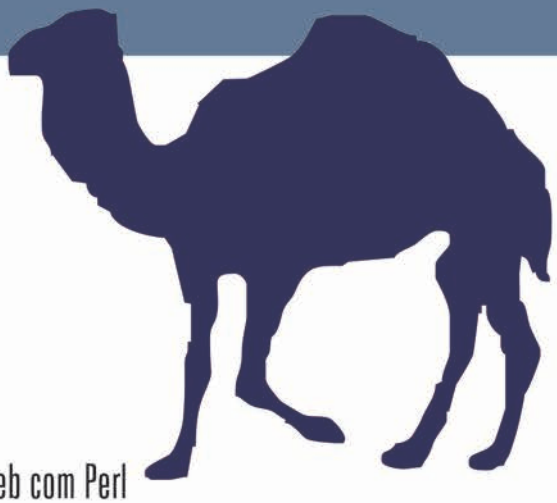
Inclui arquivos para a construção dos projetos apresentados no livro

Thiago Glauco Sanchez

Programando com Perl



- Princípios do Desenvolvimento Web com Perl
- Criação de programas em ambientes Linux, Windows, FreeBSD, Solaris e Android
- Trabalhos com hashes, arrays e variáveis escalares
- Sistemas para administrar roteadores, switches e servidores

Atualizado até a versão 5.14

Programando com

Perl

Thiago Glauco Sanchez

Programando com Perl

Copyright© 2012 por Brasport Livros e Multimídia Ltda.

Todos os direitos reservados. Nenhuma parte deste livro poderá ser reproduzida, sob qualquer meio, especialmente em fotocópia (xerox), sem a permissão, por escrito, da Editora.

Editor: Sergio Martins de Oliveira
Diretora: Rosa Maria Oliveira de Queiroz
Gerente de Produção Editorial: Marina dos Anjos Martins de Oliveira
Revisão de Texto: Maria Helena A. M. Oliveira
Editoração Eletrônica: Ingrafoto Produções Gráficas (Michelle Paula)
Capa: Paulo Vermelho

Técnica e muita atenção foram empregadas na produção deste livro. Porém, erros de digitação e/ou impressão podem ocorrer. Qualquer dúvida, inclusive de conceito, solicitamos enviar mensagem para editorial@brasport.com.br, para que nossa equipe, juntamente com o autor, possa esclarecer. A Brasport e o(s) autor(es) não assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso deste livro.

BRASPORT Livros e Multimídia Ltda.
Rua Pardal Mallet, 23 – Tijuca
20270-280 Rio de Janeiro-RJ
Tels. Fax: (21) 2568.1415/2568.1507
e-mails: marketing@brasport.com.br
vendas@brasport.com.br
editorial@brasport.com.br

site: www.brasport.com.br

Filial SP
Av. Paulista, 807 – conj. 915
01311-100 São Paulo-SP
Tel. Fax (11): 3287.1752
e-mail: filialsp@brasport.com.br

À minha esposa,
à memória de meu pai, Dr. José Antônio Sanchez,
à memória de meu avô Antônio José Sanchez Rodriguez,
à memória de meu padrinho de casamento, José Maximino Guimarães,
às comunidades de software livre: Perl Mongers e Viva o Linux.

Agradecimentos

Muitas pessoas têm méritos neste livro. Meus alunos do curso de Perl Online, minha esposa e as comunidades São Paulo Perl Mongers, Rio Perl Mongers e Viva o Linux, que fazem um ótimo trabalho divulgando o Perl e as boas práticas de programação.

Quero fazer um agradecimento aos professores do curso de Eletrônica Plena, da Escola Técnica Estadual Getúlio Vargas da turma de 1995 a 1998 - uma das últimas turmas do ensino técnico integrado ao ensino médio –, que me agraciaram com meu primeiro ensino profissionalizante. Nessa escola estudei por quatro anos os princípios de eletrônica, telecomunicações e programação que foram fundamentais para a minha vida profissional. Agradeço de coração aos professores:

Shibata - que me ensinou os princípios da eletrônica digital e da lógica booleana;

Marta - professora de eletrônica;

Celestino - professor de telecomunicações;

Fonseca - Professor de programação de computadores;

Também agradeço aos professores do curso de Tecnologia em Automação Industrial do CEFET-SP, agora IFET, onde tive a honra de aprender a programar em C e Assembler.

E por fim agradeço a você, leitor, sem o qual nenhum livro tem sentido.

Em especial:

Este livro não existiria se não fosse a dedicação especial e o esforço de uma pessoa que reuniu diversos artigos e apostilas de minha autoria e buscou uma editora que tivesse interesse em publicar um livro sobre programação em Perl.

Sempre me apoiando, revisou diversos textos, me ensinou como organizar as metas de um livro e de um curso de forma a alcançar o objetivo e me ajudou a escrever de forma didática e coesa. Metade dos méritos desta obra pertence a minha esposa, a pedagoga Luciana Sanchez.

Agradeço também à Sônia Maria Motão e à Roseli Motão, que cuidaram de minha esposa enquanto ela esteve doente para que eu pudesse terminar a revisão do livro.

Apresentação

Sobre este livro

Este livro é um dos poucos que trata exclusivamente a linguagem de programação Perl no Brasil. Seu formato é bem didático, como que seguindo um curso. Se você for um programador iniciante em Perl, recomendo que siga o livro como toda jornada de aprendizado deve ser: inicie no começo e siga até final, então estará feito. Se você for um programador experiente, pode seguir a ordem que desejar.

Abordei o ensino da linguagem buscando suprir as dificuldades que encontrei ao aprender Perl: muita documentação técnica e poucos materiais didáticos em português.

Este livro faz uma abordagem didática ao invés de uma abordagem completa. Uma abordagem completa confundiria programadores iniciantes e tornaria o texto demasiadamente longo - as abordagens completas ficam a cargo das documentações técnicas que o leitor deve buscar para complementar as áreas que deseja se aprofundar. Caso o leitor deseje mais informações sobre um assunto pode buscar nos sites, livros e tutoriais indicados no decorrer dos textos conforme as necessidades da sua realidade prática. Afinal, para algumas pessoas, automatizar a administração de servidores pode ser mais importante que a programação web.

O livro está repleto de exemplos e exercícios. Se você executar e estudar os exemplos como um laboratório, desenvolver os exercícios e estudar as documentações indicadas, vai se tornar hábil em Perl em bem pouco tempo, além de se familiarizar com as comunidades e com a cultura Perl. O leitor iniciante em programação também deverá buscar auxílio em livros e artigos que tratem de algoritmos e estruturas de dados.

Um Erro de Programação Pode Matar??

Erros de Software que foram fatais

A inteligência artificial e os sistemas automatizados vêm se tornando parte constante da vida diária das pessoas. Câmeras digitais, fornos de micro-ondas, automação residencial. Nestas condições pequenos bugs de software podem causar prejuízos pequenos ou danos leves. Mas quando um software defeituoso é responsável por um sistema de missão crítica ou mesmo em um equipamento hospitalar, responsável pela saúde de pessoas? E se um programador errar em um sistema que será utilizado para especificação de materiais para construção de prédios residenciais?

Diferentemente dos médicos, nós programadores não sentimos o pulso das pessoas nem sua respiração perto de nós. Enquanto trabalho eu apenas ouço o som das teclas, o leve som do cooler e a minha rádio de internet preferida e codifico, testo, codifico... apesar de não termos essa proximidade com a vida de outras pessoas, muitas vezes sistemas mal projetados podem custar milhões de dólares, ou até vidas humanas. Não estou sendo terrorista. Apenas quero trazer para você a responsabilidade do que é ser algo mais que um programador por hobby.

Uma das características que tornou o Perl uma linguagem famosa e de vida longa é a qualidade suficiente para ser usada em sistemas de missão crítica ou non-stop, além de sistemas da NASA e da NSA – Agência Nacional de Segurança dos EUA. O Perl é utilizado com sucesso por grandes corporações cuja hora parada pode significar perdas de milhões de dólares. Projetos relacionados ao genoma e mesmo em perícia. E se você quer aprender Perl é porque deseja ir além de apenas programar aplicativos desktop.

Portanto, você deve se preocupar com a segurança e a qualidade do seu software em certas condições onde errar não é permitido – pelo menos não quando o sistema estiver em produção.

Veja esses casos:

- Estádio desaba em Hartford em 1978. Dano de US$ 90.000.000,00 da época. Causa: Programador do sistema de CAD utilizou variáveis erradas no sistema que calculou esforços de barras de aço.
- Sistema soviético de vigilância contra ataque nuclear alarmou acidentalmente em 1983, indicando um falso ataque dos Estados Unidos. Isso quase causou uma Terceira Guerra Mundial. Erro de software fez o sistema confundir a reflexão solar com mísseis.
- Therac-25 matou cinco pessoas em 1985. Máquina de radioterapia gerou overdose de radiação devido a erro na programação.
- Sistema de radioterapia mata oito pessoas e queima vinte em 2000. Erro de cálculo no software da Multidata System International, que era utilizado para controlar a emissão de radiação em tratamentos de câncer, matou oito pessoas e causou queimaduras graves em outras vinte.

No site 20 Desastres Famosos de Software – http://www.devtopics.com/20-famous-software-disasters/ – você pode ver esses e outros desastres provocados por erros de software, ou melhor, erros dos programadores ao criarem os softwares.

Sumário

Introdução **1**
Por que aprender Perl? 1
Instalando o Perl no Linux 3
Instalando o Perl via gerenciador de pacotes do Linux 4
Instalando o Perl no Windows 5
Preparando o ambiente 5
O Perl Portable 6
Shebang! 6
Exercícios 7
Capítulo 1. Visão Geral do Perl e dos Conceitos de Programação **9**
1.1 Linguagem e programação 9
1.2 O processo de execução de um programa em Perl 11
1.3 Conceitos básicos 12
1.3.1 Perl, variáveis, constantes e dados 12
1.3.2 Tipos de dados no Perl 15
1.3.3 Comunicando-se com o mundo externo 17
1.3.4 Operadores 20
1.3.5 Tomada de decisão 24
1.3.6 Loops 27
1.4 Debug básico 29
1.4.1 Missing semicolon (você esqueceu o ponto-e-vírgula) 30
1.4.2 Runaway strings 30
1.4.3 Outros erros 31

1.4.4 Comando perl –c 32
Exercícios 33
Capítulo 2. Funções, Módulo, Referências, Objetos e Debug 34
2.1 Funções 34
2.1.1 Funções internas do Perl 34
2.1.2 Funções definidas pelo usuário 36
2.2 Variáveis predefinidas 37
2.3 Módulos 37
2.4 Instalando módulos CPAN 38
2.5 Referências 40
2.6 Objetos 41
2.7 Filehandles 43
Exercícios 45
Capítulo 3. Variáveis Simples: Strings e Expressões Regulares Simples 46
3.1 Regras para nomes 46
3.2 Palavras reservadas 47
3.3 Strings 48
3.4 Strings literais 48
3.5 Concatenando strings 49
3.5.1 As muitas cotas do Perl 50
3.6 Digitando menos com a função say 51
3.7 Textos grandes com os documentos HERE 52
3.8 O operador x 56
3.9 As funções printf e sprintf 57
3.10 Controlando a caixa de uma string 67
3.11 Transformando uma string em um array de caracteres 69
3.12 Expressões regulares 71
Exercícios 81
Capítulo 4. Variáveis Simples: Números e Operações Matemáticas 82
4.1 Formas de representar um número no Perl 82

