;

```
#!/usr/bin/perl

@adm = ("kodo no kami", "51m0n", "sir.rafiki", "mmxm");
($autor, @resto) = @adm;
```

para armazenar valores em mais de uma array fazemos outros parênteses separado por virgula

```
#!/usr/bin/perl

(@adm, @adm2) = ("kodo no kami", "51m0n"),("sir.rafiki", "mmxm");
```

podemos inserir um dado no final de uma array usando a função push, para usar ela basta usar o push seguido da array que vamos manipular seguido do valor que vamos inserir

```
#!/usr/bin/perl

@linguagem = ("c", "pascal");
push(@linguagem,"perl");

print($linguagem[2]);
```

isso também pode ser feito no começo de uma array com a função unshift sendo os parâmetros dela igual o push

```
#!/usr/bin/perl

@linguagem = ("c", "pascal");
unshift(@linguagem,"perl");

print($linguagem[0]);
```

da mesma forma podemos usar o pop para remover e retornar o ultimo valor da array

```
#!/usr/bin/perl

@linguagem = ("c", "pascal", "perl");
$ultimo = pop(@linguagem);

print($ultimo);
```

também podemos usar o shift para remover e retornar o primeiro valor da array

```
#!/usr/bin/perl

@linguagem = ("c", "pascal", "perl");
$primeiro = shift(@linguagem);

print($primeiro);
```

com a função sort podemos organizar em ordem alfabética uma array porem ela não modifica a array atual então temos que atribuir para outra array

```
#!/usr/bin/perl

@paises = ("brasil","japao","china","russia","australia");
@ordem = sort(@paises);
```

também existe a função reverse que inverte a ordem de uma array e o uso dela é parecido com o sort

```
#!/usr/bin/perl

@paises = ("brasil","japao","china","russia","australia");
@ordem = reverse(@paises);
```

além das variáveis do tipo array temos outro tipo de variável que são as hash, as variáveis hash são como as arrays porem ao invés de usarmos índice numérico usamos keys (chaves) que são nomes para cada segmento especifico dela (vou usar o termo nome para facilitar e não confundir com chaves mesmo), quando tratamos de uma hash como um todo usamos o simbolo de porcentagem porem quando tratamos uma única posição dela usamos o cifrão como as scalar e as arrays porem ao invés de usarmos colchetes para especificar o índice como na array usamos chaves seguido do nome

```
#!/usr/bin/perl

$info{nome} = "flavio";
$info{nick} = "kodo no kami";
$info{idade} = "23";
$info{habilidade} = "amnesia alcoólica XD";

print("$info{nome}, $info{nick}, $info{idade}, $info{habilidade} \n ");
```

as variáveis hash são nada mais nada menos que as arrays a diferença entre elas que as arrays armazena apenas os dados nas posições ja as hash armazena o nome em uma posição e na próxima posição o dado e assim por diante, com isso é possível transformar uma variável array em uma variável hash bastando em uma posição armazenar o nome e na outra o dado assim sucessivamente

```
#!/usr/bin/perl

@linguagem  = ("c","denis ritchie","pascal","nicklaus wirth","perl","larry wall");
%criador = @linguagem;

print($criador{perl});
```

podemos armazenar vários dados simultaneamente em uma hash como fazemos com uma array porem temos que armazenar em ordem nome e valor como no exemplo anterior

```
#!/usr/bin/perl

%linguagem  = ("c","denis ritchie","pascal","nicklaus wirth","perl","larry wall");

print($linguagem{pascal});
```

além dessa forma tem uma outra para especificar o nome e o dado bastando usar o nome sinal de igual e sinal de maior que seguido do dado

```
#!/usr/bin/perl

%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");

print($linguagem{c});
```

a gente tambem pode armazenar em uma array usando aquele método anterior porem seria o mesmo que armazenar dois dados ao mesmo tempo

```
#!/usr/bin/perl

@linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");

print(@linguagem);
```

como também podemos fazer o inverso pegar uma hash e armazenar em uma array porem vai salvar os nomes juntos e não apenas os dados, uma curiosidade que quando você salva de uma hash para uma array é armazenado invertidamente como se tivesse usado reverse nela antes

```
#!/usr/bin/perl

%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");
@kodo = %linguagem;

print(@kodo);
```

usar a função reverse permite inverter não só a ordem mais sim os nomes pelo dados

```
#!/usr/bin/perl

%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");
%fts = reverse(%linguagem);

print($fts{"nicklaus wirth"});
```

seguindo essa logica o uso normal do sort não vai funcionar por que vai colocar em ordem alfabética embaralhando os nomes e os dados

```
#!/usr/bin/perl

%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");
%fts = sort(%linguagem);

print(%fts);
```

podemos manipular apenas os nomes com a função keys

```
#!/usr/bin/perl

%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");
@nomes = keys(%linguagem);

print(@nomes);
```

com a função values fazemos o mesmo que o anterior porem apenas com os dados

```
#!/usr/bin/perl

%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");
@valores = values(%linguagem);

print(@valores );
```

podemos exibir todos os dados com a função Dumper porem para usar ela temos que declarar o módulo Data::Dumper, ele pode ser usada tanto para array quanto hash, para usar ela basta passar a array como argumento e depois ler o retorno ou armazenar em uma variavel

```
#!/usr/bin/perl

use Data::Dumper;
```

```
%linguagem  = (c => "denis ritchie", pascal => "nicklaus wirth", perl => "larry wall");

print(Dumper(%linguagem));
```

2.3 – Vetores e Matrizes

as arrays anteriores são chamadas de vetores isso por que elas tem apenas um único segmento onde podemos armazenar os dados porem existem arrays multidimensionais que são chamado de matrizes e a cada segmento dela guarda outro segmento ou seja quando você acessa uma posição ao invés de acessar os dados estaria acessando outra posição e é nessa outra posição que tem os dados armazenados e não na anterior, o uso de matriz é mais adequado para armazenar grande quantidade dados que precise segmentar em pouca quantidade de posições por exemplo se você tivesse que armazenar apenas o nome e a idade precisaria de uma array de apenas duas posições porem tivesse que armazenar 1000 dessas informações poderia usar uma array normal no caso um vetor porem ficaria mais organizado dentro de uma matriz, para armazenar dados como matrizes usamos o colchete e dentro desse colchete colocamos os dados separado por virgula sendo cada colchete um segmento separado e os dados no mesmo colchete são do mesmo segmento ou posição

```
#!/usr/bin/perl

@sis = (["debian","centos","kali"]);
```

para acessar os dados usamos dois índices o primeira acessa o primeiro segmento e o segundo a posição dos dados, sempre os dados vai estar no ultimo segmento e o anteriores serve apenas para alterna entre um segmento e outro

```
#!/usr/bin/perl

@sis = (["debian","centos","kali"]);

print("linux: $sis[0][0], $sis[0][1], $sis[0][2]\n");
```

ou tambem podemos atribuir diretamente na posição

```
#!/usr/bin/perl

$sis[0][0] = "debian";
$sis[0][1] = "centos";
$sis[0][2] = "kali";
```

podemos criar outros segmentos bastando colocar outro colchete separando eles por virgula e o acesso a eles seria pela primeira posição porem o acesso aos dados dados dele sempre é a ultima posição

```
#!/usr/bin/perl

@sis = (["debian","centos","kali"], ["win 7","win 10"]);

print("linux: $sis[0][0], $sis[0][1], $sis[0][2]\n");
print("windows: $sis[1][0], $sis[1][1] \n");
```

você também pode criar novos segmentos dentro do outro segmento assim seria novos sub-segmentos

```
#!/usr/bin/perl

@sis = ([["debian","centos","kali"]] );

print("linux: $sis[0][0][0], $sis[0][0][1], $sis[0][0][2]\n");
```

porem não vamos exagerar dessa forma né, embora isso ainda tenha utilidade cientifica ou manipulação de numeros muito grandes '-'

```
#!/usr/bin/perl

@sis = ([[[[[[[[[["android"]]]]]]]]]]);

print("linux: $sis[0][0][0][0][0][0][0][0][0][0] \n");
```

da mesma forma que o array permite uso de vetores e matrizes as hash também permite porem ela é um pouco diferente ao invés de usarmos posições numéricas usamos nomes para cada índice e os separamos com chaves e não colchetes, para criar uma hash matriz atribuímos um nome normalmente que seria aquela posição porem no lugar dos dados usamos chaves e dentro delas que seria os dados armazenados separados por virgula

```
#!/usr/bin/perl

%linguagem  = (compilada => {c => "denis ritchie",  pascal => "nicklaus wirth"});
```

para acessar os dados temos que especificar o nome da primeira posição seguido do sinal hífen e o sinal de maior que seguido do nome da posição do dado

```
#!/usr/bin/perl

%linguagem  = (compilada => {c => "denis ritchie"});

print("$linguagem{compilada}->{c}, $linguagem{compilada}->{pascal} \n");
```

e da mesma forma também podemos atribuir diretamente as posições

```
#!/usr/bin/perl

$linguagem{compilado}->{c} =  "denis ritchie";
$linguagem{compilado}->{pascal} = "nicklaus wirth";
```

como as matrizes nas arrays usamos o virgula para separar cada segmento também

```
#!/usr/bin/perl

%linguagem  = (compilada => {c => "denis ritchie"}, script => { perl => "larry wall"});

print("$linguagem{compilada}->{c} ,  $linguagem{script}->{perl} \n");
```

as hashs também é possível criar novos segmentos dentro do outro como nas matrizes das arrays

```
#!/usr/bin/perl

%linguagem  = (compilada => {pascal => { ide => "delphi"}});

print("$linguagem{compilada}->{pascal}->{ide}  \n");
```

2.4 – Ponteiros

todas as variáveis como todas as arrays ou hashs tem um endereço de memoria que é ligado ao nome da variável então quando você usa essa variável você esta puxando ou armazenando aqueles dados em um determinado endereço de memoria, na linguagem perl é possível manipular esse endereço e isso chamado de ponteiro e para manipular eles basta atribuir o endereço de memoria para uma variável sendo essa variável um ponteiro para aquele endereço, para pegar o endereço de memoria de uma variável ou array basta atribuir ela para outra variável porem colocando contra barra na frente dela isso indica que você esta passando o endereço de memoria dela e não o dado armazenado nela

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;
```

para acessar os dados naquela ponteiro usamos duas vezes o cifrão com isso a gente estaria acessando a primeira variável que pegamos o endereço

```
#/usr/bin/perl

$variavel = "eof community";
```

```
$ponteiro = \$variavel;

print($$ponteiro);
```

a gente também poderia esta atribuindo valores para o endereço no ponteiro assim estaria armazenado diretamente na variável e não no ponteiro

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;
$$ponteiro = "kodo no kami";

print($variavel);
```

se a gente usar apenas um cifrão a gente estaria acessando apenas o dado armazenado no ponteiro que é o endereço da variável que armazenamos e não os dados da variável que o ponteiro esta apontando

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;

print($ponteiro);
```

a utilidade de acessar o endereço em um ponteiro é que podemos copiar o mesmo endereço para outras variáveis ou seja essas outras variáveis também seria um ponteiro para aquela primeira variável

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;
$outro_ponteiro = $ponteiro;

print($$outro_ponteiro);
```

também podemos armazenar o endereço de memoria do ponteiro anterior em um novo ponteiro e isso é chamado de ponteiro de ponteiro, porem quando vamos ler os dados armazenado nele a gente acaba caindo no endereço da variável já que é ela que esta armazenado no ponteiro anterior e não os dados da variável

```
#/usr/bin/perl
```

```
$variavel = "eof community";
$ponteiro = \$variavel;
$outro_ponteiro = \$ponteiro;

print($$outro_ponteiro);
```

para acessar os dados em um ponteiro de ponteiro usamos mais um cifrão

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;
$outro_ponteiro = \$ponteiro;

print($$$outro_ponteiro);
```

a cada novo ponteiro usamos um novo cifrão com isso estaríamos acessando um endereço pelo o outro

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;
$outro_ponteiro = \$ponteiro;
$ponteiro_recente = \$outro_ponteiro;
$apontando = \$ponteiro_recente;

print($$$$$apontando);
```

também é possível pegar o endereço dentro de um ponteiro fazendo o ultimo ponteiro apontar para onde o ponteiro anterior esta apontando, para fazer isso basta a gente atribuir ele com mais um cifrão

```
#/usr/bin/perl

$variavel = "eof community";
$ponteiro = \$variavel;
$outro_ponteiro = \$$ponteiro;

print($$outro_ponteiro);
```

o mesmo pode ser feito com mais de ponteiro bastando atribuir eles usando mais cifrão

```
#/usr/bin/perl
```

```
$variavel = "eof community";
$ponteiro = \$variavel;
$outro_ponteiro = \$ponteiro;
$ponteiro_recente = \$outro_ponteiro;
$apontando = \$$$$ponteiro_recente;

print($$apontando);
```

também é possível criar um ponteiro para uma array bastando atribuir o endereço de uma para uma variável

```
#/usr/bin/perl

@sorvete = ("chocolate","morango");
$ponteiro = \@sorvete;
```

para acessar os dados em um ponteiro para uma array a gente não utiliza dois cifrão e sim um arroba e um cifrão para acessar ela como um todo

```
#/usr/bin/perl

@sorvete = ("chocolate","morango");
$ponteiro = \@sorvete;

print(pop(@$ponteiro));
```

para acessa as posições de uma array por um ponteiro usamos duas vezes o cifrão

```
#/usr/bin/perl

@sorvete = ("chocolate","morango");
$ponteiro = \@sorvete;

print("$$ponteiro[0]");
```

o mesmo vale para uma matriz de array

```
#/usr/bin/perl

@sorvete = (["chocolate"],["morango"]);
$ponteiro = \@sorvete;

print("$$ponteiro[1][0] \n");
```

também podemos atribuir uma hash para um ponteiro da mesma forma que fazemos com as arrays

seja vetores ou matrizes

```
#/usr/bin/perl

%pc = (video => "512mb");
$ponteiro = \%pc;

print("$$ponteiro{video} \n");
```

podemos descobrir o tipo do ponteiro que foi atribuído a variável usando a função ref que retorna o tipo

```
#/usr/bin/perl

@variavel = ("eof community");
$ponteiro = \@variavel;

$tipo = ref($ponteiro);

print($tipo);
```

2.5 – Escopo das variáveis

as variáveis na maior parte das vezes são globais com exceção as que fica dentro de funções e outros escopos, sendo elas globais podemos usar elas ate mesmo dentro de um escopo porem isso também pode acontece conflito entre uma variável dentro de um escopo e a de fora para evitar isso podemos especificar se uma variavel pertence a um escopo se é global ou local declarando ela antes de usar, só lembrando a linguagem perl é fracamente tipada ou seja isso não é um padrão obrigatário na linguagem porem é uma boa pratica para programar nela, para a gente declarar uma variável seja uma scalar, array ou hash que pertence ao escopo usamos a palavra my seguido do nome da variável, essas variáveis vão ser enxergada dentro do próprio escopo e nos escopo que estiver dentro desse mesmo escopo

```
#!/usr/bin/perl

my $nome;
my @nomes;
my %info;

$nome = "kodo no kami";
@nomes = ("flavio","hfts315","kodo no kami");
%info = (nome => "flavio", idade => 23, codename=> "kodo");
```

também podemos atribuir o valor diretamente a ela

```
#!/usr/bin/perl

my $nome = "kodo no kami";
my @nomes = ("flavio","hfts315","kodo no kami");
my %info = (nome => "flavio", idade => 23, codename=> "kodo");
```