4.2 Literais inteiros 83
4.3 Literais reais ou de ponto flutuante 83
4.4 Quando é string e quando é número? 83
4.5 Atividades comuns com os números 84
4.5.1 Revendo os operadores 85
4.5.2 Comparando números reais com ponto flutuante 85
4.5.3 Verificando se um escalar é um número 86
4.5.4 Gerando números randômicos 87
4.6 Arredondando números 87
4.7 Números complexos com Math::Complex 91
Exercícios 92
Capítulo 5. Operadores, Decisão e Controle de Fluxo 93
5.1 O Operador de atribuição '=' 94
5.2 Operadores aritméticos 95
5.3 Operadores de deslocamento binários (shift) 96
5.4 Operadores de strings 97
5.5 Operadores lógicos 97
5.6 Operadores bitwise 102
5.7 Operadores de incremento e decremento 104
5.8 Combinando os operadores 106
5.9 Operadores de comparação 107
5.10 Operador ternário 109
5.11 Loops e estruturas de decisão 110
5.11.1 Decisão 110
5.11.2 O bloco unless 112
5.11.3 O módulo switch 114
5.11.4 Given when 114
5.12 Loops 116
5.12.1 Loops para hashes e arrays 117
5.12.2 A função keys 119
5.12.3 A função map 120

5.12.4 A função values 121
5.12.5 A função each 121
Exercícios 122
Capítulo 6. Funções Internas do Perl e Funções do Programador 123
6.1 Declarando funções 123
6.2 Argumentos 125
6.3 Valor do retorno 127
6.4 A função main 130
6.5 O escopo das variáveis 130
6.6 Recursividade 136
6.7 Funções internas do Perl 138
6.7.1 Funções Perl por categoria 138
6.7.2 Como trabalhar com as funções do Perl 141
6.8 Técnicas para trabalhar com argumentos em funções 141
6.9 Passando referências como argumentos 143
6.10 Protótipos 148
6.11 Retornando valores com wantarray 148
Exercícios 151
Capítulo 7. Variáveis de Lista: Arrays e Hashes 152
7.1 Introdução e escolha de dados 152
7.2 A função qw 152
7.3 Manuseando um array 153
7.3.1 Ordenando o array 157
7.3.2 Transformando strings em arrays e arrays em strings 159
7.4 Manipulando os hashes 160
7.4.1 Funções para manusear hashes 160
7.5 Arrays de arrays... 164
7.6 Arrays de hashes 166
7.7 Hashes de arrays 167
7.8 Hashes de funções 168
Exercícios 170

Capítulo 8. Expressões Regulares **172**
8.1 As expressões regulares 173
8.2 Os metacaracteres 174
8.2.1 Atalhos para classes de caracteres 182
8.3 Os operadores m/// e s/// 182
8.4 Verificando a posição na string 187
8.5 O metacaractere âncora \b 189
8.6 O que ler agora? 189
8.7 Receita especial – match aproximado 190
Exercícios 190
Capítulo 9. Manuseando Arquivos e Outros Filehandles **191**
9.1 Por que manusear arquivos? 191
9.2 Criando um filehandle 192
9.2.1 Modos de acesso 192
9.2.2 Lendo os dados de um arquivo 193
9.2.3 Abrindo arquivos para adicionar apêndices 194
9.2.4 Abrindo arquivo para leitura e escrita 196
9.3 A função select 198
9.4 Arquivos binários 199
9.5 File locking 201
9.6 Testes de arquivos e diretórios 203
9.7 A variável @ARGV 207
9.8 O módulo Getopt::Std 210
9.8.1 O novo echo 212
9.8.2 sping.pl 215
9.9 IPC 219
9.9.1 A função system 220
9.9.2 A função exec 225
9.9.3 Obtendo a saída de um comando 226
9.10 Pipes 228
Exercícios 230

Capítulo 10. Técnicas de Debug – The Perl Debugger **231**
10.1 Antes de caçar insetos... 231
10.1.1 Definir o escopo de atuação do programa 231
10.1.2 Definir as funcionalidades do programa 232
10.1.3 Depois de tudo isso... codificar 233
10.1.4 Executar o programa 233
10.1.5 Teste e debug 233
10.1.6 Entregar e ajustar o programa 234
10.1.7 Modificações e manutenções 234
10.2 O debugger do Perl 234
Exercícios 246
Capítulo 11. Utilizando Módulos do CPAN e Criando Seus Próprios Módulos **247**
11.1 Pacotes e namespaces no Perl 249
11.2 O hash %INC 254
11.3 O array @INC 255
11.4 Criando seus próprios módulos 256
11.5 A função AUTOLOAD 260
Exercícios 265
Capítulo 12. Referências **266**
12.1 Referências a funções 270
12.2 Verificando estruturas com o Data::Dumper 272
12.3 Armazenando estruturas complexas 273
12.4 Um exemplo com Tk 278
Exercícios 279
Capítulo 13. Programação Orientada a Objetos com Perl **280**
13.1 Por que usar programação orientada a objetos? 280
13.2 Definições de um objeto 282
13.3 Existe alguma desvantagem em usar OOP? 283
13.4 Criando um objeto 284
13.5 Herança 291

13.6 Atributos, métodos de classe e destrutores 294
13.7 A classe SUPER 298
13.8 Em resumo 299
Exercícios 299
Capítulo14. Introdução ao Trabalho com Imagens no Perl 300
14.1 Desenhando com o Perl 301
14.2 Criando formas básicas 306
14.3 Adicionando texto às imagens 308
14.4 Estendendo o módulo GD::Simple 309
14.5 O que ler agora? 310
Exercícios 310
Capítulo 15. Bancos de Dados Relacionais 311
15.1 Esquema 315
15.2 Tipos de dados 315
15.3 Normalização e design do banco 316
15.3.1 Quais são as informações relevantes? 317
15.3.2 Script SQL 320
15.4 SQL e o Perl 327
15.5 Lidando com os erros de forma segura 330
15.6 Os principais comandos do SQL em programas Perl 335
15.7 Inserindo dados nas tabelas 341
15.8 As diferenças entre os métodos do() e prepare() 343
15.9 O que ler agora? 343
Exercícios 343
Capítulo 16. Suas Aplicações na Web com CGI.pm 344
16.1 Criando formulários Perl e CGI.pm 351
16.2 Desenvolvimento web com FCGI 357
16.3 O que ler agora? 358
Exercícios 358
Capítulo 17. Trabalhando com Threads 359
17.1 Mas antes vem a pergunta: o que é uma thread??? 359

17.1.1 De forma mais técnica, uma thread... 363
17.2 Como trabalhar com threads 364
17.2.1 A segurança 365
17.2.2 Criando uma thread 365
17.3 Compartilhando dados entre threads 371
17.4 A queue 373
17.5 Os semáforos 375
17.6 O que ler agora? 378
Exercícios 378
Capítulo 18. Programação Orientada a Objetos com Moose 379
18.1 Relacionamento de herança e relacionamento "tem um" 385
18.2 O que ler agora? 390
Exercícios 390
Capítulo 19. Gerenciando Elementos de Rede com o Perl 391
19.1 Entendendo o SNMP 392
19.1.1 O que você deve saber sobre MIBs 393
19.1.2 E como eu faço para minha rede responder ao protocolo SNMP? 396
19.2 Criando uma solução SNMP personalizada com o Perl 400
19.3 Telnet e SSH 411
19.4 Criando um servidor de syslog 415
19.5 O que ler agora? 417
Exercícios 417
Capítulo 20. Dicas para o Mundo Windows 418
20.1 O módulo Win32 419
20.2 Win32::EventLog 421
20.3 Manuseando o registro do Windows 424
20.4 Lendo os fabricantes de software instalados 425
20.5 DBD::WMI 428
20.6 O que ler agora? 429
Exercícios 429
Capítulo 21. Monitorando Sistemas Linux 430

21.1 Analisando um arquivo de log comum do Linux 430
21.2 Dados da estatística do sistema 436
21.3 O que ler agora? 439
Exercícios 439
Capítulo 22. Android 440
22.1 A camada de scripts do Android – SL4A 443
Capítulo 23. A Evolução do Perl 5.10 ao 5.14 447
23.1 O operador smart match 447
23.2 Substituição não destrutiva 450
Despedida 451

Introdução

Por que aprender Perl?

Existem, hoje, dezenas de linguagens de programação com as mais diversas finalidades e especialidades. Porém, podemos contar nos dedos linguagens open source que chegaram ao grau de amadurecimento e versatilidade a que o Perl chegou. Desenvolvida por Larry Wall para facilitar os trabalhos do Laboratório de Jatos Propulsores da NASA, recebeu ao longo dos anos contribuições valiosas de diversos desenvolvedores e passou do estado de uma simples linguagem de Extração e Relatórios para uma das mais poderosas – se não a mais – linguagens open source de propósito genérico: desde a simples tarefa de encontrar expressões em textos longos até a criação de e-commerce ou mesmo a criação dinâmica de imagens. Também utilizada hoje em bioinformática para processar as complexas informações das cadeias de DNA, Perl é a linguagem escolhida por milhares de desenvolvedores, administradores de sistemas e webmasters para facilitar seus trabalhos.

Os motivos que levam a esta popularidade do Perl são diversos. Não digo popularidade no sentido do marketing, pois você não verá grandes propagandas ou portais sobre Perl como vê com os produtos da Microsoft. Mas sim uma popularidade prática. Uma popularidade entre os profissionais que seguem a filosofia moderna das empresas: "faça mais com menos". E o menos, neste caso, são dinheiro e tempo. A lista de características que conferem fama ao Perl e motivam o seu aprendizado são:

- **Custo:** Perl é uma linguagem de código aberto e gratuita. Ao utilizar uma linguagem de código aberto as empresas ganham com custo. Ao ensinar linguagens de código aberto as escolas ganham com custos. Ao disseminar o uso profissional de produtos de código aberto contribuímos para a sociedade como um todo. Aumentamos a inclusão digital. Reduzimos a pirataria. Aumentamos o PIB e diminuímos a emissão de dólares ao exterior.

- **É uma linguagem simples** de aprender e com apenas um pouco de conhecimento é possível criar programas realmente úteis e práticos.
- **Portabilidade:** programas em Perl podem ser executados em Linux, Windows, Mac e algumas versões de celulares que hoje possuem suporte ao Perl.
- **Variedade de aplicações:** como linguagem de alto nível, já madura e recebendo ao longo dos anos contribuições de diversos desenvolvedores ao redor do globo, o Perl obteve uma incrível versatilidade. Hoje sites são escritos quase que inteiramente com Perl/CGI. Aplicativos para administradores de sistemas contam com as capacidades do Perl para manusear os arquivos de configuração do Linux e o Registro do Windows e de acessar, modificar e preencher tabelas em bancos de dados MySQL, Oracle ou mesmo MS-Access. O limite é a sua necessidade e sua criatividade.

Essas características tornam este livro muito importante para aqueles que desejam aprender a linguagem Perl ou simplesmente para quem quer iniciar seu aprendizado em programação e optou por fazê-lo com a filosofia do código aberto.

Sendo um livro de Perl básico, irei cobrir o núcleo da linguagem e seus módulos para as aplicações mais comuns – web, banco de dados, orientação a objetos, controle de processos e sockets.

Os objetivos deste livro são:

1. Auxiliá-lo no desenvolvimento do raciocínio lógico necessário para programação;
2. Apresentar as principais estruturas de programação dentro do Perl;
3. Trazer os conceitos de programação orientada a objetos aos aplicativos reais;
4. Criar aplicativos web voltados a banco de dados com Perl;
5. Introduzir o controle de processos e acesso à rede via sockets;
6. Utilizar o Perl para desenvolver estruturas tipo Listas e Filas.

Após a leitura deste livro, você será capaz de:

1. Instalar o Perl nos sistemas Linux e Windows;
2. Desenvolver scripts Perl para diversas atividades profissionais;
3. Reconhecer e corrigir erros em programas Perl;
4. Avaliar a funcionalidade e o desempenho de aplicativos feitos em Perl.

Instalando o Perl no Linux

A bem da verdade, o Perl já é incluído na maioria das instalações Linux padrão. Basta verificar com o comando ***perl -v***:

$ perl -v
This is perl 5, version 12, subversion 2 (v5.12.2) built for i686-linux Copyright 1987-2010, Larry Wall
Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5 source kit.
Complete documentation for Perl, including FAQ lists, should be found on this system using "man perl" or "perldoc perl". If you have access to the Internet, point your browser at http://www.perl.org/, the Perl Home Page.

Enquanto eu elaborava o livro, a última versão estável do Perl era a 5.12, mas qualquer versão estável a partir da 5.10 vai atender às necessidades deste livro.

Porém, pode acontecer de você necessitar atualizar a versão do Perl no seu sistema. Então, a primeira coisa a se fazer é acessar o site http://www.perl.org/ e baixar a última versão estável do Perl. Salve este site nos seus favoritos, pois ele tem muita informação atualizada sobre o mundo e a cultura Perl. Se você não tem familiaridade com o inglês, é bom pensar em um curso, caso pretenda seguir uma carreira séria e de respeito na área de TIC.

Como este livro não é para experts em Linux ou Windows, você pode precisar de uma "mãozinha" para instalar. Algumas versões do Linux já vão abrir um descompactador se você clicar duas vezes no arquivo. Mas, às vezes, você precisa instalar remotamente, por exemplo, via telnet ou em um ambiente que tenha apenas interface de linha de comando. Neste caso, para descompactar, use os comandos a seguir:

$gunzip perl-5.12.3.tar.gz
$ tar -xvf perl-5.12.3.tar

NOTA: É muito importante que você leia os arquivos **README, INSTALL e README.<Seu_Sistema>** para verificar detalhes sobre o sistema onde você fará a instalação. O arquivo README.solaris, por exemplo, alerta:
"If you remove the perl supplied with the OS, you will render some bits of your system inoperable."
Portanto, faça uma instalação local se estiver usando Solaris 8 ou Solaris 10.

```
$ cd perl-5.12.3
$ ./Configure -des -Dprefix=$HOME/localperl
$ make
$ make test
$ make install
```

Depois você tem que adicionar ~/localperl ao seu path para tornar a nova instalação acessível. Detalhe: mude o path apenas em ambientes de desenvolvimento. Evite modificar o path em ambientes de produção. Para mudar o path no Linux em uma shell bash basta adicionar:

```
$ export PATH=$HOME/localperl:$PATH
```

Você também pode editar o seu arquivo ~/.profile do seu Linux. Para especificar qualquer outro diretório para instalação utilize o comando:

```
$ ./Configure -Dprefix='/diretório_qualquer/dir'.
```

Para obter ajuda sobre os parâmetros da instalação use:

```
$ ./Configure –help
```

O sucesso da instalação pode ser verificado com o comando **perl -v.**

Instalando o Perl via gerenciador de pacotes do Linux

Você também pode instalar ou atualizar o Perl via gerenciador de pacotes do Linux. É mais simples para iniciantes. Nas distribuições baseadas em Debian basta o comando:

```
$ sudo aptitude install Perl
```

Para iniciar o processo de instalação do Perl nas distribuições baseadas em Red Hat, você pode verificar com o **yum**:

```
# yum install perl.i686
```

NOTA: O nome do pacote pode mudar entre as distribuições. Faça uma pesquisa com **aptitude search perl** ou **yum search perl** para saber o nome correto. Os gerenciadores de pacote Linux também possuem versões gráficas.

Instalando o Perl no Windows

No site www.perl.org, na área de Downloads, existe a opção Download StrawberryPerl. Baixe e instale como qualquer programa Windows: dê dois cliques no pacote .msi; aceite a licença; defina a pasta de instalação (a padrão é c:\strawberry) e "next... next... finish". Teste a instalação com o comando perl -v no prompt de comandos do Windows.

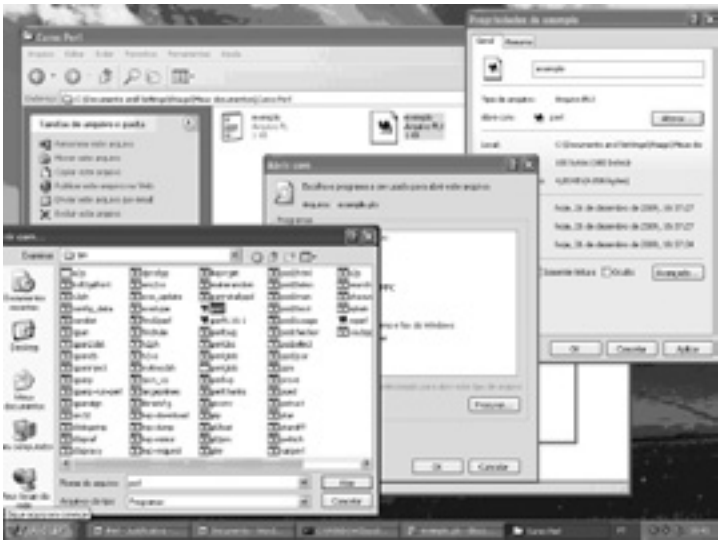
Preparando o ambiente

No Windows é preciso definir que os arquivos com a extensão .pl ou .plx serão executados pelo Perl. Para isto, abra o bloco de notas ou qualquer outro editor de textos e digite:

```
print "O Perl funciona normalmente no Windows com o Strawberry!\n";
```

Salve como windows.plx. Clique com o botão direito no ícone do arquivo salvo. Vá em Propriedades > Geral > Alterar > Procurar. Irá aparecer a janela "Abrir com". Navegue até a pasta do strawberry\perl\bin e escolha perl. Clique em abrir e pronto. Você vai perceber que o ícone do camelo vai ser adicionado em todos os arquivos .plx. Todos os arquivos com a extensão .plx serão automaticamente executados pelo Perl, sem a necessidade de chamar o compilador.

NOTA: O nome do pacote pode mudar entre as distribuições. Faça uma pesquisa com aptitude search perl ou yum search perl para saber o nome correto. Os gerenciadores de pacote Linux também possuem versões gráficas.



Abra o prompt do Windows e execute seu script:

c:> windows.plx

saída:
O Perl funciona normalmente no Windows com o Strawberry!

O Perl Portable

No mundo Windows você ainda tem a opção de utilizar o Perl Portable, uma distribuição que fica armazenada em uma USB ou em uma pasta qualquer. Está disponível no site http://strawberryperl.com/releases.html, já na versão 5.12. Além do Perl ela provê uma interface de linha de comando que permite o uso do Perl sem necessitar modificar o path da sua máquina. Além do compilador dmake, que é útil para instalação de módulos adicionais.

Como o Perl pode ser executado em cerca de duzentas plataformas, fica impossível mostrar todos os detalhes da instalação em cada plataforma. Fica para o leitor verificar os arquivos **README** e **INSTALL** de cada distribuição.

Shebang!

O ***shebang*** é formado pelos caracteres #! e deve ser colocado na primeira linha do script seguido do path onde o Perl está instalado se você estiver em um ambiente Linux. Mas o que vem a ser o path? Path, ou caminho, é a localização de determinado arquivo dentro da estrutura de diretórios do seu HD. Por exemplo, no Windows, o path provável da sua instalação Perl será **c:\Strawberry\bin\perl.** No Linux será **/usr/bin/perl ou /usr/local/bin/perl**. Mas isto não é universal. Você precisa saber onde está o Perl no sistema no qual a aplicação irá rodar. Para saber como fazê-lo no Linux, basta digitar o comando **which perl**:

$ which perl
/usr/local/bin/perl

Se houver mais de uma instalação, você pode pesquisar com o comando whereis perl:

$ whereis perl
perl: /usr/bin/perl /etc/perl /usr/lib/perl /usr/local/bin/perl /usr/local/lib/perl /usr/share/perl /usr/share/man/man1/perl.1.gz

Normalmente, na arquitetura Linux, os arquivos binários são inseridos em alguma pasta bin. No caso temos no meu sistema ***/usr/bin*** e ***/usr/local/bin***. O Perl está nas duas pastas no meu sistema, sendo o 5.12 na /usr/local/bin e o 5.10 na pasta /usr/bin, que é o original do sistema. Caso eu encontre algum bug na versão 5.12 poderei rodar meus programas na versão 5.10 apenas mudando o shebang.

NOTA: No ambiente Windows, o shebang é indiferente, a não ser que o script seja executado por um servidor Apache, que necessita do shebang mesmo no Windows.

Agora abra um editor de textos, digite o código a seguir e salve como **primeiro.pl:**

```
1 #!/usr/bin/perl
2 print "Meu primeiro script Perl.\n";
```

Na linha de comando, se estiver no Linux atribua a propriedade de executável com o comando chmod:

$ chmod +x primeiro.pl

Agora, quando for executar o script, basta digitar $ ./primeiro.pl:

$ ./primeiro.pl

saída:
Meu primeiro script Perl.

NOTA: Para programadores C/C++, as funções ***print*** e ***say*** do Perl 5.12 são equivalentes à função ***printf*** do C. O Perl 5.12 também possui uma função ***printf*** que funciona exatamente igual à sua similar no C. Para programadores Shell Script, a função **print** é semelhante à função **echo** do **bash**.

Exercícios

Os exercícios têm como objetivo, além de reforçar o conteúdo do livro, ambientar o leitor com a cultura Perl e com o uso do Perl em grandes projetos.

1. Faça uma instalação local do Perl no seu sistema.
2. Organize seu sistema conforme descrito neste capítulo.
3. Crie um pequeno script hello_world.pl.