é possível declarar elas entre parenteses usando um único my

```
#!/usr/bin/perl

my ($nome , @nome2, %info);

$nome = "kodo no kami" ;
@nomes = ("flavio","hfts315","kodo no kami"); ;
%info = (nome => "flavio", idade => 23, codename=> "kodo");
```

e o mesmo vale para atribuir os valores a ela

```
#!/usr/bin/perl

my ($nome, @nome2, %info) ;

@nomes = ("flavio","hfts315","kodo no kami"); ;
%info = (nome => "flavio", idade => 23, codename=> "kodo");

print($nome);
```

também podemos atribuir os valores diretamente para variável porem na ordem certa como já fizemos antes

```
#!/usr/bin/perl

my ($nome, $idade) = ("kodo",23);
```

além das variáveis my que pertence ao próprio escopo e podem ser enxergada por escopos internos e as vezes externos temos as variáveis locais que seria usada dentro de um escopo e só pode ser enxergada la dentro

```
#!/usr/bin/perl

my $nome = "kodo";
{
    local $idade = 23;
    print($nome);
}
```

```
print($idade);
```

no exemplo anterior a variável $nome foi mostrada normalmente isso por que o escopo interno ali pode enxerga ela porem a variável $idade não pode ser mostrada isso por que é uma variável local que pertence apenas aquele escopo e não pode ser enxergada fora dele, além da variáveis my e local temos as variáveis globais que são as our e ela seria a mais adequada para a variável $nome do exemplo anterior

```
#!/usr/bin/perl

our $nome = "kodo";

{
    print($nome);
}
```

podemos criar escopos para separar variáveis com o mesmo nome sem interferir na outra, para isso basta usar chaves

```
#!/usr/bin/perl

our  $nome = "flavio";
{
    local  $nome = "kodo";
    print($nome); #vai mostrar o kodo
}
{
    local $nome = "fts315";
    print($nome); #vai mostrar o fts315
}
print($nome); #vai mostrar o flavio
```

para especificar se você esta usando essas normas de variáveis globais e locais corretamente entre outras coisas podemos declarar dois módulos que são o strict e warnings que retorna erro caso esteja fora do padrão além de ser uma boa pratica sempre usar eles no seu codigo e a partir desse exemplo sempre vamos usar ele também (se você não usar tem o risco do mmxm fala mal do seu código, é serio isso '-')

```
#!/usr/bin/perl

use strict;
use warnings;
```

2.6 – Entrada de Dados

bom galera o que torna um programa dinâmico são as entradas de dados sem elas nosso programa é estatico, existem diversos tipos de entrada de dados podendo ser uma entrada do teclado, uma entrada de um arquivo, uma entrada de um dispositivo, uma entrada ou qualquer outra coisa que entre para programa, a entrada de dados mais básica que temos para um script são os argumentos passado pelo terminal ou pelo prompt, vamos supor que eu tenha que passar o valor 100 pelo argumento do terminal então basta eu digitar ele após o nome do script no terminal

```
perl kodo.pl 100
```

se eu tivesse que passar o valor 300 e o valor 15 bastaria da um espaço entre eles

```
perl kodo.pl 300 15
```

eu poderia fazer isso quantas vezes for necessário

```
perl kodo.pl 300 15 123456 5 10000
```

também posso passar uma string

```
perl kodo.pl flavio
```

porem se na string ou ate mesmo no nome do script tiver espaço temos que colocar entre aspas

```
perl kodo.pl "kodo no kami"
```

ate agora vimos como passar os valores pelo terminal, porem apenas passar o valor não indica que o script esta manipulado eles, no caso para acessar esses valores basta a gente acessar uma array especifica que é a @ARGV e os argumentos são armazenado na ordem que entraram, com isso se a gente entrar com o valor 315 pelo terminal ele vai para primeira posição que é o indice 0

```
#!/usr/bin/perl

use strict;
use warnings;

print($ARGV[0]);
```

se a gente entrar com dois valores um sendo o nome e o outro sendo a idade o primeiro argumento é o nome "flavio" vai ser armazendo na posição 0 do @ARGV e o segundo que é o 23 vai ser armazenado na posição 1 do @ARGV

```
#!/usr/bin/perl

use strict;
use warnings;

print("seu nome $ARGV[0] \n");
print("sua idade $ARGV[1] \n");
```

podemos saber quantos argumentos entraram bastando atribuir a array para uma variavel scalar com isso ira armazenar a quantidade de posiçoes dela na variavel scalar sendo essa quantidade a mesma de entrada

```
#!/usr/bin/perl

use strict;
use warnings;

my $tamanho = @ARGV;

print("quantidade de argumento $tamanho \n");
```

muitos scripts são usados argumentos para especificar o argumento e o argumento seguinte sendo o valor assim em um script não precisaria entrar com todos os argumentos apenas aqueles que vai ser manipulado sem precisar esta em ordem para manipular tambem

```
perl kodo.pl --idade 23 --nome flavio
```

a forma mais facil da gente criar esse argumento é usar uma hash no lugar da array para isso basta atrituir o @ARGV para uma variavel hash, lembrando que hash e array é a mesmas coisa porem a hash vai ler duas posiçoes na array sendo um o nome e a outra o valor

```
#!/usr/bin/perl

use strict;
use warnings;

my %argu = @ARGV;

print("seu nome: $argu{"--nome"}, sua idade:  $argu{"--idade"} \n");
```

tem um modulo especifico no perl que manipula essa entrada de argumento do terminal sendo ele o Getopt::Long, dentro desse modulo tem a função GetOptions que basta a gente especificar os argumentos que queremos manipular apontando para uma variável especifica e depois basta a gente apenas manipular essas variáveis, para usar o GetOptions passamos como argumento uma string sendo ela nome do argumento que vamos manipular usamos sinal de igual seguido do tipo sendo ele “s” para string e “i” para números também podemos deixar sem o igual apenas para passar um

argumento especifico, depois fazemos igual as hash um igual e o sinal de maior que seguido porem usamos ponteiro para a variável onde vai ser armazenado

```
perl kodo.pl --nome kodo
```

```
#!/usr/bin/perl

use strict;
use warnings;
use Getopt::Long;

my $nom;

GetOptions("nome=s" => \$nom);

print("seu nome: $nom \n");
```

podemos especificar vários argumentos no GetOptions bastando separar eles por virgula

```
perl kodo.pl --nome kodo --idade 23
```

```
#!/usr/bin/perl

use strict;
use warnings;
use Getopt::Long;

my ($nom, $ida);

GetOptions("nome=s" => \$nom , "idade=i" => \$ida);

print("seu nome é $nom e sua idade é $ida \n");
```

tambem podemos usar uma array e passar mais de um argumento

```
perl kodo.pl --nome kodo --nome fts315
```

```
#!/usr/bin/perl

use strict;
use warnings;
use Getopt::Long;

my @nom;
```

```
GetOptions("nome=s" => \@nom );

print("seu nome é $nom[0] e $nom[1] \n");
```

outra forma semelhante de entrar com dados para dentro do programa porem não muito usado é as variáveis de ambiente, as variáveis de ambiente são variáveis do sistema onde os programa podem carrega-las, para acessar uma variável de ambiente usamos a hash %ENV e passamos para ele o nome da variável de ambiente exemplo a variável PATH que armazena os diretórios onde os programas pode ser chamado sem precisar digita o endereço completo deles

```
#!/usr/bin/perl

use strict;
use warnings;

my $pa = $ENV{PATH};

print("seu path: $pa");
```

também podemos setar dados nessas variáveis

```
#!/usr/bin/perl

use strict;
use warnings;

$ENV{kodo} = "kodo no kami";

print("variavel de ambiente kodo = $ENV{kodo}");
```

outra forma da gente ler as variáveis de ambiente é com a função getenv do modulo POSIX, para usar ele passar como argumento o nome da variável

```
#!/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $variavel = getenv("PATH");

print("seu path atual $variavel");
```

também é possível entrar com os dados depois que o script já estiver rodando para isso basta usar dois sinais menor que e maior que, esses dois sinas juntos indica entrada de alguma coisa para o script que pode ser a entrada do teclado de um diretório de um arquivo ou de algum descritor

especifico, os descritores ou handles são tipo ponteiros porem ao invés de apontar para um endereço de memoria especifico aponta para um arquivo ou evento podendo ser ate uma coisa mais especifica como a própria leitura ou escrita naquele arquivo, se a gente usa o <> sem especificar nenhum descritor dentro o script como padrao vai executar o stdin que seria o descritor da entrada do teclado ou seja quando o script chegar nessa parte vai ficar parado esperando o usuário digitar alguma coisa e só vai prosseguir quando apertar ENTER

```
#/usr/bin/perl

use strict;
use warnings;

print ("aperte enter para sair");
<>;
```

ou especificar o descritor stdin ali dentro

```
#/usr/bin/perl

use strict;
use warnings;

print ("aperte enter para sair");
<stdin>;
```

podemos atribuir a entrada do stdin para uma variavel

```
#/usr/bin/perl

use strict;
use warnings;

print ("digite seu nome: ");
my $nome = <stdin>;

print("seu nome é $nome \n");
```

podemos usar ele quantas vezes a gente quiser;

```
#/usr/bin/perl

use strict;
use warnings;

print ("digite seu nome: ");
my $nome = <stdin>;
```

```
print ("digite sua idade: ");
my $idade = <stdin>;

print("seu nome é $nome e você tem $idade anos \n");
```

as vezes quando usamos o stdin pode vim mais coisas junto com o que a gente digito por exemplo o ENTER que a gente aperto com isso ele acaba vindo junto e da uma quebra de linha na string, para evitar isso usamos a função chomp e passamos como argumento para ela a variável

```
#/usr/bin/perl

use strict;
use warnings;

print ("digite seu nome: ");
my $nome = <stdin>;
chomp($nome);

print("seu nome é $nome \n");
```

também podemos usar o chop a diferença deles que o chomp confere o ultimo carácter antes de apagar já o chop não ele sempre apaga o ultimo carácter independente de qual seja ele

```
#/usr/bin/perl

use strict;
use warnings;

print ("digite seu nome: ");
my $nome = <stdin>;
chop($nome);

print("seu nome é $nome \n");
```

podemos usar a função getc para pegar apenas um carácter

```
#/usr/bin/perl

use strict;
use warnings;

print ("digite a primeira letra do seu nome: ");
my $letra = getc();

print("a primeira letra é $letra \n");
```

existe a função getchar do modulo POSIX, ela é igual a anterior só pega um carácter também

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

print ("digite a primeira letra do seu nome: ");
my $letra = getchar();

print("a primeira letra é $letra \n");
```

com a função gets também do modulo POSIX podemos pegar uma sequencia de caracteres e ela tem a mesma funcionalidade do <stdin>

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

print("digite o nome da sua cidade: ");
my $cidade = gets();

print("voce mora em $cidade \n");
```

2.7 – Manipulação de string

como já sabemos as string são os textos que a gente escreve e em outras linguagens as strings são uma sequencia de caracteres ou ate uma array de caracteres podendo ser manipulado como array porem isso não é possível em perl pelo menos não como as arrays, as strings na linguagem perl e também em muitas outras deve ficar entre aspas podendo ser aspas duplas ou ate aspas simplesmente, a diferença entre strings em aspas duplas e aspas simples que na dupla ela reconhece alguns caracteres de escape como por exemplo o \n que seria a quebra de linha já no aspas simples apenas alguns muito específicos

```
#/usr/bin/perl

use strict;
use warnings;

my ($string1, $string2);

$string1 = "aspas duplas \n";
$string2 = 'aspas simples \n';
```

```
print($string1);
print($string2);
```

outra forma da gente formar uma string sem ser com aspas é usando o qq seguido de um carácter especial sendo que esse carácter precisara esta no final da string indicando o fim dela

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = qq/kodo no kami/;
my $str2 = qq%mmxm%;
my $str3 = qq&sir.rafiki&;
my $str4 = qq$51m0n$;

print("$str1, $str2, $str3, $str4 \n");
```

a gente pode juntar ou concatenar duas ou mais strings em uma única string usando o ponto independente se ela é aspas simples ou duplas ou com qq

```
#/usr/bin/perl

use strict;
use warnings;

my $string1 = "aspas duplas \n" .  'aspas simples \n' . qq(\nusando qq \n);

print($string1);
```

podemos iniciar a string em uma linha e continuar em outra linha bastando colocar contra barra no final de cada nova linha

```
#/usr/bin/perl

use strict;
use warnings;

my $string = "isso é uma string \
que é um conjunto de caracteres";

print($string);
```

é possível colocar uma variável dentro de uma string de aspas duplas

```
#/usr/bin/perl

use strict;
use warnings;

my $string1 = "kodo no kami";
my $string2 = "seu nome é $string1";

print($string2);
```

porem não é possível fazer o mesmo com uma string de aspas simples

```
#/usr/bin/perl

use strict;
use warnings;

my $string1 = 'kodo no kami';
my $string2 = 'seu nome é $string1';

print($string2);
```

para fazer isso temos que concatenar elas

```
#/usr/bin/perl

use strict;
use warnings;

my $string1 = 'kodo ';
my $string2 = 'seu nome é ' . $string1;

print($string2);
```

podemos concatenar duas variáveis e atribuir a mesma variável

```
#/usr/bin/perl

use strict;
use warnings;

my $string1 = "nome: ";
my $string2 = "kodo no kami";
$string1 = $string1 . $string2;

print($string1);
```

como podemos ver no exemplo anterior a gente concateno a duas variáveis e atribuímos a ela mesma, também podemos fazer de outra forma para o mesmo caso indicando que vamos atribuir para a mesma variável concatenando uma outra, para fazer isso usamos ponto e igual que seria o mesmo de concatenar a própria variável a outra

```
#/usr/bin/perl

use strict;
use warnings;

my $string1 = "nome: ";
my $string2 = "kodo no kami";
$string1 .= $string2;

print($string1);
```

é possível ver a quantidade de caracteres em um string usando a função length, essa função retorna a quantidade de caracteres que tem a string

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "kodo no kami";
my $tamanho = length($nick);

print("essa string tem $tamanho de caracteres");
```

podemos usar a função index para retornar a quantidade de posição ate um trecho em uma string esse trecho sera sempre o primero mesmo que exista outros iguais na string, para usar essa função passamos como argumento a string da onde vamos buscar e o trecho que estamos buscando e ela retorna a quantidade de posição ate aquele trecho ou -1 caso não encontre

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "kodo no kami";
my $pos = index($nick,"kami");

print("kami esta na posição $pos");
```

pela função rindex é possível descobrir a posição de um determinado trecho na string porem sempre sera a posição do ultimo trecho que esteja na string

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "o kami do meu nick vem do xintoísmo onde kami é uma divindade";
my $pos = rindex($nick,"kami");

print("o ultimo kami esta na posição $pos");
```

a função strstr do modulo POSIX tem a mesma utilidade da função index

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $nick = "o kami do meu nick vem do xintoísmo onde kami é uma divindade";
my $pos = strstr($nick,"kami");

print("kami esta na posição $pos");
```

podemos recortar parte da string a partir de uma posição especifica usando a função substr, para usar ela especificamos a string seguindo da posição da onde vai começar

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "o kodo do meu nick vem do japones onde kõdo ou koodo significa codigo";
my $recortado = substr($nick,10);

print("$recortado");
```

também podemos especificar uma quantidade maxima de caracteres depois da posição que sera recortada, exemplo se eu quiser recortar apenas a palavra nick do exemplo anterior sendo que a palavra começa no 14 e tem 4 caracteres

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "o kodo do meu nick vem do japones onde kõdo ou koodo significa codigo";
my $recortado = substr($nick,14,4);
```