4. O MySQL é o sistema de banco de dados relacional mais utilizado no mundo. Acesse a página http://dev.mysql.com/ e verifique quanta informação útil sobre Perl pode ser encontrada lá. Visite também http://dev.mysql.com/usingmysql/perl/ e http://planet.mysql.com/.

5. O Apache é o servidor web mais utilizado no mundo até hoje. Acesse o site www.apache.org e busque informações sobre Perl, mais especificamente sobre mod_perl, emb_perl.

6. Após verificar a importância do Perl em grandes projetos como o Apache e o MySQL, o quão importante é o Perl na sua opinião?

Capítulo 1. Visão Geral do Perl e dos Conceitos de Programação

1.1 Linguagem e programação

Agora que estamos um pouco mais familiarizados com o Perl, sua história e utilidades, vamos começar a pensar como programadores. O Perl é uma linguagem de programação e, como qualquer linguagem, possui sintaxe e características próprias. Mas antes de entrar no detalhe vamos falar da generalidade.

O que é uma linguagem? Linguagem é uma forma de expressar ideias, seja de forma escrita, falada, desenhada ou gesticulada. No nosso caso é uma linguagem escrita. Isso significa que ela possui símbolos e sinais únicos como ***$, @, if, elsif*** entre outros. Os símbolos têm um significado que deve ser conhecido para que se compreenda a linguagem.

Sendo uma forma de expressar ideias, é claro que necessitamos de alguém que leia e entenda nossas ideias. No caso da programação, queremos que o computador (mais especificamente o compilador e/ou o interpretador perl*) entenda e execute uma tarefa. Linguagens de programação possuem estruturas imperativas e interrogativas. Ou seja, estamos sempre perguntando coisas e, baseados na resposta, ordenando coisas.

Exemplo:

```
Está frio?
  Sim.
      Então leve uma blusa.
  Não.
     Então vá de bermuda.
```

NOTA: Quando nos referimos a Perl com letra maiúscula estamos falando da linguagem. Quando nos referimos com letras minúsculas estamos falando do compilador e do interpretador, ou seja, o comando perl.

Ao programar temos que modificar e adaptar nosso pensamento a essa linguagem. Como quando aprendemos uma língua estrangeira e temos que nos adaptar a suas características culturais e etimológicas, o mesmo ocorre com a programação e com o Perl.

O programador quer passar um conjunto de instruções ao computador, como "agora mostre isso ao usuário." ou "Se demorar muito, vá para a próxima etapa". Mas explicar algo a um computador é diferente de explicar a um ser humano. E essa é a grande adaptação do pensamento.

Humanos replicam, respondem, perguntam de volta quando eles não entendem algo. Imagine você ensinando uma criança a atar o cordão do sapato. "Pegue as duas pontas do cordão." A criança pega e retruca: "Assim?" e você pode responder, "Sim, isso mesmo" ou "Não, não. Olha como eu faço". Você não tem essa interatividade com o computador. Ainda não existem linguagens de programação que possam transformar gestos em código. O Perl irá supor que você detalhou o suficiente e corretamente. Assim, ele irá seguir com exatidão as instruções. Você só terá um "feedback" se solicitar um.

Para que um programa dê certo você deve ser capaz de explicar a tarefa detalhadamente a você mesmo. Eu quero que você saia do piloto automático e pense num processo bem comum para você mesmo, como excluir ou mover arquivos. Como você define quais arquivos devem ser apagados? Arquivos que não são usados há muito tempo, por exemplo. Mas "muito tempo" é um termo vago para um computador. Quanto tempo é muito para você? Uma semana? Um mês? Quem sabe cinco anos?

Talvez você prefira apagar arquivos cujos assuntos não lhe interessem. Como você identifica estes arquivos e assuntos? Você é capaz de explicar isso detalhadamente? Você abre o arquivo, examina o conteúdo e, se ele tiver algum texto do seu interesse, você o mantém, se não tiver o apaga. Como explicar qual conteúdo é do seu interesse ou não?

Vamos entrar no detalhe deste contexto. Eu quero que sejam removidos da pasta da rede todos os arquivos que não são relacionados a trabalho. Como fazer isto? Um ser humano olhando os arquivos deduziria isto rapidamente. Apagaria, por exemplo, vídeos, músicas. Como explicar isso ao computador?

Talvez ordenando: "apague todos os arquivos da pasta cuja extensão não for .doc ou .xls ou .ppt"?

Mas seus usuários começariam a renomear os arquivos de música. Seria melhor pesquisar o conteúdo do arquivo. Pesquisar o cabeçalho de formato do arquivo, por exemplo.

Aprender a programar é aprender a expressar suas ideias de forma que um computador seja capaz de interpretar e executar. Desenvolver a capacidade de prever situações inesperadas e entender o que seu cliente precisa.

1.2 O processo de execução de um programa em Perl

O processamento de código Perl é um pouco diferente das outras linguagens. Quando você evoca o comando perl na linha de comando ou via shebang, uma atividade complexa, porém muito rápida, inicia. Primeiro o seu código-fonte é compilado, as diretivas e a sintaxe são analisadas. Seu código-fonte é transformado em um conjunto de códigos referido em algumas literaturas como "Perl opcode", um código que pode ser entendido pelo interpretador Perl.

NOTA: Embora Perl seja uma linguagem muito antiga, ela sempre evolui. Esta forma de tratar o código é muito semelhante à maneira como o Java e o C# lidam com seu código.

Então os opcodes são enviados para um mecanismo de execução conhecido como interpretador Perl e o programa é enfim executado. Erros de sintaxe são identificados no início do processo de compilação, logo não há riscos de iniciar execução de um processo e ser interrompido no meio por uma linha com erro de código. Loops, constantes e similares são compilados apenas uma vez.

Mas tenha em mente que a compilação consome um pouco de tempo. Então não é inteligente, embora seja possível, criar programas em Perl que evoquem diversos outros programas em Perl para dividir tarefas complexas em tarefas simples, pois estes serão compilados todas as vezes que forem evocados pelo primeiro programa. Uma técnica melhor de se fazer isso seria valer-se de módulos, classes e funções. Eles têm a mesma facilidade de dividir as tarefas complexas em tarefas simples, porém sem o problema de gerar diversas compilações, pois todos os módulos e classes necessários são compilados apenas uma única vez.

Também não é necessário se preocupar com o fato de evocar o compilador todas as vezes que for executar o programa, pois ele é muito estável, seguro e rápido. Mas se você deseja realmente manter seu código-fonte em sigilo existem métodos para transformar seu código-fonte em um executável único. Para isso você precisa do utilitário pp que é fornecido juntamente com o Perl Archive Toolkit no repositório padrão do Perl, o CPAN. Mais adiante aprenderemos a utilizar o CPAN para ampliar as funcionalidades do Perl.

1.3 Conceitos básicos

Os textos de programação que mais me ajudaram tinham uma estrutura muito similar de tratar uma linguagem de programação à maneira que eu vou usar aqui. Para evitar que nossa viagem se torne chata e pouco frutífera, vamos fazer uma abordagem rápida e pouco profunda dos principais pontos da linguagem. Terá muitos exemplos e irá permitir que executemos programas simples e práticos que podem ser utilizados em atividades corriqueiras. Assim, irei trazer para nossa discussão pontos realmente importantes da programação em geral de forma gradual. Não se preocupe se alguns pontos discutidos a seguir não ficarem muito claros, pois todos os tópicos deste módulo serão revistos em profundidade e detalhe.

1.3.1 Perl, variáveis, constantes e dados

Dados, numa definição bem simples e humana, são fatos pura e simplesmente. E informações são os dados processados. Por exemplo: se o repórter do tempo diz que está 30º C isto é um dado. É um fato comprovado e medido por um termômetro. Mas se ele diz que o dia hoje está quente, isso é uma interpretação do dado. Um processamento sobre o dado. Isto se chama informação. E normalmente você quer que o seu software processe dados e os transforme em informações.

Vamos supor um programa em Perl que some dois números. Os dois números seriam os dados, a soma o processamento requerido e o resultado a informação desejada. Em programação, chamamos os dados que fornecemos ao programa, bem como o resultado processado, de variáveis.

```
1 #!/usr/bin/perl
 2 #exemplo1.1
 3
 4 $numero1 = 5;
 5 $numero2 = 10;
 6 $resultado = $numero1 + $numero2;
 7 print "$numero1 + $numero2 = $resultado\n";

saída:
5 + 10 = 15
```

Em termos de linguagem podemos pensar nas variáveis como um substantivo. **Um nome que tem um valor ou significad .** Um nome que remete a algo, que significa algo. Por exemplo: minha mãe me batizou como "Thiago Glauco". Agora sempre que alguém diz "Thiago Glauco" sou eu. O substantivo "carro" remete/significa aquele objeto que tem quatro rodas, um motor à combustão que serve para levar as pessoas de um lugar a outro.

Quando eu disse ao Perl que ***'$numero1 = 5'*** estou gerando uma palavra, um nome que é ***$numero1*** e significa o valor 5. Eu posso fazer isso com números, textos ou arquivos. Por exemplo:

```
1 #!/usr/bin/perl
  2 #exemplo1.2
  3
  4 $perl_creator = "Larry Wall";
  5 print $perl_creator,"\n";

saída:
Larry Wall
```

Eu defini o nome ***$perl_creator*** e informei ao Perl que isso significa ***"Larry Wall"***.

As variáveis, dentro da programação, têm a finalidade de armazenar, ou seja, significar um determinado dado para que ele possa ser processado no tempo devido.

NOTA: Tecnicamente falando, uma variável é um nome pelo qual um programador pode se referir a um espaço na memória do computador que deve armazenar um determinado valor. Este valor pode ser numérico, um conjunto de caracteres ou um arquivo.

A esta altura você, como um leitor atento, percebeu que o sinal igual '=' significa em Perl o mesmo que o sinal matemático '=' ou o mesmo que ***'equivale a***' ou mesmo ***'significa'***. Ele é chamado de operador de atribuição porque atribui significado/valor às variáveis.

$numero1 = 5;
$numero1 significa/é igual a/é o mesmo que/vale 5;
$perl_creator = "Larry Wall";
$perl_creator significa/é igual a/é o mesmo que/vale "Larry Wall";

Outra noção importante das variáveis dentro da programação é o poder do programador de modificar seu valor/significado durante a execução do programa ou entre diversas execuções. O exemplo 1.3 demonstra essa característica das variáveis:

```
  1 #!/usr/bin/perl
  2 #exemplo1.3
  3
  4 $numero1 = 5;
  5 $numero2 = 10;
  6
  7 $resultado = $numero1 + $numero2;
  8 print "$resultado\n";
  9
 10 $numero1 = 10;
 11 $numero2 = 5;
 12 $resultado = $numero1 * $numero2;
 13
 14 print "$resultado\n";

saída:
15
50
```

Perceba que o programa compila e executa perfeitamente. E as variáveis ***$numero1*** e ***$numero2*** foram inicialmente significadas com 5 e 10 respectivamente e depois foram alteradas para 10 e 5. E $resultado também teve seu significado alterado. Ou seja, teve seu valor alterado durante a execução do programa.