```
print("$recortado");
```

também é usar o substr invertidamente fazendo ele pegar os caracteres de trás para frente, para fazer usamos números negativos nele

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "o kodo do meu nick vem do japones onde kõdo ou koodo significa codigo";
my $recortado = substr($nick,-31);

print("$recortado");
```

da mesma forma também podemos especificar uma quantidade máxima de caracteres

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "o kodo do meu nick vem do japones onde kõdo ou koodo significa codigo";
my $recortado = substr($nick,-31,14);

print("$recortado");
```

podemos usar o substr para substituir ou adicionar caracteres em uma string bastando adicionar o novo carácter ou string depois das posições, sendo a primeira posição a onde vai adicionar e a segunda quantidade de caracteres que vão ser removido

```
#/usr/bin/perl

use strict;
use warnings;

my $deathnote = "pague 50% da sua vida para ganhar os olhos de shinigami";
substr($deathnote,6,1,"2");

print("oferta para kira: $deathnote");
```

com a função join podemos juntar varias string ou variáveis em uma única string com um separador entre cada uma delas, para usar o join basta a gente especificar o separador e depois usar as variáveis ou string separadas por virgula

```
#/usr/bin/perl

use strict;
use warnings;

$anime1 = "trinity seven";
$anime2 = "highschool of the dead";
$anime3 = "seikon no qwaser";
$anime4 = "noragami";

my $junto = join(":",$anime1,$anime2,$anime3,$anime4);

print($junto);
```

também pode usar array ao invés de variáveis assim eles vai juntar todos dados dentro

```
#/usr/bin/perl

use strict;
use warnings;

my @animes = ("death note","attack on titan","elfen lied", "school days");
my $junto = join(":",@animes);

print("$junto \n");
```

a gente pode usar a função split para fazer o oposto da função join no caso separar e adicionar essas string separadas em arrays com base em um separador que pode ser um carácter ou ate uma string, para usar essa função especificamos o carácter que vai ser o separador e depois a string

```
#/usr/bin/perl

use strict;
use warnings;

my $animes = "battle programmer shirase - highschool dxd - campione - monogatari series";
my @separado = split("-",$animes);

print($separado[0]);
```

para a gente deixar todos os caracteres em maiúsculos ou caixa alta usamos a função uc

```
#/usr/bin/perl

use strict;
use warnings;
```

```
my $nick = uc("kodo no kami");

print("$nick");
```

para fazer o oposto deixando todos em minúsculos ou caixa baixa usamos a função lc

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = lc("KODO NO KAMI");

print("$nick");
```

veja um exemplo de como deixar apenas o primeiro carácter maiúsculo usando o uc e substr

```
#/usr/bin/perl

use strict;
use warnings;

my $nick = "kodo no kami";
substr($nick,0,1,uc(substr($nick,0,1)));

print("$nick");
```

2.8 – Expressão regular (regex)

existem formas mais eficiente para manipular strings que o substr e uma dessas formas são as expressões regulares ou regex, as regex são formatos de códigos que são feitos para filtrar uma string formatando ela, podemos usar regex para comparar uma string se bate com a regex, podemos usar regex para recortar determinado trecho de uma string, podemos usar regex ate para substituir determinado trecho em uma string, as regex em perl são nativas da linguagem diferente de python que tem usar o modulo re e para usar regex na linguagem perl apenas colocamos a string usamos o sinal de igual seguido de til depois usamos nossa regex entre barras

```
#/usr/bin/perl

use strict;
use warnings;

"eu sou o kodo no kami" =~ //;
```

a gente pode atribuir o resultado para uma variável scalar

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ //;
```

também é possível comparar com a regex uma variável

```
#/usr/bin/perl

use strict;
use warnings;

my $texto = "eu sou o kodo no kami";
my $resultado = $texto  =~ //;
```

se a regex casar ou seja a comparação for verdadeira ele retorna 1 se não ele retorna 0 (podemos usar isso futuramente em uma estrutura condicional)

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ //;

printf("regex: %d \n",$resultado);
```

no exemplo anterior a gente não especifico nenhuma regex então seria verdadeiro qualquer coisa na string com isso retorno é 1, a gente pode especificar uma palavra existente na nossa string que o retorno sera 1 tambem

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ /kodo no kami/;

printf("regex: %d \n",$resultado);
```

se a gente usar uma regex que não tenha na nossa string o retorno é 0 ou seja falso por que não casou

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ /fts315/;

printf("regex: %d \n",$resultado);
```

a regex deve bater exatamente com a string para ser verdeiro por exemplo se tivesse o kodo no kami porem alguma coisa a mais ali na regex ia retornar falso, veja no exemplo a baixo na regex depois do kami tem um espaço já na string não ou seja o espaço ali não casa com a string

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ /kodo no kami /;

printf("regex: %d \n",$resultado);
```

porem se fosse oposto casaria por existir aquele padrão na string e com isso retornando 1

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami " =~ /kodo no kami/;

printf("regex: %d \n",$resultado);
```

não precisa ser toda a palavra apenas um único caracter é valido

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami " =~ /k/;

printf("regex: %d \n",$resultado);
```

quando a gente não sabe um determinado carácter da string e ainda precisa especificar ele podemos

usar ponto no lugar dele que seria qualquer carácter da string

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ /ko.o no kami/;

printf("regex: %d \n",$resultado);
```

lembrando que ate os espaços são caracteres com isso é necessário especificar eles também caso exista para casar, no exemplo a baixo eu usei duas vezes o ponto para especificar o do porem tinha dois espaço entre o kodo e kami que não especificado com isso não casou

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ /kodo..kami/;

printf("regex: %d \n",$resultado);
```

quando temos o trecho de código desconhecido ate mesmo na quantidade podemos usar ponto seguido de asterisco ele vai ler qualquer trecho ate o próximo caractere da regex

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eu sou o kodo no kami" =~ /eu.*kami/;

printf("regex: %d \n",$resultado);
```

para especificar um numero usamos \d

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "hfts315" =~ /hfts\d\d\d/;

printf("regex: %d \n",$resultado);
```

para especificar um carácter que não seja numero usamos \D

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "hfts315" =~ /\D\D\D\D\d\d\d/;

printf("regex: %d \n",$resultado);
```

para especificar um carácter que seja uma letra usamos \w

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "hfts315" =~ /\w\w\w\w\d\d\d/;

printf("regex: %d \n",$resultado);
```

para especificar um carácter que não seja uma letra usamos \W

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "hfts315" =~ /\w\w\w\w\W\W\W/;

printf("regex: %d \n",$resultado);
```

para especifica o carácter espaço usamos \ seguido do espaço

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "kodo no kami" =~ /kodo\ no\ kami/;

printf("regex: %d \n",$resultado);
```

para especificar a quebra de linha usamos \n

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "kodo no kami\nhfts315" =~ /kami\nhfts/;

printf("regex: %d \n",$resultado);
```

para especificar o carácter contra barra usamos \\

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "c:\\windows\\system32" =~ /windows\\system32/;

printf("regex: %d \n",$resultado);
```

para especificar o carácter barra usamos \/

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "/usr/bin" =~ /\/usr\/bin/;

printf("regex: %d \n",$resultado);
```

se a gente precisar especificar uma quantidade do mesmo tipo de carácter ate a próximo carácter na regex basta adicionar um + logo após o carácter, exemplo o numérico \d+

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "hfts315" =~ /\w+\d+/;

printf("regex: %d \n",$resultado);
```

isso pode ser feito com qualquer outro carácter por exemplo

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "eeeeeeeeaaaaaeeee blz =~ /e+a+e+ blz/;

printf("regex: %d \n",$resultado);
```

podemos especificar uma quantidade de ocorrência de um determinado carácter na string usando chaves depois do carácter e dentro da chaves a quantidade

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "00 0000-0000" =~ /\d{2} \d{4}-\d{4}/;

printf("regex: %d \n",$resultado);
```

também podemos especificar entre uma quantidade de ocorrência bastando separar com virgula o começo e fim com isso se tiver uma ocorrência entre essa faixa retorna verdadeiro,

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "1000001 101 11110" =~ /\d{0,8} \d{0,8} \d{0,8}/;

printf("regex: %d \n",$resultado);
```

podemos especificar determinados caracteres usando colchetes e dentro deles os caracteres, no exemplo pode ser gato como pode ser gata

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Eu tenho 6 gatos e 2 cachorros" =~ /gat[ao]/;
```

```
printf("regex: %d \n",$resultado);
```

para não precisar digitar todos os caracteres por exemplo de a ate z podemos colocar um hífen na frente

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Eu tenho 6 gatos e 2 cachorros" =~ /gat[a-z]/;

printf("regex: %d \n",$resultado);
```

porem dessa forma só vai especificar de a ate z minusculas, podemos especificar A ate Z e 0 ate 9 também

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Eu tenho 6 gatos e 2 cachorros" =~ /[A-Z]u tenho [0-9] [a-g]at[a-z]/;

printf("regex: %d \n",$resultado);
```

também é possível colocar todos em um único colchete por exemplo sendo a ate z maiúsculos e minúsculos e números

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Eu tenho 6 gatos e 2 cachorros" =~ /[a-zA-Z0-9]u tenho/;

printf("regex: %d \n",$resultado);
```

para a gente especificar algumas ocorrência de palavra usamos o pipe para separar as palavras, no exemplo abaixo a gente verifica se existe camelos ou gatos na string

```
#/usr/bin/perl

use strict;
```

```
use warnings;

my $resultado = "Eu tenho 6 gatos e 2 cachorros" =~ /tenho 6 camelos|tenho 6 gatos/;

printf("regex: %d \n",$resultado);
```

podemos agrupar ele entre parênteses para testar um trecho especifico por exemplo se eu quiser verificar se eu tenho 6 gatos ou tenho 6 camelos na string sem precisar escreve eu tenho 6 duas vezes

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Eu tenho 6 gatos e 2 cachorros" =~ /tenho 6 (camelos|gatos)/;

printf("regex: %d \n",$resultado);
```

as vezes temos que testar determinado string e não sabemos se eles esta em maiúsculos ou minúsculos, então para especifica a regex o uso de tanto caixa alta quanto baixa podemos usar o i no final

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Kodo No Kami" =~ /kodo no kami/i;

printf("regex: %d \n",$resultado);
```

podemos usar o s no final para verificar a string toda incluindo depois da quebra de linha

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "nick kodo no kami\nnome flavio\nidade 23" =~ /kodo no kami.*\d/s;

printf("regex: %d \n",$resultado);
```

tambem é possível especifica o inicio de uma string usando ^ bastando colocar ele no começo , um exemplo de como verificar se o primeiro caracter é uma letra e maiuscula

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Kodo no kami" =~ /^[A-Z]/;

printf("regex: %d \n",$resultado);
```

tambem é possível especificar o final com $ bastando colocar ele no final da regex

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "Kodo no kami" =~ /kami$/;

printf("regex: %d \n",$resultado);
```

uma regex verificando o primeiros caracteres e o ultimo

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = "kodo no kami" =~ /^kodo.*kami$/;

printf("regex: %d \n",$resultado);
```

ate agora a gente viu como checar uma string com uma regex sem recortar o texto, no caso para recortar um trecho da string usando regex é muito simples basta atribuir ela para uma array ao invés de uma variável scalar, também temos que agrupar com parenteses a parte que queremos recortar

```
#/usr/bin/perl

use strict;
use warnings;

my @resultado = "nick kodo no kami" =~ /nick (kodo no kami)/;

print($resultado[0]);
```

podemos usar os exemplos anteriores para recortar mais precisamente

```
#/usr/bin/perl

use strict;
use warnings;

my @resultado = "nick kodo no kami" =~ /^nick (.*)/;

print($resultado[0]);
```

as vezes usar o .* não é muito preciso isso por que se tiver mais de uma palavra igual na string ele vai recortar ate a ultima ocorrência e não a primeira

```
#/usr/bin/perl

use strict;
use warnings;

my @resultado = "sou kodo no kami - boku wa kodo no kami" =~ /(.*kami)/;

print($resultado[0]);
```

para recortar ate a primeira ocorrência podemos usar o interrogação depois do asterisco

```
#/usr/bin/perl

use strict;
use warnings;

my @resultado = "eu sou kodo no kami - boku wa kodo no kami desu" =~ /(.*?kami)/;

print($resultado[0]);
```

podemos usar mais agrupamentos com parêntese para armazenar em outras posições da array

```
#/usr/bin/perl

use strict;
use warnings;

my @resultado = "eu sou kodo no kami - boku wa kodo no kami desu" =~ /(.*) - (.*)/;

print("pt: $resultado[0]\n");
print("jp: $resultado[1]\n");
```

a gente pode usar o “g” no final para recortar um grupo ou seja qualquer outro texto igual a regex e armazenar eles na array

```
#/usr/bin/perl

use strict;
use warnings;

my @resultado = "cor vermelho - cor amarelo - cor azul" =~ /cor (\w+)/g;

print("$resultado[0]\n");
print("$resultado[1]\n");
print("$resultado[2]\n");
```

é possível substituir parte da string usando uma regex, para fazer isso usamos a letra s no começo da regex e colocamos duas barra a primeira é o trecho da string que a gente vai substuir e a segunda sera a palavra

```
#/usr/bin/perl

use strict;
use warnings;

my $texto = "bolo de baunilha";
$texto  =~ s/de baunilha/de chocolate/;

print("$texto \n");
```

3.0 – Logica aritmética

além da manipulação de string também podemos manipular números fazendo contas como adição, subtração, multiplicação, divisão, modulos isso permite a gente criar quase todos os tipos algorítimos seja matemático ou não, para a gente fazer operaçoes de soma usamos o sinal de mais

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 300 + 15;

printf("%d",$resultado);
```

podemos somar vários valores ao mesmo temporariamente

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my $resultado = 300 + 15 + 80 + 2000;

printf("%d",$resultado);
```

também é possível somar valores em variáveis, não só a adição como as outras operações também

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 300;
my $numero2 = 15;
my $resultado = $numero1 + $numero2;

printf("%d",$resultado);
```

podemos usar sobrecarregar o operador assim somando duas ou mais e atribuindo a ela mesmo usando o operador adição seguido do igual

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 300;
my $numero2 = 15;
$numero1 +=  $numero2;

printf("%d",$numero1);
```

além da soma temos a subtração que é usado o sinal de menos

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 300 - 15;

printf("%d",$resultado);
```

também podemos atribuir subtrair determinada variável e atribuir a ela mesmo com o operador

menor e o sinal de igual

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 300;
my $numero2 = 15;
$numero1 -=  $numero2;

printf("%d",$numero1);
```

além da adição e subtração temos a multiplicação que é usado o asterisco

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 300 * 15;

printf("%d",$resultado);
```

da mesma forma é possível atribuir a ele mesmo usado o operador asterisco e igual

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 300;
my $numero2 = 15;
$numero1 *=  $numero2;

printf("%d",$numero1);
```

temos a divisão também que é usado o barra como operador

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 300 / 15;
```

```
printf("%d",$resultado);
```

ela não é diferente dos demais, também da para atribuir uma variavel a ela mesmo fazendo essa operação com uma outra com operador barra seguido de igual

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 300;
my $numero2 = 15;
$numero1 /=  $numero2;

printf("%d",$numero1);
```

além da divisão normal a gente pode conseguir o resto dela tambem chamado de modulo, para isso usamos o operador porcentagem

```
#/usr/bin/perl

use strict;
use warnings;

my $resu = 10 / 7;
my $modu = 10 % 7;

printf("resultado = %d \n",$resu);
printf("resto = %d \n",$modu);
```

As variáveis também pode ser atribuído a ela mesmo fazendo essa operação bastando usar o operador porcentagem seguido do igual