Esse recurso é importante quando temos que processar diversos dados diferentes mas não ao mesmo tempo. Assim podemos economizar espaço na memória do computador em vez de criar uma variável para cada valor que deve ser processado.

As constantes, por sua vez, são valores que, uma vez definidos, não podem ser alterados. Você pode usar variáveis normalmente como constantes e manter seu valor inalterado. É simples, prático e funcional. Porém alguns programadores gostam de deixar claro que determinados significados não podem ser modificados. Também é mais seguro deixar um valor definido como constante para evitar que alguém de forma inadvertida faça uma alteração no código que se torne difícil de consertar.

```
  1 #!/usr/bin/perl
  2 #exemplo1.4
  3
  4 use constant PI => 4 * atan2(1, 1);
  5 use constant EMPRESA => "MinhaTerra SA";
  6
  7 print "O valor de PI é: ", PI, "\n";
  8 print EMPRESA,"\n";

saída:
O valor de PI é: 3.14159265358979
MinhaTerra SA
```

Este exemplo é executado normalmente. Porém, se você tentar modificar algum valor constante, vai gerar um erro de compilação: "Can't modify constant item in scalar". Para ver essa característica basta adicionar uma linha: Empresa = "NossaTerra SA" e ver o erro de compilação.

Você deve ter reparado que para dar o significado a uma constante não usamos o símbolo = e sim => e usamos a diretiva ***'use constant'.*** Todas as diretivas ***'use'*** são avaliadas durante a compilação.

1.3.2 Tipos de dados no Perl

Se você vem de outra linguagem de programação, alguma "fortemente tipada" como o C ou o Java, deve estar estranhando a facilidade de criar variáveis a qualquer tempo no Perl sem ter de identificar o tipo. Se não vem, sorte sua. Mas apenas para deixá-lo a par do assunto: em diversas linguagens devemos identificar o tipo de dado que uma variável vai armazenar: se é um número, um caractere, um texto com muitos caracteres, um número inteiro, um número racional, um número muito grande ou um número muito pequeno... uma complicação desnecessária. Tarefa que o interpretador do Perl terá prazer em cuidar por nós programadores.

No Perl existem três ***tipos de dados básicos***. Os ***escalares, do inglês scalars, os arrays e os hashes***. Os escalares são especialmente elaborados para armazenar números (sejam eles inteiros, racionais, positivos ou negativos), textos (seja um

simples caractere ou um texto longo), referências, objetos e filehandles que serão discutidos mais adiante. Em nossos exemplos anteriores todas as variáveis eram do tipo escalar. O Perl identifica o tipo escalar pelo prefixo $ que antecede o nome da variável. Pense no $ como o s de eScalar.

Os arrays, também conhecidos como vetores, listas ou arranjos, armazenam um grupo de escalares organizados por índice numérico iniciando por zero. Os arrays são identificados pelo compilador Perl através do prefixo arroba (@). Pense no @ como o a da palavra Array. Como não tivemos nenhum exemplo com arrays até o momento, vamos mostrar um.

```
  1 #!/usr/bin/perl
  2 #exemplo1.5
  3
  4 @meus_sonhos = ('casa', 'carro', 'apartamento');
  5 @meus_numeros = (1, 2, 3, 4, 10, 100);
  6
  7 print "meus números são @meus_numeros\n";
  8 print "$meus_sonhos[2] vale $meus_numeros[5]\n";
  9 print "Soma de $meus_numeros[2] com $meus_numeros[4] é:
",
 10 $meus_numeros[2]  + $meus_numeros[4];

saída:

meus números são 1 2 3 4 10 100
apartamento vale 100
Soma de 3 com 10 é: 13
```

Perceba que, como foi dito, os nomes dos arrays iniciam com @. Podemos pensar nos arrays como substantivos coletivos ou o plural da linguagem Perl. O sinal '=' é o mesmo que define o valor/significado nos valores escalares e arrays. Os valores são **separados por vírgula** e colocados entre parênteses ( ). Mas para acessar um conteúdo específico, precisamos determinar seu índice – sua posição – no grupo. Por exemplo, **100** é o item de índice **5** do array ***@meus_números***. E como um item é um valor único, refere-se a ele como um escalar, porém com um índice **[5]**. Assim o interpretador sabe o valor ao qual você se refere dentro do array.

Já os ***hashes*** são um tipo bem especial de variável. Também conhecidos como dicionários ou tabelas, eles são similares aos arrays, porém seus índices são ***chaves e podem ser textos ou números***. O prefixo de um ***hash*** é o símbolo de porcentagem %.

```
  1 #!/usr/bin/perl
  2 #exemplo1.6
  3
  4 %tabela_host = ( Servidor => '192.167.2.21',
  5                        Computador1 => '192.167.2.2',
  6                        Computador2 => '192.167.2.3',
  7                        Roteador => '192.167.2.254');
  8
  9 print "$tabela_host{Servidor}\n";
 10 print "$tabela_host{Roteador}\n";
saída:
192.167.2.21
192.167.2.254
```

O leitor, sempre atento, mais uma vez percebeu que o ***hash*** relaciona a chave com o valor pelo símbolo =>, o mesmo símbolo usado na diretiva ***'constant'.*** Isso ocorre porque, internamente, ***'constant'*** usa um ***hash*** como argumento (veremos argumentos com detalhes no capítulo que trata de funções). O item à esquerda do símbolo '=>' é a ***chave*** e o item à direita é o ***valor relacionado à chave***. E, como nos arrays, para acessar o valor de um item individual, usamos o prefixo ***$***, pois se trata de um ***escalar***. Porém, os valores do ***hash*** são acessados por { } e dos arrays por [ ]. Pelos símbolos em torno do índice o compilador saberá diferenciar quando se trata de um array, um hash ou um escalar.

1.3.3 Comunicando-se com o mundo externo

É muito triste ser sozinho. Um programa que não pode se comunicar com o mundo externo é mais triste ainda. É praticamente inútil. Eu pelo menos não enxergo uma utilidade prática para um programa que não receba dados, os processe e apresente o resultado do processamento. E você? Aliás... como você percebeu, a função ***'print'*** tem sido a ferramenta que nossos programas usaram para apresentar os dados processados a você. E isso já resolve metade de nossos problemas. Temos como apresentar os dados para a saída. Mas e a entrada?

Vamos começar falando do conceito de ***filehandle*** – vamos falar a fundo disso em capítulos posteriores. No Perl toda comunicação com o mundo exterior é feito via ***filehandle***. Isso é bom, porque uma vez que você aprende a "manusear" os ***filehandles*** você estará apto a controlar as entradas e saídas de um programa, seja pelo teclado, mouse, placa de rede ou até mesmo entre processos. Todo programa Perl, se não definido o contrário pelo programador, tem três ***filehandles*** abertos: ***STDIN, STDOUT e STDERR***.

STDOUT é a saída padrão, que é o seu monitor quando você está trabalhando numa linha de comando. Você pode configurar a saída padrão para outra localidade, mas não é o escopo deste capítulo. A sintaxe completa do comando **'print'** é ***'print FILEHANDLE STRING'***. Quando o ***filehandle*** for ***STDOUT*** ele pode ser implícito na função, como fizemos até agora.

STDERR é o filehandle padrão para saída de erros, que normalmente também é o seu monitor. O que diferencia a saída padrão da saída de erros? Se sua linha de comandos for configurada para exibir as saídas de erro com uma cor diferente da usada pela saída padrão, você vai perceber visualmente quais saídas são motivadas por algum erro e quais são saídas normais do programa. Você também pode optar por redirecionar a saída de erros para um arquivo de logs com o comando "2>" no shell linux, por exemplo.

O programador também pode modificar o local das saídas de erros no código do programa. Assim as saídas para ***STDERR*** poderão ser enviadas não somente para um arquivo de logs, mas para um servidor de banco de dados ou mesmo um sistema de supervisão. Mas isto nós veremos mais adiante. Vamos fazer um exemplo simples para você entender as saídas.

```
1 #!/usr/bin/perl
  2 #exemplo1.7
  3
  4 print STDOUT "Olá - esta é a STDOUT\n";
  5 print "Olá - esta também é a STDOUT\n";
  6 print STDERR "Oi - esta é uma mensagem de erro!\n";
  7 print "A mensagem anterior representa uma mensagem de
erro\n";

saída:
Olá - esta é a STDOUT
Olá - esta também é a STDOUT
Oi - esta é uma mensagem de erro!
A mensagem anterior representa uma mensagem de erro
```

Mas ainda falta uma coisa. Precisamos entrar dados nos programas, ou vamos ter de alterar as variáveis no código-fonte todas as vezes que quisermos utilizar dados diferentes. Isso não seria muito prático para nós e seria inviável para programas comerciais. Para entrada de dados temos a "entrada de dados padrão" ***STDIN,*** que é o teclado, a menos que configurada para ser outra coisa pelo programador, ou redirecionada de um arquivo via "<" no shell. Então vamos fazer nosso primeiro programa, que pega dados do usuário.

```
  1 #!/usr/bin/perl
  2 #exemplo1.8
  3
  4 print "Digite o seu nome: ";
  5 $nome = <STDIN>;
  6 print "Bom dia, $nome. Que tal aprender mais um pouco
de Perl hoje?";

saída:
Digite o seu nome: Thiago Glauco Sanchez
Bom dia, Thiago Glauco Sanchez
. Que tal aprender mais um pouco de Perl hoje?
```

Note que ***STDIN*** está entre <> , que é chamado de **operador diamante**. Ele informa que o Perl deve receber dados deste ***FILEHANDLE***. Nosso programa tem um probleminha ainda. Quando o usuário digita o nome e tecla 'enter', o caractere de nova linha é adicionado ao dado de entrada, que gera uma nova linha indesejada na saída. Para resolver isto criaram a função **'chomp'**, que remove o último caractere de uma string se este for um caractere de nova linha. Então, nosso novo programa é:

```
   1 #!/usr/bin/perl
   2 #exemplo1.9
   3
   4 print "Digite o seu nome: ";
   5 chomp($nome = <STDIN>);
   6 print "Bom dia, $nome. Que tal aprender mais um pouco
de Perl hoje?";

saída:
Digite o seu nome: Thiago Glauco Sanchez
Bom dia, Thiago Glauco Sanchez. Que tal aprender mais um
pouco de Perl hoje?
```

Claro que a entrada do usuário pode ser numérica também. Veja:

```
  1 #!/usr/bin/perl
  2 #exemplo1.10
  3
  4 print "Vamos somar? Digite o primeiro número da soma:
";
  5 chomp($numero1 = <STDIN>);
  6 print "Agora digite o segundo: ";
  7 chomp ($numero2 = <STDIN>);
  8 $soma = $numero1 + $numero2;
  9 print "$numero1 + $numero2 = $soma\n";

saída:
Vamos somar? Digite o primeiro número da soma: 5
Agora digite o segundo: 10
5 + 10 = 15
```

1.3.4 Operadores

Como já dissemos, a informação é o resultado do processamento dos dados. Os dados fornecidos pelo usuário a um programa devem ser processados. Quem realiza a tarefa de processar os dados são os ***operadores***. Existem os operadores ***matemáticos***, que realizam operações como somas e divisões, os ***operadores de comparação*** comparam os valores – maior que, menor que, igual a. Os operadores de comparação são muito importantes nos processos de tomada de decisão que veremos mais adiante. Os operadores de strings realizam operações de concatenação e modificação em strings.