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 10;
my $numero2 = 7;
$numero1 %=  $numero2;

printf("%d",$numero1);
```

também podemos usar potencia usando o operador asterisco seguido de asterisco

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 3 ** 2;

printf("%d",$resultado);
```

e da mesma forma também podemos atribuir ele a mesma variável fazendo essa operação

```
#/usr/bin/perl

use strict;
use warnings;

my $numero1 = 3;
my $numero2 = 2;
$numero1 **=  $numero2;

printf("%d",$numero1);
```

também é possível usar potencia com a função pow do modulo POSIX

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $resultado = pow(3,2);

printf("%d",$resultado);
```

a gente pode fazer varias dessas operação junto

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 300 + 1 - 30 + 50 - 8 + 9;

printf("%d",$resultado);
```

porem como regra na matemática alguns operadores como multiplicação e divisão tem prioridade

do que adição e subtração, com isso o exemplo a baixo vai ter como resultado o numero 330 e não o numero 630 isso por que ele fez o 15 * 2 primeiro

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = 300 + 15 * 2;

printf("%d",$resultado);
```

para a gente dar prioridade a determinada operação basta agrupar ela entre parênteses, o exemplo anterior deveria ser feito dessa forma

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = (300 + 15) * 2;

printf("%d",$resultado);
```

a gente pode agrupar quantas vezes a gente quiser inclusive dentro do próprio agrupamento e a prioridade sempre sera sempre de dentro para fora,

```
#/usr/bin/perl

use strict;
use warnings;

my $resultado = (((300 + 15) * 2) + 10) * 5 ;

printf("%d",$resultado);
```

também é possível usar números negativos bastando colocar um menos antes do numero

```
#/usr/bin/perl

use strict;
use warnings;

my $positivo = 315;
my $negativo = -315;
```

```
printf("%d %d",$positivo,$negativo);
```

podemos conseguir o valor absoluto do numero com a função abs, no caso ele remove o negativo do numero deixando ele sempre positivo

```
#/usr/bin/perl

use strict;
use warnings;

my $positivo = abs(315);
my $negativo = abs(-315);

printf("%d %d",$positivo,$negativo);
```

para conseguir a raiz quadrada de um numero usamos a função sqrt

```
#/usr/bin/perl

use strict;
use warnings;

my $raiz = sqrt(81);

printf("%d",$raiz);
```

para incrementar mais 1 a variável podemos usar duas vezes o sinal de mais depois da variável, isso é bastante útil em loop

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 315;
$num++;

printf("%d",$num);
```

o exemplo anterior seria equivalente a esse

```
#/usr/bin/perl

use strict;
use warnings;
```

```
my $num = 315;
$num = $num + 1;

printf("%d",$num);
```

também podemos decrementar 1 usando menos menos depois do nome da variavel

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 315;
$num--;

printf("%d",$num);
```

alem dos numeros do tipo inteiro podemos usar numeros do tipo quebrado (reais, com casa decimais, flutuante, chame como você quiser), esses numero são usados pontos e não virgulas

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 3.15;

printf("%f",$num);
```

podemos forçar um numero flutuante a se tornar um inteiro usando typecast que seria usar o tipo como função para converter ele (ou para forçar ele naquele tipo), no caso do float para int ele vai apenas zerar os numeros depois do ponto

```
#/usr/bin/perl

use strict;
use warnings;

my $num = int(3.15);

printf("%f",$num);
```

outra forma para zerar os números depois do ponto é usar a função floor do modulo POSIX

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $num = floor(3.15);

printf("%f",$num);
```

se a gente usar a função ceil tambem do modulo POSIX, ele arredonda o numero para o próximo e zera depois do ponto

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $num = ceil(3.15);

printf("%f",$num);
```

também é possível manipular números muito grandes com notação cientifica, para isso usamos um numero base seguido da letra 'e' do sinal de mais e o numero expoente, por exemplo o numero 3000000 seria o 3e+6 que seria equivalente tambem a 3*10^6, uma forma mais simples de dizer que a gente adiciono mais seis números 0 ao nosso numero base

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 3e+6;

printf("%d",$num);
```

também podemos fazer o inverso pegar um numero grande e retirar as casas bastando usar o sinal de menos no lugar do sinal de mais, por exemplo o numero 15000000 poderia ser manipulado apenas como o numero 15 dessa forma 15000000e-6 ou calculando eke como 15000000/10^6

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 15000000e-6;
```

```
printf("%d",$num);
```

também podemos fazer o printf exibir números com notação cientifica com o %e

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 2e+10;

printf("%e",$num);
```

na maioria das vezes a gente esta mexendo com a base 10 ou decimal que é a base que usamos para contar, essa base contem números que vai de 0 ate 9 totalizando 10 por isso é chamado de decimal, depois do numero 9 a gente tem um incremento um novo numero a esquerda do numero e o mesmo foi zerado assim sucessivamente e infinitamente tambem

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

para manipular numeros decimais em perl ou numeros base 10 apenas é necessário digitar o numero, e a representação dele no printf é o %d

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 315;

printf("%d",$num);
```

além da base 10 existe uma base que é 16 também chamada de hexadecimal, ela é muito usada para manipulação de bytes já que um byte é igual 8bits ou 256 estados diferentes e esses 256 é divisível por 16 e não é por 10, além do mais existe os nimble que é igual ao bytes porem tem 4bits apenas que é equivalente a exatamente 16 estados perfeito para ser manipulado pelo base 16 não é?, os numeros hexadecimais vai de 0 a 15 como não existe números depois do 9 é usado letras de A ate F, e depois do F que é incrementado um novo numero a esquerda

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10
```

para usar o base 16 ou hexadecimal em perl adicionamos antes dos numeros o 0x, e a sua representação no printf é %x

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 0x13b;

printf("%x",$num);
```

também existe a base 8 que também é muito usada na computação e é chamada de octal, como a base dele é 8 ele só tem números de 0 ate 7 com isso é excluído o numero 8 e 9

```
0, 1, 2, 3, 4, 5, 6, 7, 10
```

para a gente usar os números octais apenas colocamos o numero 0 antes de qualquer numero e sua representação no printf é o %o

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 0473;

printf("%o",$num);
```

por fim chegamos na base mais usada pelo computador pelo menos em baixo nível que é base 2 também conhecida como binário, como ele só tem duas base ele vai de 0 a 1, ele pode ser usado para manipular um bit dentro de um byte ou uma logica booleana, ate mesmo a separação de um byte para outro é nada mais nada menos que um agrupamento de alguns bits ou seja uma sequencia fixa de numeros binarios

```
0, 1, 10
```

na linguagem perl para manipular numeros binarios usamos 0b antes do numero e sua representação no printf é %b

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 0b100111011;
```

```
printf("%b",$num);
```

todos os números em uma base especifica pode ser lido em outra base embora em outra base o numero vai mudar embora eles ainda vão ser equivalente a outro numero na outra base embora não o mesmo numero com exceção os primeiros numeros, por exemplo o numero 315 em decimal é o numero 0x13b em hexadecimal ou 0473 em octal, mesmo em base diferente eles ainda são o numero 315 em decimal, com isso podemos entrar com um numero em uma base e sair com ele em outra

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 0x13b;

printf("%d",$num);
```

também é possível fazer conta mesmo com bases diferentes

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 1000 + 0x13b - 0b111;

printf("%d",$num);
```

3.1 – Logica booleana

pela logica booleana podemos dizer se determinado objeto, ação, evento ou qualquer outra coisa é dita como verdadeiro ou falso sobre uma condição especifica, podemos usar a logica booleana para testar determinada ação para controlar o fluxo do nosso programa, podemos usar a logica booleana para detectar se um programa fez determinada ação, podemos usar a logica booleana para manter ele preso em um loop sobre uma condição, podemos usar a logica booleana para manipular os bits, ela é usada para muitas outras coisas, de acordo com a logica booleana só pode existir dois estados diferente ou é verdeiro ou é falso não existe um meio termo e os dois estados sendo um oposto ao outro (ligado/desligado, sim/nao, movimentando/parado, vivo/morto, existe/nao existe), pela logica booleana dizemos que o numero 1 ou números positivos são verdadeiros por outro lado o numero 0 e os outros números negativos são falsos, um bom exemplo da logica booleana na linguagem perl são as regex que retorna o numero 1 quando a string bate com a regex e o numero 0 quando não bate

```
#/usr/bin/perl

```

```
use strict;
use warnings;

my $ve = "kodo no kami" =~ /kodo.*kami/;
my $fa = "kodo no kami" =~ /fts315/;

printf("verdadeiro: %d \n",$ve );
printf("falso: %d \n",$fa );
```

podemos usar a logica booleana ate no nosso dia a dia por exemplo “você esta lendo esse ebook na segunda feira!”, verdade ou falso?

3.2 – Operação bit a bit

as operações bit a bit permite a gente manipular os bits dentro de um byte, as operações bit a bit usa dois bytes na maior parte das vezes e o primeiro byte é o que vamos modificar já o segundo byte sera usando para modificar ele e com base nesses dois ele retorna uma nova sequencia de bits, a forma mais fácil de manipular os bits é usando base 2 isso por que o binário é uma sequencia dos bits então modificar ele em binário é mais simples que usar decimal porem isso não faz diferença para maquina apenas para a gente mesmo, uma dessas operações bit a bit é o “OR”, com essa operação a gente compara os bits de um byte equivalente aos bits do outro byte se um dos bits for verdadeiro bit de retorno é verdadeiro se os dois for falso o bit de retorno é falso

```
01100001
01100011 or
–----------
01100011
```

na linguagem perl para fazer a operação bit a bit or é usado o operador |

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 0b1100001;
my $num2 = 0b1100011;

my $resu = $num1 | $num2;

printf("%b",$resu);
```

também existe a operação and que compara os bits e retorna verdadeiro caso os dois bits equivalente seja verdadeiro, se um deles ou os dois for falso o retorno é falso

```
01100001
01100011 and
–----------
01100001
```

na linguagem perl o sinal de operação bit a bit para o and é o &

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 0b1100001;
my $num2 = 0b1100011;

my $resu = $num1 & $num2;

printf("%b",$resu);
```

também existe o XOR esse é o contrario do OR, só retorna verdadeiro se um bit for verdadeiro e o outro falso ou seja eles forem opostos

```
01100001
01100011 xor
–----------
00000010
```

na linguagem perl o operador bit a bit xor é representado pelo sinal de ^

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 0b1100001;
my $num2 = 0b1100011;

my $resu = $num1 ^ $num2;

printf("%b",$resu);
```

existe o operador bit a bit NOT esse usa apenas um byte e ele inverte todos os bits desse byte ou seja se é verdadeiro vira falso e se é falso vira verdadeiro

```
01100001 not
```

```
––----------
10011110
```

em perl o operador bit a bit NOT é representado por ~

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 0b1100001;

my $resu = ~ $num1;

printf("%b",$resu);
```

a gente esta usando números binários porem pode ser feito o mesmo com outras bases ou ate caracteres lembrando que tudo é byte

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 97;
my $num2 = 120;

my $resu = $num1 | $num2;

printf("%d",$resu);
```

3.3 – Operadores de comparação e logico

os operadores de comparação e logicos são parecidos com os operadores bit a bit porem eles ao invés de comparar um unico bit por vez e retornar um novo bit com base no anterior ele compara tudo e retorna verdadeiro ou falso como um todo, podemos usar operadores de comparação para comparar duas strings e retornar verdadeiro ou falso se elas são iguais, podemos usar o operadores de comparação para comparar números se um numero é igual a outro, podemos usar operadores de comparação para verificar se são diferentes, iguais, maior ou menor que o outro, na linguagem perl o operador que compara um numero é diferente do que compara uma string, na comparação de numeros usamos duas vezes o sinal de igual

```
#/usr/bin/perl

use strict;
use warnings;
```

```
my $num1 = 315;
my $num2 = 315;

my $resu = $num1 == $num2;

printf("%d",$resu);
```

já na comparação de string usamos a palavra eq como operador

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo no kami";
my $str2 = "kodo no kami";

my $resu = $str1 eq $str2;

printf("%d",$resu);
```

lembrando que também podemos usar regex para comparar

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo no kami";
my $str2 = "kodo no kami";

my $resu = $str1 =~ /$str2/;

printf("%d",$resu);
```

para comparar para ver se os numeros são diferente e com isso retornando verdadeiro usamos o operador !=

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 315;
my $num2 = 100;
```

```
my $resu = $num1 != $num2;

printf("%d",$resu);
```

na comparação de string para verificar se são diferentes usamos o operador ne

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo no kami";
my $str2 = "fts315";

my $resu = $str1 ne $str2;

printf("%d",$resu);
```

para ver se o numero é maior usamos o operador >

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 315;
my $num2 = 100;

my $resu = $num1 > $num2;

printf("%d",$resu);
```

na string usamos o operador gt para verificar se um string é maior que outra, porem para as strings não é verificado o tamanho e sim a ordem alfabética

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo no kami";
my $str2 = "fts315";

my $resu = $str1 gt $str2;

printf("%d",$resu);
```

no caso para comparar pelo tamanho usamos a função length e comparar o tamanho delas

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo no kami";
my $str2 = "fts315";

my $resu = length($str1) > length($str2);

printf("%d",$resu);
```

para comparar se o numero é menos usamos o operador <

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 15;
my $num2 = 300;

my $resu = $num1 < $num2;

printf("%d",$resu);
```

para as string usamos o operador lt para comparar se a string é menor em ordem alfabética que a outra

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "fts315";
my $str2 = "kodo no kami";

my $resu = $str1 lt $str2;

printf("%d",$resu);
```

também podemos verificar se é maior ou igual usando o operador >=

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my $num1 = 300;
my $num2 = 300;

my $resu = $num1 >= $num2;

printf("%d",$resu);
```

para comparação de string usamos o ge para maior ou igual

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo no kami";
my $str2 = "kodo";

my $resu = $str1 ge $str2;

printf("%d",$resu);
```

para o oposto menor ou igual usamos o operador <=

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 300;
my $num2 = 315;

my $resu = $num1 <= $num2;

printf("%d",$resu);
```

para menor ou igual para as strings usamos le

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "fts315";
```

```
my $str2 = "kodo";

my $resu = $str1 le $str2;

printf("%d",$resu);
```

é possível comparar vários operadores simultaneamente usando os operadores logicos para gerar um retorno final sobre esses outros operadores, esses operadores logicos compara o retorno dos outros operadores e dependendo da logica dele retorna verdadeiro ou falso, os operadores logicos são parecidos com os operadores bit a bit porem não confunda eles os operadores bit a bit compara cada bit em um byte e gera um retorno de um novo bit já os operadores logicos compara dois ou mais retornos e gera um novo retorno podendo ser verdadeiro ou falso, o operador logico OR compara dois ou mais retornos e se um deles for verdadeiro o retorno é verdadeiro, para usar o operador logico OR usamos ||

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = 315;
my $str2 = "kodo";
my $str3 = "kodo";

my $resu = ($str1 == 100) || ($str2 eq $str3 );

printf("%d",$resu);
```

o operador logico AND apenas retorna verdadeiro se todos os outros retornar verdadeiro tamabem, e para usar ele usamos o operador &&

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = 315;
my $str2 = "kodo";
my $str3 = "kodo";

my $resu = ($str1 == 315) && ($str2 eq $str3 );

printf("%d",$resu);
```

também existe o operador NOT que inverte se é verdadeiro vira falso ou se é falso vira verdadeiro, para usar ele usamos operador ! antes

```
#/usr/bin/perl

use strict;
use warnings;

my $str1 = "kodo";
my $str2 = "kami";

my $resu =  !($str1 eq $str2);

printf("%d",$resu);
```

3.4 – Estruturas condicionais

com as estruturas condicionas a gente pode controlar o fluxo do nosso programa permitindo ele executar determinado trecho ou não isso com base em uma condição podendo ser ela operadores logicos ou de comparação, existem diversos tipos de estruturas condicionais na linguagem perl porem a mais usada é o if que permite executar determinado bloco caso a condição dentro dele seja verdeiro caso ela seja falsa não executa, para usar o if a gente coloca a condição entre parênteses e tambem cria um escopo se a condição for satisfeita ele executa o trecho do codigo dentro do escopo se não ele não executa

```
#/usr/bin/perl

use strict;
use warnings;

my $senha = "kodo no kami";

if($senha eq "kodo no kami")
{
    print("acesso permitido");
}
```

podemos usar quantos if a gente quiser

```
#/usr/bin/perl

use strict;
use warnings;

my $idade = 23;

if($idade < 18)
{
    print("voce é menor de idade");
}
```

```
if($idade > 18)
{
    print("voce é maior de idade");
}
```

também é possível usar o if dentro de outro if

```
#/usr/bin/perl

use strict;
use warnings;

my $idade = 16;

if($idade < 18)
{
    print("voce é menor de idade\n");
    if($idade >= 13 )
    {
        print("e tem entre 13 a 17 anos")
    }
}
```

além do if existe o else que é usado em conjunto do if, ele permite ser executado caso o if não seja e ele deve ficar sempre depois de um if embora o uso dele é opcional

```
#/usr/bin/perl

use strict;
use warnings;

my $opcao = 5;

if($opcao == 1)
{
    print("voce escolheu opção 1\n");
}
else
{
   print("essa opção não é valida\n");
}
```

podemos usar o else if ou elsif que permite adicionar vários if um após o outros, quando o primeiro deles for satisfeito ele para de verificar, e ele deve ser usado depois de um if e antes do else

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my $opcao = 2;

if($opcao == 1)
{
    print("uva\n");
}

elsif($opcao == 2)
{
    print("morango\n");
}

elsif($opcao == 3)
{
    print("pera\n");
}

else
{
   print("essa opção não é valida\n");
}
```

além do if existe o unless que seria o if com uma condição invertida

```
#/usr/bin/perl

use strict;
use warnings;

my $senha = "123";

unless($senha eq "321")
{
    print("senha valida\n");
}
else
{
   print("senha invalida")
}
```

o unless seria o mesmo que usar o if com operador logico not

```
#/usr/bin/perl

use strict;
```

```
use warnings;

my $senha = "123";

if(!($senha eq "321"))
{
    print("senha valida\n");
}
else
{
   print("senha invalida")
}
```

também temos o if ternário que é uma forma muito compacta de usar o if apenas retornando alguma coisa caso seja verdadeiro e outra caso seja falso, para usar o if ternário colocamos a condição depois o interrogação o retorno caso seja verdadeiro depois dois pontos e o retorno caso seja falso

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 315;
my $num2 = 300;

my $retorno = ($num1 > $num2) ? "maior" : "menor";

print($retorno);
```

também é possível usar funções diretamente nos retornos dele

```
#/usr/bin/perl

use strict;
use warnings;

my $num1 = 300;
my $num2 = 315;

(315 > 60) ? print("maior") : print("menor");
```

3.5 – Estruturas de repetição

além das estruturas condicionais temos as estruturas de repetição ou de laço que permite repetir determinado trecho de código ate que uma condição seja satisfeita, e dentro dessa condição também é usado a logica booleana como as estruturas condicionais porem a cada loop é checado novamente para ver se a condição foi satisfeita, uma das estruturas de repetição é o while que repete enquanto

sua condição for verdadeira, para usar o while basta passar a condição entre parenteses e dentro de um escopo usamos nosso código que vai repetir

```
#/usr/bin/perl

use strict;
use warnings;

my $loop = 1;

while($loop == 1)
{
    print("repetindo\n");
}
```

no exemplo anterior gero um loop infinito e isso pode ser bem problematico e perigoso em alguns pontos, para evitar um loop infinito usamos uma variavel e vamos incrementando ela a cada loop como contador quando essa variavel chegar em um determinado ponto o loop para

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 0;

while($cont != 10)
{
    print("repetindo\n");
    $cont++;
}
```

também podemos fazer a logica inversa decrementando o contador

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 10;

while($cont != 0)
{
    print("repetindo\n");
    $cont--;
}
```

podemos usar esse contador tambem por exemplo como a cada loop o contador incrementa 1 ou seja a cada loop o contador é o valor antigo adicionado mais 1 com isso podemos usar ele para uma tabuada

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 0;
my $num;

while($cont != 10)
{
    $num = $cont * 5;
    print("5 x $cont = $num \n");
    $cont++;
}
```

a gente pode usar uma estrutura dentro da outra, o exemplo anterior porem para uma tabuada de 0 ate 10

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 0;
my $cont2 = 0;
my $num;

while($cont <= 10)
{
    while($cont2 <= 10)
    {
        $num = $cont * $cont2;
        print("$cont x $cont2 = $num \n");
        $cont2++;
    }
    $cont2 = 0;
    $cont++;
}
```

se a gente precisar parar um loop usamos o last

```
#/usr/bin/perl

use strict;
```

```
use warnings;

while(1 == 1)
{
    print("loop infinito");
    last;
}
```

podemos usar ele em conjunto com o if

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 0;

while(1 == 1)
{
    print("$cont, ");
    if($cont == 100)
    {
        last;
    }
    $cont++;
}
```

na estrutura de repetição while apenas executa o escopo se a condição for verdadeira ou seja se a condição for falsa de cara ele não executa nada, para executa apenas uma vez podemos usar a estrutura do-while que executa uma vez o escopo e depois checa a condição while se for verdadeiro ele repete o “do” ate a condição seja falsa, para usar o do-while criamos um escopo “do” onde colocamos o codigo que sera executado pelo menos uma vez e depois o while com a condição caso seja verdeiro ele vai repetir novamente o do

```
#/usr/bin/perl

use strict;
use warnings;

do
{
   print("repetindo\n");
}
while(1 == 0);
```

na linguagem perl também existe a estrutura until que é parecida com a estrutura while a diferença que ela repete se for falso diferente da while que repete se for verdadeiro

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 1;

until($cont == 0)
{
    print("vish \n");
}
```

no caso o until seria nada mais nada menos que o while usando a logica not

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 1;

while(!($cont == 0))
{
    print("vish \n");
}
```

as estruturas while e until tambem permite usar o continue que seria uma estrutura usada depois deles que permite a cada loop da estrutura anterior tambem executar uma vez o continue

```
#/usr/bin/perl

use strict;
use warnings;

while(1 == 0)
{
   print("isso é o while\n");
}
continue
{
   print("isso é o continue\n");
}
```

também existe a estrutura for que permite especificar o valor inicial na variável, condição e o incremento da variável separando eles por ponto e virgula, e a estrutura for repete enquanto a condição for verdadeira

```
#/usr/bin/perl

use strict;
use warnings;

my $cont;

for($cont = 0; $cont <= 10; $cont++)
{
    print("tabuada de 5: $cont \n");
}
```

podemos fazer o mesmo com mais de uma variavel apenas separando elas por virgula

```
#/usr/bin/perl

use strict;
use warnings;

my ($cont, $contreg);

for($cont = 0, $contreg = 10; $cont <= 10; $cont++, $contreg--)
{
    print("contador: $cont \n");
    print("regressivo: $contreg \n\n");
}
```

na estrutura for os argumentos são opcionais embora temos sempre que especificar o separador ponto e virgula, uma coisa legal que deixa seu professor doido é ele pedir para você fazer um programa com for sem usar o while e você fazer isso para zoar com a cara dele XD

```
#/usr/bin/perl

use strict;
use warnings;

my $cont;

$cont = 0;
for( ; $cont <= 10 ;)
{
   print("contador: $cont \n");
   $cont++;
}
```

outra estrutura de repetição é foreach que permite pecorrer todas as posições em uma array, para usar o foreach basta passar como argumento a array que a cada loop ele vai esta na próxima posição da array

```
#/usr/bin/perl

use strict;
use warnings;

my @kodo = ("kodo no kami","fts315","flavio");

foreach(@kodo)
{
    print("pos \n");
}
```

para acessar o valor atual da array usamos a variavel $_

```
#/usr/bin/perl

use strict;
use warnings;

my @kodo = ("kodo no kami","fts315","flavio");

foreach(@kodo)
{
    my $atual = $_ ;
    print("pos $atual \n");
}
```

outra forma seria especificar uma variável depois do foreach e antes do parênteses;

```
#/usr/bin/perl

use strict;
use warnings;

my @kodo = ("kodo no kami","fts315","flavio");
my $var;

foreach $var (@kodo)
{
    print("pos $var \n");
}
```

a estrutura foreach não passa de um loop while porem armazenando os valores

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my @kodo = ("kodo no kami","fts315","flavio");
my $var;
my $cont;

$tam = @kodo;
$cont = 0;
while($cont < $tam)
{
    $var = $kodo[$cont];
    print("pos $var \n");
    $cont++;
}
```

também existe o goto e label que são pulos embora não são tão usados em linguagens de alto nivel e linguagens imperativas ou estruturadas porem ele é usado muito em linguagens de baixo nivel como a linguagem assembly, o goto permite a gente pular para um label especifico embora em muitas linguagens de alto nivel ficamos limitado a pular para dentro do label da mesma função ou escopo, para criar um label basta a gente digitar o nome dele que pode ser qualquer um e no final dele usamos dois pontos especificando que ele é um label

```
#/usr/bin/perl

use strict;
use warnings;

kodo: ;
```

para a gente pular para aquele label basta usar o goto seguido do nome do label

```
#/usr/bin/perl

use strict;
use warnings;

goto kodo;
print("programando em python");
kodo:;
print("programando em perl");
```

podemos voltar para uma parte anterior do codigo porem temos que ter cuidado que o goto tambem pode gerar loop infinito

```
#/usr/bin/perl
```

```
use strict;
use warnings;

kodo: ;
print("programando em perl \n");
goto kodo;
```

o label tambem pode ser criado antes de uma função ou ate estrutura

```
#/usr/bin/perl

use strict;
use warnings;

kodo: print("programando em perl \n");
goto kodo;
```

podemos usar o goto com o if para gerar o mesmo efeito das estruturas de repetição (uma coisa legal que as estruturas de repetição ate as condicionais não passa de goto em linguagens como assembly)

```
#/usr/bin/perl

use strict;
use warnings;

my $cont = 0;

kodo: ;

if($cont == 10)
{
    goto sair;
}

print("$cont \n");
$cont++;

goto kodo;

sair:
```

4.0 – Funções

as funções são trechos de código já prontos que permite fazer exatamente aquele mesmo procedimento inúmeras vezes sem precisar criar o mesmo código inúmeras vezes e elas não

interfere uma na outra mesma que use a mesma função inumeras vezes e simultaneamente , as funções são bastante útil isso é podemos reaproveitar elas em outros códigos futuramente sem precisar criar elas de novo, para chamar uma função na linguagem perl a gente pode simplesmente usar o nome dela seguido de abre e fecha parentestes, no caso se ela esta dentro de algum modulo especifico temo que declarar esse modulo tambem

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

if(getuid() == 0)
{
   print("voce é usuario root no linux então cuidado \n");
}
else
{
   print("voce não é root no sistema linux\n");
}
```

uma outra forma de especificar que é uma função seria usar o & antes do nome assim não precisamos usar o parênteses

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

if(&getuid == 0)
{
   print("voce é usuario root no linux então cuidado \n");
}
else
{
   print("voce não é root no sistema linux\n");
}
```

também podemos passar os argumentos para a função entre parenteses depois do nome da função e se tiver mais de um argumento podemos usar o virgula para separar eles

```
#/usr/bin/perl

use strict;
use warnings;

my $num = 315;
```

```
printf("numero = %d",$num);
```

os argumentos são as entradas para a função também existe o retorno dela que é saida dela onde podemos atribuir uma função a uma variável para pegar o retorno

```
#/usr/bin/perl

use strict;
use warnings;

my $num = sqrt(9);

printf("numero = %d",$num);
```

para a gente criar uma função usamos o sub seguido do nome da função no final do nosso codigo e usamos um escopo para escrever nosso codigo dentro dele

```
#/usr/bin/perl

use strict;
use warnings;

sub kodo
{
}
```

dentro do escopo da nossa função usamos as outras funções que quando a nossa função for chamada os codigos dentro do escopo sera interpretado

```
#/usr/bin/perl

use strict;
use warnings;

sub kodo
{
    print("essa é minha função \n");
}
```

para chamar a nossa função basta fazer da mesma forma que as anteriores

```
#/usr/bin/perl

use strict;
```

```
use warnings;

kodo();

sub kodo
{
    print("essa é minha função \n");
}
```

podemos chamar a nossa função quantas vezes a gente quiser

```
#/usr/bin/perl

use strict;
use warnings;

kodo();
kodo();
kodo();

sub kodo
{
    print("essa é minha função \n");
}
```

para retornar um valor da nossa função usamos a palavra return seguido do valor que vamos retornar

```
#/usr/bin/perl

use strict;
use warnings;

my $nome = kodo();
print("a fun kodo retorno $nome");

sub kodo
{
   return "kodo no kami";
}
```

uma vez que o intepretador acha o return ele termina a função ou seja se tiver mais codigos depois que o return for executado eles são ignorados

```
#/usr/bin/perl

use strict;
```

```
use warnings;

my $nome = kodo();
print("a fun kodo retorno $nome");

sub kodo
{
   return "kodo no kami";
   print("depois do retorno");
}
```

podemos retornar uma variavel da nossa função

```
#/usr/bin/perl

use strict;
use warnings;

my $n = kodo();
printf("o numero foi %d",$n);

sub kodo
{
   my $numero = 315;
   return $numero;
}
```

ou uma array com vários valores

```
#/usr/bin/perl

use strict;
use warnings;

my @ns = kodo();
printf("o numero foi %d %d %d",$ns[0],$ns[1],$ns[2]);

sub kodo
{
   my @numeros = (315,200,50);
   return @numeros;
}
```

para a gente entrar com um valor como argumento para nossa função passamos os valores separado por virgula entre parênteses

```
#/usr/bin/perl
```

```
use strict;
use warnings;

kodo("fts",315);

sub kodo
{
}
```

para a gente manipular esses argumento dentro da nossa função usamos a array @_ ,sendo cada posição dela é um argumento que foi passado

```
#/usr/bin/perl

use strict;
use warnings;

kodo("kodo no kami","fts315");

sub kodo
{
    print("primeiro argumento: $_[0] \n");
    print("segundo argumento: $_[1] \n");
}
```

veja um exemplo de uma entrada de dois e o retorno da soma delas