1.3.4.1 Operadores Matemáticos

Exemplo	Operador
$x + $y	soma
$x * $y	multiplicação
$x % $y	módulo
$x / $y	divisão
$x ** $y	exponencial
$x = $y	atribuição

Os operadores matemáticos funcionam no Perl da mesma forma que na matemática. Tenho que fazer uma ressalva para o operador de atribuição. $x = $y significa atribuir o valor de $y a $x. Isso significa que ele é executado da direita para a esquerda. Veja o exemplo a seguir:

```
 1 #!/usr/local/bin/perl
 2 #exemplo1.11
 3
 4 $x = 0;
 5 $y = 10;
 6
 7 print $x + $y, "\n";
 8 print "O valor de $x e o valor de $y.\n";
 9
10 $x = $y;
11
12 print "O valor de x é $x e o valor de y é $y.\n";

saída:
10
O valor de 0 e o valor de 10.
O valor de x é 10 e o valor de y é 10.
```

Perceba que $x = $y atribuiu o valor de $y à variável $x, e não o contrário. A **execução do operador de igualdade é da direita para a esquerda.**

O operador módulo (%) também merece nossa atenção. Ele retorna o resto inteiro de uma divisão. Por exemplo, **3/2 resulta em 1 com resto 1. 10/3 resulta em 2 com resto 1. 8/3 é 2 com resto 2.**

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.12
  3
  4 $x = 3;
  5 $y = 10;
  6
  7 print $y/$x,"\n";
  8 print int($y / $x), "\n";
  9 $mod = $y%$x;
 10 print "O resto inteiro da divisão $y/$x = $mod\n";

saída:
3.33333333333333
3
O resto inteiro da divisão 10/3 = 1
```

NOTA: A função int ( ) retorna a parte inteira de um número dado como argumento da função. Não realiza o arredondamento.

1.3.4.2 Operadores de comparação

Operadores de comparação são como uma pergunta que o programador faz para o interpretador. Por exemplo: '$x > $y'. A variável x é maior que y? Ou '$x == $y' "A variável $x é igual à variável $y"? Se a resposta for sim, o Perl irá retornar 1; se não, não irá retornar nada. Pense nos operadores lógicos como perguntas do tipo Sim ou Não. Responda: 2 é maior que 3? Não. 2>3 retorna: ''. Responda: 2 é menor que 3? Sim. 2<3 retorna: '1'.

Exemplo	Operador
$x == $y	igualdade (é igual a?)
$x != $y	Diferença (é diferente de?)
$x > $y	Maior (é maior que?)
$x < $y	Menor (é menor que?)
$x >= $y	Maior ou igual (é maior que ou igual a?)
$x <= $y	Menor ou igual (é menor que ou igual a?)

```
 1 #!/usr/local/bin/perl
 2 #exemplo1.13
 3
 4 $var1 = 53;
 5 $var2 = 26;
 6 $scalar = 5<3;
 7 print 5>3,"\n";                    #retorna 1
 8 print "$scalar \n";               #Não retorna nada
 9 print $var1 != $var2,"\n";       #retorna 1
10 print 5==4,"\n";                 #Não retorna nada
11 print $var1 > $var2,"\n";  #retorna 1
12 print $var1 < $var2,"\n";  #Não retorna nada

saída:
1
1
1
```

Antes de continuarmos quero ressaltar alguns detalhes sobre o operador de igualdade '==‘. Este é comumente confundido com o operador de atribuição '=‘. O operador de igualdade (==) apenas verifica se os valores de duas variáveis são iguais. O operador de atribuição, como já foi dito, substitui o valor da variável à esquerda pelo valor da variável à direita.

Além dos operadores numéricos existem os operadores lógicos para strings de texto, ou seja, escalares que têm um string como valor atribuído.

Exemplo	Operador
`$str1` **`eq`** `$str2`	**`equal:`** é igual a?
`$str1` **`ne`** `$str2`	**`not equal:`** é diferente de?

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.14
  3
  4 $str1 = "Olá Amigo";
  5 $str2 = "Adios Amigo";
  6 $str3 = "Olá Amigo";
  7
  8 print $str1 eq $str3,"\n";
  9 print $str1 ne $str2, "\n";
 10 print "Digite uma frase: ";
 11 chomp ($str4 = <STDIN>);
 12
 13 print "Você digitou $str4.\n";
 14 print $str4 eq $str1,"\n";
 15 print $str4 ne $str1,"\n";

saída:
1
1
Digite uma frase: Olá Amigo
Você digitou Olá Amigo.
1

1
1
Digite uma frase: Ei você
Você digitou Ei você.

1
```

Se você está pensando qual a intenção de ficar verificando valores e receber retornos como 1 e '' – bem, nada –, as suas questões serão respondidas agora.

1.3.5 Tomada de decisão

A aparente inteligência dos programas das mais diferentes linguagens provém da capacidade do programador de prever situações e criar fluxos de programa diferentes para cada caso. O elemento essencial de tomada de decisão no Perl é o if. O if, da palavra 'se' em inglês, é justamente isso: uma condição.

A estrutura é mostrada a seguir:

if (PERGUNTA) { Código a ser executado se a resposta for 1 - verdadeira }

Pergunta é uma operação lógica, como visto anteriormente. Podemos comparar números e textos.

Vamos supor um programa que solicite a idade do usuário. Então ele compara a idade com um valor predeterminado. Digamos 18. Se for maior ou igual a 18, imprima "você é maior de idade". Se for menor que 18 imprima: menor de idade.

Para isso servem os operadores lógicos vistos anteriormente. Eles respondem o "Se a idade for menor que 18 pare o programa." "Se for maior, continue".

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.15
  3
  4 use constant IDADE_MINIMA => 18;
  5
  6 print "Este programa não é recomendável para menores.
Qual sua idade? ";
  7 chomp($idade = <STDIN>);
  8
  9 if ($idade < IDADE_MINIMA){
 10         print "Este programa não pode continuar.\n";
 11         exit;
 12 }
 13
 14 print "Você pode continuar.\n";

saída:
Este programa não é recomendável para menores. Qual sua
idade? 25
Você pode continuar.

Este programa não é recomendável para menores. Qual sua
idade? 13
Este programa não pode continuar.
```

Como o conceito de tomadas de decisão é muito importante, eu vou explicar aqui detalhadamente o programa anterior. O começo é bem simples. Você cria uma constante IDADE_MINIMA que representa o valor 18. É uma constante porque esta idade será a mesma em todas as execuções do programa. Como maioridade é uma definição legal que muda raramente, é um caso que vale o esforço de usarmos uma constante. Depois que pedimos para o usuário informar a sua idade, damos a este valor o nome de ***$idade***. Então pedimos que o Perl cheque se o valor de ***$idade*** é menor que IDADE_MINIMA.

if ($idade < IDADE_MINIMA). Caso a comparação retorne 1, ou seja, é verdadeira, o Perl irá executar o código entre colchetes { } e encerrar o programa. A função ***exit*** serve para encerrar o programa. Se a comparação for falsa irá retornar "nada" e o Perl não executará o código entre { }.

Outro bom exemplo é a comparação com nomes de usuários em um programa restrito. Se o nome não estiver cadastrado o programa não executa.

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.16
  3
  4 @usuario = ('Frodo','Eduardo','Mônica');
  5
  6  print "Digite o nome de usuário: ";
  7  chomp($nome = <STDIN>);
  8
  9  if ($nome eq $usuario[0]){
 10         print "Olá, $nome!\nO chefe te espera\n";
 11         #Aqui executa coisas apenas para o usuário Frodo
 12         exit;
 13  }
 14  if ($nome eq $usuario[1]){
 15         print "Olá, $nome.\nNão esqueça de ligar para sua
esposa.";
 16         #Aqui executa código apenas para o usuário Eduardo
 17         exit
 18  }
 19  if ($nome eq $usuario[2]){
 20         print "Olá, $nome...\nPor que você não veio on-
tem?";
 21         #Aqui executa código apenas para o usuário Mônica
 22         exit;
 23  }
 24
```

```
 25  print "Você não é usuário deste programa!!!"
saída:

Digite o nome de usuário: Bilbo
Você não é usuário deste programa!!!

Digite o nome de usuário: Frodo
Olá, Frodo!
O chefe te espera

Digite o nome de usuário: Eduardo
Olá, Eduardo.
Não esqueça de ligar para sua esposa.

Digite o nome de usuário: Monica
Olá, Monica...
Por que você não veio ontem?
```

Nesse programa criamos um ***array @usuario***. O array contém os nomes dos usuários autorizados a rodar o programa. O programa solicita um nome ao usuário. Depois o nome solicitado é armazenado em $nome e comparado em cada ***'if'*** aos elementos do array: ***$nome eq $usuario[0], $nome eq $usuario[1] e $nome eq $usuario[2]***. Claro que você se lembra que eu disse que o primeiro índice de um array é [0]. Quando o Perl encontrar uma comparação que retorne '1' ele executará o código que está entre os colchetes { }.

O exemplo mostrou quatro execuções do programa. Na primeira usei o nome Bilbo, que não está cadastrado. O programa retornou a mensagem correta: "Você não é usuário deste programa!!!". Nas outras execuções utilizei nomes cadastrados e para cada nome eu obtive a mensagem correta para o usuário.

1.3.6 Loops

Computadores são muito bons em matemática. Mas em uma coisa eles são melhores que em matemática: repetição. Eles são capazes de repetir uma tarefa dezenas de milhares de vezes num segundo. E isso sem pedir férias, assistência médica ou participação nos lucros. Tudo que as empresas querem. Repetir, repetir e repetir. Milhares de vezes num segundo. Opa... eu já disse isso!

A estrutura ***while*** provê esta repetição. ***while(VERDADEIRO) { EXECUTE O CÓDIGO ENTRE COLCHETES }***. Em detalhes, vamos supor: while significa 'enquanto'. Então, digamos:

Enquanto $x for menor que $y imprima $x é menor que $y e some 1 a $x.

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.17
  3
  4 $x = 0;
  5 $y = 10;
  6
  7 while ($x < $y){
  8         print "$x é menor que $y.\n";
  9         $x = $x + 1;
 10 }

saída:
0 é menor que 10.
1 é menor que 10.
2 é menor que 10.
3 é menor que 10.
4 é menor que 10.
5 é menor que 10.
6 é menor que 10.
7 é menor que 10.
8 é menor que 10.
9 é menor que 10.
```

Outro exemplo: enquanto o usuário não adivinhar a palavra secreta, não continuar o programa.

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.18
  3
  4 use constant SENHA => 'por-favor';
  5
  6 print "Informe a senha: ";
  7 chomp($tentativa = <STDIN>);
  8
  9 while ($tentativa ne SENHA){
 10         print "Senha incorreta. Tente novamente: ";
 11         chomp($tentativa = <STDIN>);
 12 }
 13
 14 print "Senha correta. Iniciando sistema...";
```

```
saída:
Informe a senha: cachorro
Senha incorreta. Tente novamente: gato
Senha incorreta. Tente novamente: sol
Senha incorreta. Tente novamente: desculpe
Senha incorreta. Tente novamente: anda logo
Senha incorreta. Tente novamente: por-favor
Senha correta. Iniciando sistema...
```

1.4 Debug básico

Nos primórdios da era da computação, quando os computadores não eram feitos de pastilhas de silício, mas de válvulas e relés, era comum que estas máquinas tivessem dimensões que hoje seriam, no mínimo, grotescas.