```
#/usr/bin/perl

use strict;
use warnings;

my $valor = kodo_add(100,800);
print("$valor");

sub kodo_add
{
    my $temp = $_[0] + $_[1];
    return $temp;
}
```

no exemplo anterior ele só entra com dois valores podemos usar o foreach para somar todos os valores independente da quantidade de argumentos

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my $valor = kodo_add(100,800,10,80,66,75,12);
print("$valor");

sub kodo_add
{
    my $temp = 0;
    foreach my $v (@_)
    {
        $temp += $v;
    }
    return $temp;
}
```

a gente pode chamar a própria função dentro dela mesmo isso é chamado de recursivo e quando não tratado direito gera um loop infinito

```
#/usr/bin/perl

use strict;
use warnings;

kodo();

sub kodo
{
    print("função recursiva \n");
    kodo();
}
```

uma forma de tratar é entra com argumento para ela e incrementar essa valor a cada nova chamada e enviar para próxima ate um certo valor que para o loop

```
#/usr/bin/perl

use strict;
use warnings;

kodo(0);

sub kodo
{
    $atual = $_[0];
    if($atual >= 10)
    {
        return 0;
```

```
    }
    print("função recursiva $atual \n");
    $atual += 1;
    kodo($atual);
}
```

podemos usar a função eval para transformar uma string em um codigo que vai ser interpretado pelo compilador porem tome muito cuidado com ele isso é dependendo da logica do seu programa ele abre uma baita lacuna quando se trata na parte de segurança podendo deixar falhas no seu programa como a RCE

```
#/usr/bin/perl

use strict;
use warnings;

my $comando = "print('e ae isso é um string só que não')";
eval($comando);
```

no caso o eval pode ser usado para gera códigos dinamicamente para ser interpretado isso pode facilitar muito permitindo gerar strings com nomes de funções e interpretar elas (volto a repetir eval deve ser usado com muito cuidado principalmente se ele esta sendo concatenado com uma entrada do usuário, uso errado do eval na pior das hipóteses pode da acesso ao invasor na maquina apanas deformando a string com uma nova)

```
#/usr/bin/perl

my $codigo = 'use strict; use warnings; my $nome = "kodo no kami"; printf($nome);';
eval($codigo);
```

também é possível usar ponteiro para uma função bastando usar \& seguido do nome da função

```
#/usr/bin/perl

use strict;
use warnings;

my $fts = \&kodo;
&$fts("hacker fts315");

sub kodo
{
    print($_[0]);
}
```

isso é uma boa forma de passar funções para funções embora acho que seja pratico chamar as

funções dentro da outra diretamente porem em alguns casos quando temos algo abstrato ou polimórfico é necessário fazer dessa forma

```
#/usr/bin/perl

use strict;
use warnings;

kami(\&kodo,"a função kami é apenas um intermediário para chamar outra função");

sub kodo
{
    print($_[0]);
}

sub kami
{
    my $fun = $_[0];
    my $arg = $_[1];
    &$fun($arg);
}
```

4.1 – Criando módulos

criar um modulo é uma forma de reaproveitar funções em outros scripts sem precisar recriar a função apenas importar o modulo que tenha ela, boa parte das linguagens de programação são modular ou seja permite importação de modulos ou bibliotecas que contenha funções prontas, os módulos na linguagem perl usa extensão .pm e fica na mesma pasta do script ou em uma sub-pasta dele porem também podemos colocar eles em uma variavel de ambiente e sendo essa variavel de ambiente é dita como o mesmo diretorio que esta o script, a pasta que fica armazenado os modulos no windows quando instalado o active state é c:\perl\lib e no linux é /usr/share/perl/?, para importar um modulo como já vimos podemos usar o use ou require

```
#/usr/bin/perl

use strict;
require warnings;
```

para criar um modulo é muito facil basta a gente coloca as função em um arquivo com extensao .pm por exemplo kodo.pm e coloca ele na pasta que esta o script ou em uma pasta que seja diretório de ambiente,o nosso modulo também deve ter o numero 1 depois da ultima função se não o modulo não sera aceito

```
sub kodo_quadrado
{
   my $num = $_[0];
   $num *= $num;
```

```
   return $num;
}

1;
```

no codigo que vamos chamar a função do nosso modulo basta importar ele e pronto usar a função que queremos

```
#/usr/bin/perl

use strict;
use warnings;
use kodo;

my $n = kodo_quadrado(10);

print("$n");
```

quando o modulo esta em um sub-diretório por exemplo se o kodo.pm tivesse dentro do diretório fts a gente ia separar cada diretório usando duas vezes o dois pontos, podemos usar isso para separar cada modulo para uma coisa diferente

```
#/usr/bin/perl

use strict;
use warnings;
use fts::kodo;

my $n = kodo_quadrado(10);

print("$n");
```

a gente tambem pode chamar modulos dentro de modulos ou usar funções de outros modulos no nosso modulo e as vezes ate melhorando ele ou agilizando o nosso

```
use Digest::MD5 "md5_hex";

sub kodo_md5salt
{
	my $string = "pule coelhinho" . $_[0];
	return md5_hex($string);
}

1;
```

os comentários na linguagem perl tem mais uma utilidade que seria o uso do perldoc onde podemos

ver informações dos modulos, porem se a gente tentar usar o perldoc para o nosso modulo não vai funcionar isso por que a gente não documento ele

```
perldoc kodo
```

para documentar o nosso código usamos o comentário head1 seguido da palavra que vai separar, para facilitar a modificação use ele sempre no começo do código ou no final do código, se a gente usar o perldoc agora a gente ia ver o comentário

```
sub kodo_quadrado
{
   my $num = $_[0];
   $num *= $num;
   return $num;
}

1;

=head1 NOME

kodo.pm

=head1 SINOPSE

esse modulo serve para fazer calculos

=head1 EXEMPLO

my $k = kodo_quadrado(2);

=head1 AUTOR

kodo no kami

=cut
```

5.0 – Manipulando o tempo

a manipulação do tempo na linguagem perl não implica nas leis da física como a teoria da relatividade é algo bem mais simples como pegar a hora atual embora simples isso é bastante util acredite, para gente pegar o horário em segundos usamos a função time e atribuímos a uma variável, no caso ele vai armazenar os segundos deis de 1970 ate o segundo atual

```
#/usr/bin/perl

use strict;
use warnings;

my $tempo = time();

print($tempo);
```

podemos pegar o segundo, minutos, horas, dias, mes e ano com a função localtime, se a gente atribuir a função localtime para uma variável scalar é atribuído em forma de texto o dia da semana o mês o dia do mês a hora e o ano

```
#/usr/bin/perl

use strict;
use warnings;

my $tempo = localtime();

print($tempo);
```

se a gente atribui a função localtime para uma array a gente pode manipular esses valores separado em cada posição da array

```
#/usr/bin/perl

use strict;
use warnings;

my @tempo = localtime();

print("segundo:  $tempo[0] \n" .
"minuto:  $tempo[1] \n" .
"hora:  $tempo[2] \n" .
"dia:  $tempo[3] \n" .
"mes:  $tempo[4] \n" .
"ano:  $tempo[5] \n");
```

podemos usar a função gmtime no lugar de localtime a diferença das duas que a localtime pega o tempo de acordo com o fuso-horario do computador já o gmtime pega o fuso-horario principal de acordo com greenwich

```
#/usr/bin/perl

use strict;
use warnings;
```

```
my @t_gm = gmtime();
my @t_local = localtime();

print("gmtime: $t_gm[2]:$t_gm[1]  \n");
print("localtime: $t_local[2]:$t_local[1]  \n");
```

com a função strftime do modulo POSIX podemos criar ou formar um tempo, bastando passa a string de formatação e 6 outros argumentos sendo eles segundos (%S), minutos (%M), hora (%H), dia (%d), mês (%m) e o ano (%y)

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my @novo = (0,15,3,10,5,115);
my $tempo = strftime("%H:%M:%S %d/%m/%y",@novo);

print("data criada: $tempo");
```

da mesma forma a gente poderia passar os argumentos do tempo sem precisar de uma array

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $tempo = strftime("%H:%M:%S %d/%m/%y",0,15,3,10,5,115);

print("data criada: $tempo");
```

uma função parecida com anterior é o mktime também do modulo POSIX, porem ao invés de retornar uma string formatada retorna o tempo em segundos igual a função time

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

my $novo = mktime(0,15,3,10,5,115);

print($novo);
```

5.1 – Manipulando arquivos

além da manipulação do tempo podemos manipular arquivos e diretórios pela linguagem perl entre outras coisas, para a gente manipular um arquivo temos que abrir o arquivo com permissão de leitura ou escrita depois ler ele podendo ser ele todo como também linha por linha ou carácter por carácter, para abrir um arquivo usamos a função open passamos como argumento um handle que pode ser qualquer nome e como já explicado handles ou descritores são especie de variáveis, a permissão que pode ser para ler o arquivo ou escrever no arquivo sendo ela para leitura usamos o simbolo de maior que e para escrita usamos o simbolo de menor que, e o ultimo argumento que seria o arquivo que vamos ler ou escrever, quando a gente usa permissão de escrita se o arquivo não existir ele é criado porem isso não ocorre na permissão de leitura apenas na de escrita

```
#/usr/bin/perl

use strict;
use warnings;

open(ARQUIVO,"<","kodo.txt");
```

o mesmo vale para escrita só mudando a permissão no arquivos

```
#/usr/bin/perl

use strict;
use warnings;

open(ARQUIVO,">","kodo.txt");
```

também existe a permissão de concatenação que seria a permissão de escrita usada duas vezes, essa permissão é parecida com a de escrita a diferença que ela não apaga o conteúdo do arquivo para escrever um novo apenas escreve no final dele assim juntando o novo com o antigo

```
#/usr/bin/perl

use strict;
use warnings;

open(ARQUIVO,">>","kodo.txt");
```

também podemos usar apenas dois argumentos bastando concatenar a permissão junto com o arquivo

```
#/usr/bin/perl
```

```
use strict;
use warnings;

open(ARQUIVO,">kodo.txt");
```

além de abrir o arquivo podemos fechar o arquivo aberto com a função close e passando como argumento para ela o handle do arquivo aberto, precisamos fechar um arquivo aberto por dois motivos o primeiro deles é que não é possivel abrir o mesmo arquivo por vários programas ao mesmo tempo então podemos fechar o arquivo quando terminar para que outro programa possa ler ou escrever nele e isso também vale para mover ou ate remover ele, o segundo motivo que os dados é apenas armazenado no arquivo quando ele é fechado para evitar vários problemas ou seja enquanto não for fechado não sera salvo no arquivo, então sempre quando não for mais armazenar nada no arquivo feche ele

```
#/usr/bin/perl

use strict;
use warnings;

open(ARQUIVO,">kodo.txt");
close(ARQUIVO);
```

a forma mais rapida de ler um arquivo é simplesmente atribuir o descritor para uma array e depois ler essa array

```
#/usr/bin/perl

use strict;
use warnings;

my @dados;

open(ARQUIVO,"<kodo.txt");
@dados = <ARQUIVO>;
close(ARQUIVO);

foreach my $linha (@dados)
{
    print $linha;
}
```

a gente também poderia usar o descritor em um loop e usar a variavel $_ para cada linha do arquivo que seria equivalente a cada loop

```
#/usr/bin/perl

use strict;
```

```
use warnings;

open(ARQUIVO,"<kodo.txt");

while(<ARQUIVO>)
{
    print("$_ ") ;
}

close(ARQUIVO);
```

também podemos usar a função read para ler os dados de um arquivo porem diferente das anteriores que le linha por linha a função read ler uma quantidade de caracteres especifica e armazena em uma variável, o primeiro argumento da função read é o descritor do arquivo aberto, o segundo argumento dela é a variável onde vai ser armazenado os dados, o terceiro é quantidade de caracteres que vai ser lida no arquivo, e o ultimo é offset ou seja a posição da onde vai começar a ler

```
#/usr/bin/perl

use strict;
use warnings;

my $primeiro;
my $segundo;

open(ARQUIVO,"<kodo.txt");

read(ARQUIVO,$primeiro,10,0);
read(ARQUIVO,$segundo,10,11);

close(ARQUIVO);

print("primeiro 10 caracteres: $primeiro \n");
print("mais 10 caracteres: $segundo \n");
```

podemos ler uma única linha usando a função readline e nela basta passar como argumento o descritor do arquivo e armazenar em uma variável, a próxima chamada da função readline sera a próxima linha e assim sucessivamente

```
#/usr/bin/perl

use strict;
use warnings;

my $linha;

open(ARQUIVO,"<kodo.txt");
```

```
while($linha = readline(ARQUIVO))
{
    print($linha);
}

close(ARQUIVO);
```

também é possível ler caracteres por caracteres usando a função getc

```
#/usr/bin/perl

use strict;
use warnings;

my $caracter;

open(ARQUIVO,"<kodo.txt");

while($caracter = getc(ARQUIVO))
{
    print($caracter);
}

close(ARQUIVO);
```

podemos especificar que é um arquivo binario usando a função binmode seguido do descritor

```
#/usr/bin/perl

use strict;
use warnings;

my $byte;

open(ARQUIVO,"<kodo.exe");
binmode(ARQUIVO);

while($byte = getc(ARQUIVO))
{
    printf("%x ",ord($byte));
}

close(ARQUIVO);
```

com a função eof é possível ver se descritor esta apontando para o final do arquivo retornando verdadeiro se estiver e falso se não

```
#/usr/bin/perl

use strict;
use warnings;

open(ARQUIVO,"<kodo.txt");

while($caracter = getc(ARQUIVO))
{
    if(eof(ARQUIVO))
    {
        print("fim do arquivo");
    }
}

close(ARQUIVO);
```

com a função tell podemos retornar a posição atual que o descritor esta apontando

```
#/usr/bin/perl

use strict;
use warnings;

my $pos;

open(ARQUIVO,"<kodo.txt");

while(readline(ARQUIVO))
{
    $pos = tell(ARQUIVO);
    print("$pos ");
}

close(ARQUIVO);
```

podemos mudar a posição atual com a função seek, passamos como argumento para ele o descritor seguido do offset e por fim a posição incial sendo 0 para o inicio do arquivo, 1 para a posição atual e 2 para final do arquivo

```
#/usr/bin/perl

use strict;
use warnings;

my $pos;

open(ARQUIVO,"<kodo.txt");
```

```
seek(ARQUIVO,10,0);

$pos = getc(ARQUIVO);

close(ARQUIVO);
```

a gente pode usar o seek para ir para ultima posição no arquivo e usar o tell para pegar a ultima posição e com isso descobrir o tamanho do arquivo

```
#/usr/bin/perl

use strict;
use warnings;

my $tam;

open(ARQUIVO,"<kodo.txt");

seek(ARQUIVO,0,2);
$tam = tell(ARQUIVO);

print($tam);

close(ARQUIVO);
```

para a gente escrever em um arquivo basta usar a permissão de escrita no lugar da leitura depois usamos o print passamos como argumento para ele o descritor do arquivo seguido da string que sera armazenada porem não separamos elas com virgula e nem nada disso apenas o espaço mesmo

```
#/usr/bin/perl

use strict;
use warnings;

open(ARQUIVO,">kodo.txt");

print(ARQUIVO "saida para o arquivo");

close(ARQUIVO);
```

tambem é possível faz o mesmo usando o printf

```
#/usr/bin/perl

use strict;
use warnings;
```

```
my @tempo = localtime();

open(ARQUIVO,">kodo.txt");

printf(ARQUIVO "%d:%d:%d", $tempo[2], $tempo[1], $tempo[0]);

close(ARQUIVO);
```

é possível usar a permissão de concatenar para não apagar o texto no arquivo e sim salvar depois dele

```
#/usr/bin/perl

use strict;
use warnings;

my @tempo = localtime();

open(ARQUIVO,">>kodo.txt");

printf(ARQUIVO "script executado %d:%d:%d \n", $tempo[2], $tempo[1], $tempo[0]);

close(ARQUIVO);
```

outra maneira de abrir, ler e escrever em um arquivo é usando o modulo IO::File, primeiramente a gente deve instancia o objeto com o operador new bastando atribuir ele para uma variavel (vamos aprender mais sobre orientação objeto alguns capítulos a frente)

```
#/usr/bin/perl

use strict;
use warnings;
use IO::File;

my $arquivo = new IO::File;
```

agora usamos o método open no objeto instanciado e nele passamos como argumento o arquivo que vamos abrir seguido da permissão sendo ela “r” para leitura “w” para escrita e “a” para concatenar