O Mark I, primeiro computador eletromecânico da história, tinha dezoito metros de largura por dois metros de comprimento, 70 toneladas de peso e 800 km de fios e cabos. Já o ENIAC, primeiro computador eletrônico, tinha cerca de 27 toneladas e realizava operações na ordem do que hoje chamaríamos de 5khz... seu computador hoje deve ter entre 1 a 3 Ghz... daí pra mais. Comparando em números, teremos:

5.000 palavras por segundo – ENIAC

2.000.000.000 palavras por segundo– meu PC de mesa.

De forma que, nessas máquinas imensas, era comum a presença de insetos que danificavam componentes. E um dos processos para detectar erros em um programa – quando um programa tem uma atitude inesperada – era buscar e remover insetos (bug) de dentro das máquinas. Daí a expressão debug no mundo da programação quando estamos buscando e removendo erros de um programa.

A forma de debug inicial que você deve aprender é erro de sintaxe. Um erro de sintaxe ocorre quando o programador comete um erro de digitação ou esquece algum item fundamental, como o ponto-e-vírgula no final da sentença, ou esquece de fechar os parênteses em uma expressão. Os erros de sintaxe que você deve conhecer por agora são:

1.4.1 Missing semicolon (você esqueceu o ponto-e-vírgula)

Veja o exemplo a seguir:

```
  1 #!/usr/local/bin/perl
  2 #exemplo1.19
  3
  4 $var1 = "Oi, galera\n";
  5 $var2;
  6
  7 print $var1;
  8 print "Oi dinovo\n"
  9
 10 $var2 = 5;
 11
 12 print $var2 * 3, "\n";
 13 print "$var2\n";

saída:
Scalar found where operator expected at exemplo1.19.pl line
10, near "$var2"
     (Missing semicolon on previous line?)
syntax error at exemplo1.19.pl line 10, near "$var2 "
Execution of exemplo1.19.pl aborted due to compilation errors.
```

Observe a parte em negrito. O compilador do Perl informou a linha e a provável causa do erro. **Line 10... M issing semicolon on previous line?** Se colocarmos o ponto-e-vírgula ao fim da linha anterior, no caso a linha 8, veremos que o compilador executa corretamente.

1.4.2 Runaway strings

Vamos mudar a terceira linha do script anterior e remover o fechamento das aspas:

```
  3 $var1 = "Oi, galera\n ;
$./ exemplo1.19.pl

saída:
Bareword found where operator expected at Livro_exem-
plo1.19.pl line 8, near "print "Oi"
  (Might be a runaway multi-line "" string starting on line
4)
     (Do you need to predeclare print?)
```

Um **erro runaway multi-line "" string** significa que existem aspas abertas em uma linha que não foram fechadas corretamente, e isto está atrapalhando o compilador a definir o que são funções da linguagem e o que são variáveis fornecidas ao programa. O Perl está dizendo: "se você não teve a intenção de fazer uma string de várias linhas veja a string na linha 3. Se a string tem realmente mais de uma linha o fechamento das aspas está confuso e deve ser revisto."

1.4.3 Outros erros

Em alguns casos as dicas do compilador não serão tão precisas. Quando o compilador não conseguir supor o que você deseja fazer ele simplesmente dirá: **Syntax error at <seu script> line xx, near <declaração de referência>**. Por exemplo, modifique o exemplo1.19 para o código a seguir:

```
 1 #!/usr/local/bin/perl
 2 #exemplo1.20
 3
 4 $var1 = "Oi, galera\n"
 5
 6 print $var1;
 7 print "Oi dinovo\n";
 8
 9 $var2 = 5;
10
11 print $var2  3, "\n";
12 print "$var2\n";

saída:
syntax error at exemplo1.19.pl line 6, near "print"
Execution of exemplo1.19.pl aborted due to compilation errors.
```

O compilador acusa erro na linha 6, quando na verdade nos esquecemos do ponto-e-vírgula na linha 4. Por que isso? Porque deve haver um operador entre a string "Oi, galera\n" e a função **print**, não necessariamente o ponto-e-vírgula. Pode ser apenas um operador de concatenação de strings como o operador ponto (.). Daí o valor de $var1 será "Oi, Galera\n1".

```
  4 $var1 = "Oi, galera\n" .
  5
  6 print $var1;
  7 print "Oi dinovo\n";
  8 print $var1;  #linha adicionada para mostrar o valor da
string $var1

saída:
Oi dinovo
Oi, galera
15
```

Como foi dito, as funções podem executar tarefas, retornar valores ou ambos. A função **print** executa uma tarefa e retorna 1 se esta tarefa foi executada com sucesso. O valor 1 é concatenado à string pelo operador ponto (.).

Em contrapartida, se adicionarmos um operador **and** na mesma posição do ponto-e-vírgula teremos:

```
  4 $var1 = "Oi, galera\n" and
  5
  6 print $var1;
  7 print "Oi dinovo\n";

saída:
Oi, galera
Oi dinovo
5
```

O Perl permite diversos operadores nesta situação; cada operador irá gerar uma saída diferente para o usuário. O interpretador não sabe exatamente o que deveria ser executado no exemplo1.19 porque as possibilidades são tantas quantos os operadores da linguagem.

1.4.4 Comando perl -c

Quando você quiser checar a sintaxe de um script antes de executá-lo, existe o switch -c no comando perl. Esta switch faz com que o comando não execute o script e apenas verifique erros de sintaxe para análise. Exemplo:

```
$perl -c ./ exemplo1.19.pl
./exemplo1.19.pl  syntax OK
```

Claro que aqui eu corrigi todos os erros do script.

Exercícios

1. Monte um programa Perl que imprime a soma, o produto, a diferença e o quociente de valores informados pelo usuário.

2. Monte um programa Perl que calcula a área de uma figura geométrica solicitada pelo usuário.

3. Monte um programa Perl que separe os dígitos de um número inteiro informado pelo usuário.

4. Monte um programa Perl que verifica se um número é par ou ímpar.

5. Monte um programa Perl que separe os dígitos de um número informado pelo usuário e imprima uma linha com tantos asteriscos '*' quanto for o valor do número.

6. Para transformar um algoritmo em um programa é preciso o uso de pelo três estruturas de programação. Pesquise na internet ou em livros quais são estas estruturas.

7. Pesquise em perldoc perlintro quais as estruturas básicas de controle do Perl.

8. O Perl possui diversas variáveis iniciadas pelo próprio interpretador. Elas podem ser úteis em diversos programas. Treine a leitura de páginas de documentação Perl com o comando perldoc perlvar.

9. Uma das brincadeiras de adolescentes são as frases que são idênticas mesmo com os caracteres invertidos. Faça um algoritmo em Perl que verifique se uma sequência de números digitados pelo usuário é idêntica em ordem inversa. Ex.: 313 é idêntico mesmo com os caracteres em ordem inversa.

10. Faça um programa que converta um número binário informado pelo usuário em um número decimal.

11. Escreva um script que solicite um número para o usuário indefinidamente e a cada nova entrada o programa calcule qual o menor, o maior e a média dos valores digitados até o momento.

12. Monte um pequeno programa que solicite ao usuário uma frase ou caractere e um número. Então o programa irá repetir a frase digitada tantas vezes quantas o usuário informou no número.

Capítulo 2. Funções, Módulo, Referências, Objetos e Debug

No capítulo anterior apresentamos muitos conceitos novos para os iniciantes em programação, o que deve ter sido um importante desafio. Neste capítulo vamos ir mais além nestes princípios. Vamos falar de funções, módulos e diretivas de compilação.

2.1 Funções

Anteriormente, utilizamos funções internas ao Perl, como ***print, int e chomp***. Mas ainda não havia me dado ao trabalho de explicar funções e para que elas servem. **Funções são porções de código reutilizáveis**. É um bloco de código associado a um nome, como as variáveis. A utilidade das funções é evitar retrabalho ao programador. As funções internas, ou seja, as que já vêm com o Perl, são funções muito importantes e úteis para qualquer programador. Imagine ter que escrever e reescrever códigos toda vez que precisar realizar tarefas básicas como imprimir dados na tela?

2.1.1 Funções internas do Perl

Larry Wall, em seu livro "Programming Perl", apresentava as funções internas por categoria. Então existem funções internas para lidar com **strings**, **números, controle de arquivos** e uma infinidade de outras coisas, de forma que a partir de agora utilizaremos mais funções internas, que serão apresentadas gradativamente conforme o decorrer do livro.

As funções internas têm a seguinte sintaxe:

nome_da_função argumento1, argumento2, … argumento_n;

Vamos utilizar nossa bem conhecida função **print** como exemplo. Dentro da documentação do Perl, a utilização da função **print** é mostrada a seguir:

print FILEHANDLE LIST

print LIST

Onde **FILEHANDLE** e **LIST** são os argumentos. No capítulo anterior dissemos que quando o **FILEHANDLE** for **STDIN** não há necessidade de identificarmos explicitamente o **FILEHANDLE**. Argumentos, numa definição mais formal, são "Variáveis passadas a uma função". São os dados que queremos que a função processe. Na documentação do Perl, LIST significa que a função pode receber uma lista de argumentos. Ou que internamente os dados passados serão utilizados como listas (arrays ou hashes).

Exemplo:

```
  1 #!/usr/local/bin/perl
  2 #exemplo2.1
  3
  4 print "Um elemento ", "Outro Elemento ", " E um tercei-
ro elemento\n";
  5 print "Digite seu primeiro nome: ";
  6 chomp($nome = <STDIN>);
  7 print "Digite seu sobrenome: ";
  8 chomp($sobrenome = <STDIN>);
  9 print "Olá ", $nome, " ", $sobrenome,"\n";

saída:

Um elemento Outro Elemento  E um terceiro elemento
Digite seu primeiro nome: Julius
Digite seu sobrenome: Cesar
Olá Julius Cesar
```

Perceba que a função **print** pode receber **strings de texto** separados por vírgula e envia-os para a tela como um único bloco de texto. Ou seja, ela recebe uma lista de argumentos, processa e envia para STDOUT.

2.1.2 Funções definidas pelo usuári

Muitas vezes você irá criar suas próprias funções, seja para organizar o código, seja para reutilizar o código. Para criar uma função, siga este procedimento:

sub minha_função {

bloco_de_código

}

```
1 #!/usr/local/bin/perl
2 #exemplo2.2
3 $time = hora_brasil ( localtime ());
4 print "$time\n";
5
6 sub hora_brasil(){
7         $_[5] = $_[5] + 1900;
8         return qq($_[2]:$_[1]:$_[0] $_[3]/$_[4]/$_[5]);
9 }
```

saída:

23:51:46 22/7/2011

Os argumentos da função são armazenados na variável interna @_, cujos itens são $_[0], $_[1], $_[2]...