```
#/usr/bin/perl

use strict;
use warnings;
use IO::File;

my $arquivo = new IO::File;
```

```
$arquivo->open("fts.txt","r");
```

tambem podemos usar o método close para fechar ele

```
#/usr/bin/perl

use strict;
use warnings;
use IO::File;

my $arquivo = new IO::File;
$arquivo->open("fts.txt","r");
$arquivo->close();
```

a gente pode usar o método read para ler o arquivo nele passamos como argumento a variavel que vai ser armazenado, a quantidade de dados que sera lida e o offset inicial

```
#/usr/bin/perl

use strict;
use warnings;
use IO::File;

my $dados;

my $arquivo = new IO::File;
$arquivo->open("fts.txt","r");
$arquivo->read($dados,10,0);
$arquivo->close();

print($dados);
```

para escrever no arquivo usamos o metodo write

```
#/usr/bin/perl

use strict;
use warnings;
use IO::File;


my $arquivo = new IO::File;
$arquivo->open("fts.txt","w");
$arquivo->write("arquivo de teste");
$arquivo->close();
```

podemos pegar a posição atual com o método getpos

```
#/usr/bin/perl

use strict;
use warnings;
use IO::File;

my $dados;
my $pos;

my $arquivo = new IO::File;
$arquivo->open("fts.txt","r");
$arquivo->read($dados,10,0);
$pos = $arquivo->getpos();
$arquivo->close();

print($pos);
```

a gente pode remover um arquivo com a função unlink e como argumento para ela o arquivo que sera removido

```
#/usr/bin/perl

use strict;
use warnings;

unlink("kodo.txt");
```

como já sabemos se a gente usar a permissão de escrita ou de concatenar ele cria o arquivo caso não exista porem podemos usar a função creat do modulo POSIX para criar o arquivo, nessa função basta passar como argumento o arquivo que vai ser criado seguido da flag

```
#/usr/bin/perl

use strict;
use warnings;
use POSIX;

creat("fts.txt",O_CREAT);
```

5.2 – Manipulando diretórios

para manipular um diretorio é quase mesmo coisa que manipular um arquivo ou seja temos que abrir o diretorio antes de ler ele, para abrir um diretorio usamos a função opendir e ela é parecida

com a função open ou seja passamos como argumento um descritor e o diretorio que vamos abrir

```
#/usr/bin/perl

use strict;
use warnings;

opendir(PASTA,"/home/kodo");
```

tambem usamos a função closedir para fechar o diretorio

```
#/usr/bin/perl

use strict;
use warnings;

opendir(PASTA,"/home/kodo");
closedir(PASTA);
```

para ler o diretorio ou seja os arquivos e diretorios nele usamos a função readdir passamos como argumento para ela o descritor

```
#/usr/bin/perl

use strict;
use warnings;

opendir(PASTA,"/home/kodo/Desktop");
my @dirs = readdir(PASTA);
closedir(PASTA);

foreach my $d (@dirs)
{
    print("$d \n");
}
```

é possível ler um diretório diretamente em um loop com o readdir

```
#/usr/bin/perl

use strict;
use warnings;

opendir(PASTA,"/home/kodo/Desktop");

while(readdir(PASTA))
```

```
{
    print("$_ \n");
}

closedir(PASTA);
```

também podemos usar o objeto IO::Dir para manipular diretorio ele tambem é bem parecido com IO::File, no caso para usar ele basta a gente instanciar o objeto

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Dir;

my $pasta = new IO::Dir;
```

usamos o método open para abrir o diretório e o close para fechar

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Dir;

my $pasta = new IO::Dir;
$pasta->open("/home/kodo/Desktop");
$pasta->close();
```

também usamos o método read para ler os diretórios

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Dir;

my $pasta = new IO::Dir;
$pasta->open("/home/kodo/Desktop");
my @past = $pasta->read();
$pasta->close();

foreach my $p(@past)
{
    print("$p \n");
}
```

podemos criar um diretório usando a função mkdir seguido do diretório ou endereço dele

```
#/usr/bin/perl

use strict;
use warnings;

mkdir("fts");
```

também podemos remover um diretório com a função rmdir

```
#/usr/bin/perl

use strict;
use warnings;

rmdir("fts");
```

é possível diferenciar se é um arquivo usando o operador -f seguido do arquivo ou diretório, se for um arquivo retorna verdadeiro se não retorna falso

```
#/usr/bin/perl

use strict;
use warnings;

if(-f "/home/kodo/Desktop/kodo.txt")
{
    print("é um arquivo");
}
else
{
    print("não é um arquivo");
}
```

com o operador -d retorna verdadeiro se for um diretorio

```
#/usr/bin/perl

use strict;
use warnings;

if(-d "/home/kodo/Desktop")
{
    print("é um diretorio");
}
```

```
else
{
    print("não é um diretorio");
}
```

o -z retorna verdadeiro caso o arquivo esta vazio

```
#/usr/bin/perl

use strict;
use warnings;

if(-z "/home/kodo/Desktop/kodo.txt")
{
    print("esta vazio");
}
else
{
    print("tem coelho nesse mato (ou camelo)");
}
```

com a função stat podemos armazenar as informações do arquivo em uma array, sendo elas a dev, inode, permissão, numeros de hardlinks, usuario id, grupo id, device indetificador, tamanho do arquivo, data do ultimo acesso, data da ultima modificação, data do inode, tamanho do bloco do sistema, tamanho do bloco

```
#/usr/bin/perl

use strict;
use warnings;

my @info = stat("/home/kodo/Desktop/kodo.txt");

foreach my $i (@info)
{
    print("$i \n");
}
```

lembrar cada posição do stat é complicado para facilitar podemos declarar o modulo File::stat e usar a função stat dele

```
#/usr/bin/perl

use strict;
use warnings;
use File::stat "stat";
```

```
my $info = stat("/home/kodo/Desktop/kodo.txt");

print("tamanho: ". $info->size ."\n");
print("ultimo acesso:". $info->atime ."\n");
print("ultima modificação:". $info->mtime ."\n");
```

bom galera a gente já sabe manipular arquivos e diretorios isso quer dizer que podemos criar scripts para manipular grande quantidade de arquivo isso com base em heurística ou seja você pode criar scripts para descobrir se um arquivo é uma imagem apenas olhando os primeiros bytes ou se é um arquivo de som ou video e qual formato dele, você pode criar scripts para analise malwares (virus, trojan), você pode criar scripts para ocultar arquivos dentro de outro arquivos ou ate descobrir e extrair arquivos dentro de outros arquivos e milhares de outras coisas

5.3 – Manipulando terminal

outra coisa que também podemos manipular via scripts perl é o próprio terminal podendo usar qualquer comando que seria usado nele ou seja podemos adicionar no nosso script tudo que o sistema consegue fazer pelo terminal (isso em linux então você tem uma infinidade de coisas), a forma mais simples seria usar a função system e passar como argumento o comando que a gente ia usar no terminal (lembrando que esses comandos são baseado no sistema ou nos programas instalados no sistema e pode variar de sistema para sistema exemplo no windows podemos usar o comando dir para listar os diretorios no linux usamos o comando ls);

```
#/usr/bin/perl

use strict;
use warnings;

system("ls");
```

uma forma da gente diferenciar um sistema operacional do outro é usando a variável $^O que retorna uma string dizendo qual é o sistema com isso podemos usar ele para facilitar em comandos ou funções que só pode ser usados em um sistema

```
#/usr/bin/perl

use strict;
use warnings;

if($^O eq "linux")
{
    system("ls");
}
else
{
    system("dir");
}
```

a função system não é muito pratica isso por que não retorna a saida apenas imprime ela na tela ou seja não tem como a gente manipular a saida pelo menos diretamente na função, uma gambiarra que seria possivel fazer isso é salvar a saida para um arquivo e depois ler esse arquivo

```
#/usr/bin/perl

use strict;
use warnings;

system("ls > kodo.txt");
open(ARQ,"kodo.txt");
my @saida = <ARQ>;
close(ARQ);

foreach my $linhas (@saida)
{
    print("$linhas  \n");
}
```

tambem é possível usando a função exec que executa programas do sistema

```
#/usr/bin/perl

use strict;
use warnings;

exec("vlc");
```

uma forma mais simples que o system ou exec é usando pipes que são encanamento que liga uma saida de programa para uma entrada de outro programa (miss mario kkk), existem muitas formas de usar pipes na linguagem perl a mais simples é com a função open só que ao invés da gente especificar um arquivo usamos um comando que seria usado no terminal colocamos o sinal de pipe no final dele, depois basta ler o descritor dele normalmente como um arquivo porem o que vai ter nele é a saída do comando

```
#/usr/bin/perl

use strict;
use warnings;

open(MARIO, "ls |");

while(readline(MARIO))
{
    print("$_");
}
```

```
close(MARIO);
```

5.4 – Manipulando socket

sockets pode ser um assunto bem complexo e longo por isso não vou me aprofundar tanto nele, no caso sockets são meios de comunicação entre uma maquina e outra, eles são usados para programação em rede ou ate mesmo para comunicação entre um programa e outro podendo esta em rede ou na mesma maquina,o socket é dividido em duas partes sendo eles o socket servidor que fica aguardando uma comunicação e o socket cliente que se conecta ao socket servidor, depois que a conexão entre eles é estabelecida a comunicação pode ser de ambos os lados ou seja tanto o cliente quanto o servidor pode mandar dados para outro lado porem a forma que eles vão se comunicar deve seguir um padrão especifico isso é chamado de protoloco de rede e sem esse protocolo mesmo que tenha comunicação outro lado não vai saber o que tem que ser feito, toda comunicação entre duas ou mais maquinas segue um protocolo especifico um bom exemplo disso é seu navegador que vai ler uma pagina de um site que esta em um servidor web, no caso seu navegador depois que estabelece a comunicação envia informações especifica para o servidor web para dizer qual pagina ele deve enviar e a comunicação entre eles é por um socket cliente do seu navegador e o socket servidor no servidor web, se você sabe como a comunicação daquele protocolo funciona você pode refazer ela usando um socket e assim refazendo aquele protocolo, como eu disse antes não vou me aprofundar tanto em sockets nesse ebook sendo que seria ate possível escrever um outro ebook só para falar de sockets e protocolos (quem sabe futuramente ne), deixando a teoria um pouco de lado e indo pouco mais para pratica vamos pelo menos aprender fazer dois sockets que seria o servidor e o cliente lembrando mesmo que isso ainda seja um pouco superficial da para fazer muita coisa, para manipular os sockets em perl podemos usar o modulo Socket ou IO::Socket porem esses dois são bem brutos então vamos usar o modulo IO::Socket::INET sendo ele bem mais simples

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;
```

depois de declarar o modulo do socket vamos decidir se vamos fazer o servidor ou cliente primeiro isso é tanto pode ser feito o servidor quanto o cliente da mesma forma, alem do mais o que esse servidor vai fazer exatamente e como sera o nosso protocolo?, no caso o que vamos fazer é o servidor de tempo quando alguém estabelecer conexao no nosso servidor e enviar a palavra kami nosso servidor vai enviar o horário atual simples não?, para começar instanciamos o objeto do nosso socket e passamos como argumento hashs para ele sendo eles o LocalPort com a porta que vamos usar que pode ser entre 0-65535 porem recomendado usar portas altas acima de 5000, isso por que as portas baixas são usadas por padrao pelo sistema ou protocolos padrões, tambem passamo como argumento a hash com o tipo de protocolo que é o Proto podendo ser tcp ou udp como valor, tambem temos que passar como argumento a hash Listen seguido do valor 1 que diz que o socket vai ficar em escuta

```
#/usr/bin/perl

use strict;
```

```
use warnings;
use IO::Socket::INET;

my $servidor = new IO::Socket::INET(LocalPort => 20315, Proto => "tcp",Listen=> 1);
```

teoricamente o socket tá configurado agora falta a gente usar o método accept para ficar esperando e aceitar a comunicação, quando usamos ele o script vai ficar travado naquela parte ate que alguém se conecte quando isso acontecer vai continuar executando o resto do script, também atribuímos o método accept para uma outra variável ou seja se tiver mais de uma comunicação cada variável é uma conexão diferente

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;

my $servidor = new IO::Socket::INET(LocalPort => 20315, Proto => "tcp",Listen=> 1);

print("esperando a comunicação....\n");

my $novo = $servidor->accept();

print("comunicação estabelecida....\n");

print("fim da comunicação....\n");
```

o socket já esta pronto para estabelecer a comunicação (a gente poderia conectar ao nosso script usando o telnet ou ate o ncat para testar), porem como o nosso exemplo anterior não faz nada depois que é aceito a comunicação o script fecha ou seja a comunicação e fechada quando o script termina e para evitar isso podemos colocar ele em um loop assim não vai fechar

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;

my $servidor = new IO::Socket::INET(LocalPort => 20315, Proto => "tcp",Listen=> 1);

print("esperando a comunicação....\n");

my $novo = $servidor->accept();

print("comunicação estabelecida....\n");

while(1 == 1)
{
```

```
}

print("fim da comunicação....\n");
```

para a gente ler os dados do cliente usamos o método recv passamos como argumento a variável que vamos armazenar e quantidade de bytes que vamos receber, quando perl encontra o método recv acontece igual o accept fica parado esperando algum dado chegar

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;

my $servidor = new IO::Socket::INET(LocalPort => 20315, Proto => "tcp",Listen=> 1);

print("esperando a comunicação....\n");

my $dados;
my $novo = $servidor->accept();

print("comunicação estabelecida....\n");

while(1 == 1)
{
    $novo->recv($dados,100);
    print("$dados");
}

print("fim da comunicação....\n");
```

para enviar usamos o metodo send bastando passar como argumento a string, como dito antes nosso servidor vai esperar a palavra kami quando ela chegar vamos enviar o horario atual

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;

my $servidor = new IO::Socket::INET(LocalPort => 20315, Proto => "tcp",Listen=> 1);

print("esperando a comunicação....\n");

my $dados;
my $novo = $servidor->accept();

print("comunicação estabelecida....\n");
```

```
while(1 == 1)
{
    $novo->recv($dados,4);
    if($dados eq "kami")
    {
        my @tempo = localtime();
        my $tempo2 = "hora: " . $tempo[2] . ":" . $tempo[1]  . ":"  . $tempo[0] . "\n";
        $novo->send($tempo2);
        last;
    }
}

print("fim da comunicação....\n");
```

o script servidor esta pronto agora vamos criar o script que vai ser o cliente para isso vamos instancia um socket igual o anterior porem nesse script nos argumentos vamos usar o PeerHost, PeerPort e Proto, no PeerHost a gente coloca o IP do alvo (caso seja a própria maquina podemos usar 127.0.0.1), o PeerPort é a porta que deve ser mesma que definimos no script anterior e o mesmo vale para o Proto

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;

my $cliente = new IO::Socket::INET(PeerHost =>"127.0.0.1", PeerPort=>20315, Proto=> "tcp");
```

para enviar e receber os dados é a mesma coisa do servidor no caso é send e recv, no nosso protocolo pelo menos para essa logica não precisamos deixar ele em um loop só precisamos enviar o kami e ler o horário, porem se o protocolo fosse diferente ou ate um chat ou alguma coisa desse gênero ai seria necessário

```
#/usr/bin/perl

use strict;
use warnings;
use IO::Socket::INET;

my $dados;
my $cliente = new IO::Socket::INET(PeerHost =>"127.0.0.1", PeerPort=>20315, Proto=> "tcp");
$cliente->send("kami");
$cliente->recv($dados,100);

print($dados);
```

como eu disse antes não vamos nos aprofundar em socket nesse ebook talvez futuramente eu

modifique ele e adicione novos conteúdos

5.5 – Manipulando Request HTTP

existem diversos módulos na linguagem perl então não precisamos manipular uma coisa a nivel de sockets para criar um script que automatiza uma busca de um texto em um site ou ate que baixe um arquivo de um site embora o mesmo possa ser feito usando usando sockets porem isso não é muito pratico como podemos ver na parte anterior que para a gente fazer um simples cliente temos que conhecer o protocolo que estamos trabalhando, essa busca que fazemos em um site é chamada requisição ou request e existem modulos na linguagem perl que permite fazer essa requisição sem a gente precisar mexer diretamente com socket já que o modulo faz isso por traz dos panos, no caso o uso desses modulos retorna na maioria das vezes apenas o codigo html do site ou seja com base nele você poderia criar um navegador apenas interpretando o codigo e formando ele em uma parte grafica ou criando um script que automatiza uma determinada busca naquele site ou ate um bot, o modulo mais simples para fazer um request para o protocolo http é o modulo LWP::Simple com a função get que basta a gente passar como argumento para ele a url do site que vamos fazer a requisição depois atribuir ele para uma variavel e manipular o codigo html por ela

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::Simple;

my $codigo = get("http://eofcommunity.com/forum");

print($codigo);
```

podemos baixar a pagina ou ate um arquivo com a função getstore passando como argumento a url seguido do nome do arquivos que sera salvo

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::Simple;

getstore("http://eofcommunity.com/forum","index.html");
```

o modulo LWP::Simple é muito simples apesar de ter mais algumas funções não citadas recomendo o uso do LWP::UserAgent que da para definir mais informação como agent, proxy, cookie alem de fazer outros tipos de request como o metodo POST

```
#/usr/bin/perl

use strict;
```

```
use warnings;
use LWP::UserAgent;
```

para usar o LWP::UserAgent temos que instanciar o objeto usando o operador new e atribuído o mesmo para uma variável

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $kodo = new LWP::UserAgent;
```

depois basta a gente usar o metodo get para fazer a requisição em um site e atribuir o mesmo para uma variavel, no caso ele retorna um objeto HTTP::Response

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $kodo = new LWP::UserAgent;
my $kami = $kodo->get("http://eofcommunity.com/forum");
```

podemos usar o método content do objeto HTTP::Response para mostrar o codigo html

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $kodo = new LWP::UserAgent;
my $kami = $kodo->get("http://eofcommunity.com/forum");
print($kami->content);
```

antes da gente fazer a requisição podemos usar o o método agent para mudar o agente, isso pode ser útil já que muitos websites usa user-agent para descobrir qual o nosso sistema operacional ou ate mesmo nosso navegador

```
#/usr/bin/perl

use strict;
```

```
use warnings;
use LWP::UserAgent;

my $kodo = new LWP::UserAgent;
$kodo->agent("sou um bot do google, deixe-me entrar XD");
my $kami = $kodo->get("http://eofcommunity.com/forum");
print($kami->content);
```

também é possível fazer uma requisição passando por um proxy usando o metodo proxy, nele especificamos o tipo de proxy seguido da url do proxy

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $kodo = new LWP::UserAgent;
$kodo->proxy("http","http://192.168.1.2:315");
my $kami = $kodo->get("http://eofcommunity.com/forum");
print($kami->content);
```

além do método get podemos usar o método post e passar determinados parametros que seria passado em um formulario ou seja podemos automatizar uma pagina de formulario, para isso usamos o metodo post no lugar do get e o segundo argumento dele depois da url os parametros que são equivalente a hashs entre colchete

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;

my $kodo = new LWP::UserAgent;

my $kami = $kodo->post("http://192.168.1.1/kodo.php",[usuario=>"kodo", senha=> "123456"]);
print($kami->content);
```

no metodo get tambem é possível passar parâmetros igual o post porem é tudo na url sendo um interrogação no final e eles separados por &

```
#/usr/bin/perl

use strict;
use warnings;
use LWP::UserAgent;
```

```
my $kodo = new LWP::UserAgent;

my $kami = $kodo->get("http://192.168.1.1/kodo.php?usuario=kodo&senha=123456");
print($kami->content);
```

além do LWP::UserAgent temos o meu preferido que é o WWW::Mechanize, eu usei esse modulo para criar toda requisição do script “fts anitube down”, ele seria uma melhoria do LWP sendo que não precisamos atribuir a saida get para outro variavel para manipular o content pelo WWW::Mechanize isso faz parte do próprio modulo ele tambem pode buscar links e outros tipos de inputs por ordem nome ou ate atributo especifico como id ou class, envia formulario da pagina pelo próprio request, tambem mantem sessão e suporta SSL entre outras coisas, o uso dele tambem é bem parecido com anterior no caso não vou abordar ele nesse ebook por hora

```
#/usr/bin/perl

use strict;
use warnings;
use WWW::Mechanize;

my $kodo = new WWW::Mechanize;
$kodo->get("http://eofcommunity.com/forum");
print($kodo->content);
```

6.0 – Orientação a objeto

o paradigma da orientação a objeto é tratado todas as funções (métodos) e variáveis (atributos) que pertence a um determinado objeto como um todo ou seja uma variável em um objeto faz parte daquele objeto e é tratada como próprio objeto, a orientação a objeto permite acessar variáveis e atributos do próprio objeto e permite que métodos acesse outros métodos do próprio objeto, cada novo objeto instanciado é idêntico ao outro porem não é o mesmo ou seja um atributo em um objeto não interfere em outro objeto semelhante apenas nele mesmo, a orientação a objeto tambem permite herdar atributos e métodos de outro objeto assim permitindo ganhar novas funcionalidades com isso, a orientação a objeto também permite abstrair determinadas informações

6.1 – Orientação a objeto: Pacotes

na linguagem perl a estrutura que define um objeto é um pacote e não uma classe como muitas outras linguagens (c++, java, python, pascal), para a gente criar um pacote usamos o palavra package seguido do nome do nosso pacote

```
#/usr/bin/perl

use strict;
use warnings;

package kodo;
```

depois do pacote todo código pertence aquele pacote ate que seja criado outro pacote ai os próximos vai pertence ao outro e não a ele

```
#/usr/bin/perl

use strict;
use warnings;

package kodo;

package fts;
```

como dito no começo desse ebook a linguagem perl não usa funções principais igual a linguagem C porem usa um pacote chamado main

```
#/usr/bin/perl

package main;

use strict;
use warnings;

package kodo;
```

podemos usar a constante __PACKAGE__ para saber o nome do pacote atualiza

```
#/usr/bin/perl

package main;

use strict;
use warnings;

print(__PACKAGE__);
```

6.2 – Orientação a objeto: Construtores

na linguagem perl a gente deve criar um metodo que sera o construtor embora muitas linguagens o construtor é o new (tirando o pascal que é o create kkk), na linguagem perl podemos criar construtor com qualquer nome mais vamos deixar o new por padrao, para criar um construtor no nosso pacote primeiro vamos criar um método com o nome do nosso construtor

```
#/usr/bin/perl
```

```
use strict;
use warnings;

package kodo;

sub new
{
}
```

usamos a função shift para pegar o ponteiro do próprio objeto (em muitas linguagens isso seria o this ou self)

```
#/usr/bin/perl

use strict;
use warnings;

package kodo;

sub new
{
    my $this = shift;
}
```

agora atribuímos uma estrutura para uma variável essa estrutura são as variaveis são os atributos e as variáveis iniciada

```
#/usr/bin/perl

use strict;
use warnings;

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
}
```

para terminar usamos a função bless que transforma em nosso objeto e permite que os demais metodos receba as instruções correta, passamos como argumento para ele a variável que tem os valores iniciado e o ponteiro this

```
#/usr/bin/perl

use strict;
```

```
use warnings;

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}
```

nosso construtor já esta pronto agora já podemos instanciar um objeto do nosso pacote, para isso basta atribuir ele para uma variável usando em conjunto o operador new

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo;

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}
```

outra forma de instanciar o objeto seria usar o método new depois do nome do pacote

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = kodo->new;

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}
```

por padrao vamos sempre usar o metodo new para constutor porem pode ser usado qualquer outro nome

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = kamehameha kodo;

package kodo;

sub kamehameha
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}
```

como podemos ver o nosso objeto instanciado seria toda estrutura do pacote e todo objeto criado desse pacote é uma copia do mesmo porem não interfere nos outro instanciados daquele pacote

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo;
my $objeto_diferente = new kodo;

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}
```

6.3 – Orientação a objeto: Atributos e Métodos

os atributos e métodos na orientação a objeto são nada mais nada menos que variáveis e funções a única diferença entre eles que os atributos e métodos diferente das variáveis e funções são limitados para dentro do objeto ou seja se o objeto não existe os atributos e metodos tambem não, para criar um atributo basta a gente declarar os atributos no construtor

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my $objeto = new kodo;

package kodo;

sub new
{
    my $this = shift;

    my $kami;

    my $ini = {};
    bless($ini,$this);
}
```

só de declarar ela não indica que a gente vai conseguir usar ela para isso temos que especifica ele dentro daquela estrutura para ser enviada pela a função bless para as demais, também precisamos colocar um valor inicial na variável para evitar erros embora isso seja opcional se tentar usar esse atributo sem setar valor nenhum vai retornar erro, dentro da estrutura usamos hashs podemos colocar qualquer nome embora seja recomendado usar o mesmo nome da variável e no valor passamos a variável,

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo;

package kodo;

sub new
{
    my $this = shift;

    my $kami = 0;

    my $ini = { kami => $kami };
    bless($ini,$this);
}
```

podemos usar o atributo pelo objeto como se fosse uma hash

```
#/usr/bin/perl
```

```
use strict;
use warnings;

my $objeto = new kodo;

$objeto->{kami} = 315;
print($objeto->{kami}  . "\n");

package kodo;

sub new
{
    my $this = shift;

    my $kami = 0;

    my $ini = { kami => $kami };
    bless($ini,$this);
}
```

para atribuir valores pelos atributos basta receber eles pela array @_ e atribuir ela para uma variavel tipo hash depois manipular os valores com base na entrada, podemos usar a condição if com a função defined seguido da entrada do argumento ou seja se o argumento existe basta atribuir ele para variável que sera enviada para bless

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo(kami => 1000);

print($objeto->{kami}  . "\n");

package kodo;

sub new
{
    my $this = shift;
    my %argu = @_;

    my $kami = 0;

    if(defined($argu{kami}))
    {
        $kami = $argu{kami};
    }

    my $ini = { kami => $kami };
    bless($ini,$this);
```

```
}
```

para a gente criar metodos é mais simples bastando criar a funções

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo();

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}

sub fts
{
    print("isso é um metodo");
}
```

podemos chamar os métodos diretamente do nosso objeto

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo();
$objeto->fts();

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}

sub fts
{
    print("isso é um metodo");
}
```

para a gente acessar os atributos passado para o metodo usamos o shift como no construtor, atribuímos ele para uma variável e por ela manipulamos os atributos passado como se fosse uma hash

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo();
$objeto->{nome} = "kodo no kami";
$objeto->fts();

package kodo;

sub new
{
    my $this = shift;
    my $nome = "";
    my $ini = {nome => $nome};
    bless($ini,$this);
}

sub fts
{
    my $this = shift;
    print("seu nome é " . $this->{nome});
}
```

também é possivel pegar argumentos passado para um metodo com a array @_

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo();
$objeto->fts("kodo no kami");

package kodo;

sub new
{
    my $this = shift;
    my $nome = "";
    my $ini = {nome => $nome};
    bless($ini,$this);
}
```

```
sub fts
{
    my $this = shift;
    my @argu = @_;
    print("seu nome é " . $argu[0]);
}
```

também podemos atribuir a array @_ para uma variável hash para manipular como tal

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo();
$objeto->fts(nome => "kodo no kami");

package kodo;

sub new
{
    my $this = shift;
    my $nome = "";
    my $ini = {nome => $nome};
    bless($ini,$this);
}

sub fts
{
    my $this = shift;
    my %argu = @_;
    print("seu nome é " . $argu{nome});
}
```

na orientação objeto é muito usado os métodos set e get para atribuir ou ler um determinado atributo isso é uma parte de encapsulamento embora não seja tao pratico criar um forma de encapsular em perl manualmente porem ainda podemos usar o set o get para atribuir

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kodo();
$objeto->setNome("kodo no kami");
print $objeto->getNome();
```

```
package kodo;

sub new
{
    my $this = shift;
    my $nome = "";
    my $ini = {nome => $nome};
    bless($ini,$this);
}

sub getNome
{
    my $this = shift;
    return $this->{nome};
}

sub setNome
{
    my $this = shift;
   @argu = @_;
    $this->{nome} = $argu[0];
}
```

6.4 – Orientação a objeto: Herança

herança permite que determinados objeto herde as mesmas características e funcionalidade de outro objeto como métodos e atributos, a orientação a objeto na linguagem perl permite heranças múltiplas ou seja você pode herdar mais de um objeto e suas funcionalidade para dentro do pacote porem ela funciona de forma um tanto complexa para variar embora a logica dela pode ser um pouco parecida com a herança múltipla do c++, para a gente herdar basta a gente instanciar o objeto dentro do construtor e depois passar o objeto para estrutura que passamos o atributo

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kami();

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}

sub fts
```

```
{
     print("esse e o pacote kodo");
}

package kami;

sub new
{
    my $this = shift;
    my $kobjeto = new kodo();
    my $ini = { kodo => $kobjeto};
    bless($ini,$this);
}
```

para acessar algum atributo ou método herdado a gente teria que acessar a hash daquele objeto que foi passado na estrutura e depois acessar o metodo ou o atributo (se for uma atributo seria mais uma hash)

```
#/usr/bin/perl

use strict;
use warnings;

my $objeto = new kami();
$objeto->{kodo}->fts();

package kodo;

sub new
{
    my $this = shift;
    my $ini = {};
    bless($ini,$this);
}

sub fts
{
     print("esse e o pacote kodo");
}

package kami;

sub new
{
    my $this = shift;
    my $kobjeto = new kodo();
    my $ini = { kodo => $kobjeto};
    bless($ini,$this);
}
```

7.0 - EOF

e ae galera chegamos em mais um final de ebook, espero futuramente escrever outros e atualizar esse e os anteriores com novos conteúdos além de corrigir alguns erros, tambem não posso deixar de agradece os adms do eof que tem me atormentado digo digo tem me ajudado a alguns anos como o viciado em programação do mmxm que também programa nessa linguagem e não quer aprender pascal u.u , o massacre do portão elétrico também chamado de sir.rafiki ou f.ramon ou code universal não chegue perto ou sera esmagado por um portão autoprogramado para isso '-' , grande mano s1m0n e seus projetos dominação global com drones pense no magneto é esse cara \o , alem de outros manos como gjunnioor e sua devoção ao editor vim (use o NANO caramba), mano nass que sempre me confundo de quantos s tem o nick dele kkk, neofito e seu vicio em animes hentai, lend4pop que nunca vi cantando uma musica, e oracle e seu passado obscuro de destruição em massa '-' , resumindo esse grupo só tem doido mesmo kkkk

http://eofcommunity.com/forum

http://www.facebook.com/hacker.fts315

https://gist.github.com/hackerfts315

http://fts315.xp3.biz

http://ftslinksdeepweb.esy.es

também não posso deixar de recomenda alguns sites e fóruns que tenho entrado e alguns parceiros dele também

http://hc0der.blogspot.com.br/

http://injectionsec.com/

http://brutalsecurity.com.br