2.2. Variáveis predefinida

O Perl fornece, para facilitar nossos trabalhos, algumas variáveis predefinidas. As mais importantes são:

$	Variável padrão de entrada, loops e match
$.	Linha corrente do filehandle
$@	Mensagem de erro definida pelo eval()
$!	Valor da última mensagem de erro fatal, usada com o operador die
$0	Contém o nome do programa em execução
$$	Contém o número do PID do programa
$^V	A versão do Perl
@ARGV	Contém os argumentos da linha de comando
@INC	Path das bibliotecas do Perl
@_	Contém os argumentos de uma sub-rotina
%ENV	Contém as variáveis de ambiente
%SIG	Contém os handles de sinais de processos

2.3 Módulos

Módulos são arquivos de código Perl com a extensão .pm. Os módulos são usados para armazenar funções e objetos de forma a estender a funcionalidade do Perl e organizar e facilitar a sua vida enquanto programador. O Perl possui diversos módulos internos que facilitam a vida do programador. Para incluir um módulo já instalado no seu código basta utilizar a cláusula use:

use nome_do_modulo "nome_da_funcionalidade";

```
  1 #exemplo2.3
  2 use Scalar::Util "looks_like_number";
  3 #O módulo Scalar::Util já vem na sua instalação padrão
e exporta
  4 #diversas facilidades, como looks_like_number que veri-
fica se o valor
  5 #armazenado em uma variável escalar pode ser usado como
número.
  6
  7 $numero = 100;
  8 if (looks_like_number $numero){
```

```
  9         print $numero, "\n"; #imprime se $numero arma-
zenar um número
 10 }

saída:

100
```

2.4 Instalando módulos CPAN

O CPAN é um repositório central de módulos Perl na internet que possui hoje cerca de 20.000 módulos à disposição dos programadores Perl. Para instalar os módulos você pode usar o comando:

cpan install Nome_do_Modulo

As configurações padrão do comando ***cpan*** servem para a maioria dos casos, e com certeza servem para seu computador doméstico. Mas, em empresas, os servidores e computadores não acessam diretamente a internet. Por isso podemos configurar o shell do comando cpan. Para abrir o shell do cpan digite apenas o comando ***cpan***. Depois de aberto o shell você pode ver a lista de itens passíveis de configuração com o comando ***o conf***. As opções mais importantes estão listadas a seguir:

```
cpan[1]> o conf
$CPAN::Config options from '/usr/local/lib/perl5/5.12.2/
CPAN/Config.pm':
    commit              [Commit changes to disk]
    defaults            [Reload defaults from disk]
    help                [Short help about 'o conf' usage]
    init                [Interactive setting of all options]
#SAÍDA OMITIDA...
    cpan_home           [/home/thiago/.cpan]
 #Diretório onde guardar as informações
    ftp_proxy           [ftp://meuproxy.com]
 #proxy ftp
    http_proxy          [http://meuproxy.com:8080]
 #proxy http
```

```
    pager              [/usr/bin/less]
    prerequisites_policy [follow]
 #instalar dependências
    proxy_pass         [senha_do_proxy] #sua senha do proxy
    proxy_user         [usuario_do_proxy] #seu usuário do
proxy
    shell              [/bin/bash]
    urllist
#espelhos do CPAN
   0 [http://www.cpan.org]
```

Para configurar um item digitamos ***o conf item parâmetro***. Exemplo:

```
cpan[2]> o conf urllist "http://www.cpan.org"
commit: wrote '/usr/local/lib/perl5/5.12.2/CPAN/Config.pm'
```

Outra opção, no Linux, é utilizar o gerenciador de arquivos da sua distribuição:

```
$ sudo aptitude search libsdl-perl
i A libsdl-perl        - SDL bindings for the Perl language
```

Uma terceira opção é baixar o arquivo fonte do módulo compactado no site http://www.cpan.org/modules. Após baixar, basta descompactar o arquivo, entrar na pasta do módulo via prompt (Windows) ou shell (Linux) e executar os comandos a seguir:

```
$ perl Makefile.PL
$ make
$ make test
$ make install
```

NOTA*:* No Windows, substitua o comando ***make*** por ***dmake***. Além do ***cpan***, os programadores Windows podem utilizar o comando ***ppm***.

Caso este procedimento não funcione, leia o arquivo README do módulo.

2.5 Referências

Referências são variáveis escalares que guardam uma referência a outra variável de qualquer tipo. Ou seja, elas armazenam um valor que representa o endereço da variável dentro do seu arranjo de memória.

```
  1 #exemplo2.4
  2 use warnings;
  3 $idade = 25;
  4 @nomes = qw(Frodo Sam Fred Vilma) ;
  5 %conta = ( banco => "Perl Bank" ,
  6              correntista => "Willy Kitty",
  7              saldo => 1500 ,
  8            );
  9
 10 # Criando Ponteiros:
 11 $pointer1 = \$idade;   # Cria ponteiro para um escalar
 12 $pointer2 = \@nomes;  # Cria ponteiro para um array
 13 $pointer3 = \%conta; # Cria ponteiro para um hash
 14 $pointer4 = [ qw(céu árvore mar) ] ; #Cria ponteiro
para um array anônimo
 15 #Cria ponteiro para um hash anônimo
 16 $pointer5 = { SP => "São Paulo", RS=>"Rio Grande do
Sul" };
 17 #removendo a referência:
 18 say $$pointer1; # Imprime o valor de idade
 19 say @$pointer2; # Imprime o conteúdo do array @nomes
 20 say %$pointer3; #  Imprime as chaves e valores do hash
 21 say $pointer2->[1] ;  #Imprime um elemento do array @
nomes
 22 say $pointer3->{"correntista"}; # imprime o valor de
$conta{correntista}
 23 say "@{$pointer4}"; # Imprime céu árvore mar
 24 say %{$pointer5}; #Imprime SPSao PauloRS Rio Grande do
Sul
```

```
saída:

25
FrodoSamFredVilma
correntistaWilly Kittysaldo1500bancoPerl Bank
Sam
Willy Kitty
céu árvore mar
SPSão PauloRSRio Grande do Sul
```

2.6 Objetos

A programação orientada a objetos foi desenvolvida para melhorar as práticas de programação com conceitos como reusabilidade de código, herança, encapsulamento, entre outros. O Perl tem suporte a OOP. O código adiante é apenas um exemplo simples de uma classe que utiliza OOP. Não se preocupe se não entender isso agora, pois conversaremos sobre OOP com calma e detalhe mais adiante.

NOTA: A função ***say*** não existe em versões anteriores à 5.10. Se estiver utilizando Perl 5.8 ou 5.6, substitua **say** por **print** e adicione "\n" ao final do texto que deseja imprimir.

```
   1 use lib (".");
   2 use Pet;
   3 #exemplo2.5
   4
   5 $cachorro = new Pet();
   6 print $cachorro, "\n"; #imprime uma referência para
$cachorro
  7 $cachorro->set_pet("Frodo", "Thiago", "Lhasa-Apso");
  8 $cachorro->get_pet;

Módulo Pet.pm:

  1  package Pet;
  2  #exemplo2.6
  3     sub new{
```

```
  4  # Construtor
  5      $class = shift; #o operador shift será estudado
mais à frente
  6      $pet = { #Cria uma referência a um hash
  7          Nome  => undef,
  8          Dono => undef,
  9          Especie  => undef,
 10      };
 11      bless($pet, $class);
 12     }
 13
 14  #Retorna um ponteiro
 15     sub set_pet{    # Accessor methods
 16                 my $self = shift;
 17                 my ($name, $owner, $type)= @_;
 18          $self->{Nome} = $name;
 19          $self->{Dono}= $owner;
 20          $self->{Especie}= $type;
 21      }
 22      sub get_pet{
 23          $self = shift;
 24          while(($key,$value)=each(%$self)){
 25                 print "$key: $value\n";
 26          }
 27      }
 28  1

saída:

Pet=HASH(0x3e8ed4)
Especie: Lhasa-Apso
Dono: Thiago
Nome: Frodo
```

Porém, o mais importante aqui é aprender a utilizar módulos do Perl que sejam orientados a objetos. Para utilizar uma função ou propriedade em um módulo orientado a objetos utilizamos o operador **->.** Desta forma, vamos a um exemplo:

```
  1  #exemplo2.7
  2  use FileHandle; #Carrego o módulo.
  3
  4  #Crio um objeto com o método new.
  5  $file = FileHandle->new("/etc/passwd", "r");
  6
  7  #O objeto é acessado pela variável $file.
  8  #Um método é o mesmo que uma função
  9  $first_line = $file->getline; #Acesso o método getline
do objeto.
 10  print $first_line;
 11
 12  #Acesso o método getlines, que lê todas as linhas de
um arquivo
 13  @other_lines = $file->getlines;
 14  print "@other_lines";

saída:

 gdm:x:42:42::/var/lib/gdm:/sbin/nologin
 nm-openconnect:x:495:492:NetworkManager user for OpenCon-
nect:/:/sbin/nologin
 mailnull:x:47:47::/var/spool/mqueue:/sbin/nologin
 smmsp:x:51:51::/var/spool/mqueue:/sbin/nologin
 sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/
sbin/nologin
 tcpdump:x:72:72::/:/sbin/nologin
 thiago:x:500:500:thiago glauco:/home/thiago:/bin/bash
 jetty:x:494:491::/usr/share/jetty:/bin/sh
```

2.7 Filehandles

Os filehandles são variáveis escalares que podem ler ou escrever dados em arquivos de forma simples e rápida. Mais uma vez, se você não entender os filehandles agora, aguarde que vamos utilizá-los com calma mais adiante. Este item serve para programadores experientes de outras linguagens que desejam saber como trabalhar com arquivos no Perl.

```
open($file, "<","filename");
open ($file,"<", "/home/ellie/myfile") or die "Can't open
file: $!\n";
```

Para abrir um Filehandle para escrita:

```
open($file, ">","filename");
```

Para abrir um Filehandle para apêndice:

```
open($file, ">>","filename");
```

Para abrir um Filehandle para escrita:

```
open($file, "+<","filename");
open($file, "+>","filename");
```

Fecha o arquivo:

```
close($file);
```

Ler de um arquivo:

```
while(<$file>){ print; }
@lines = <$file>; print "@lines\n";
```

Escrever em um arquivo:

```
open($file, ">","arquivo.txt") or die "Erro ao ler o arqui-
vo: $!\n";
print $file "escrevi no arquivo.\n";
```

Veja um exemplo:

```
  1 #exemplo2.8
  2 open( $file, "<", "/etc/passwd");    #abre /etc/passwd
  3 while(<$file>){    #lê linha à linha. $_ recebe o valor
da linha corrente
  4         print $_;  #imprime a linha
  5 }
```

saída:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
```

Exercícios

1. O **CPAN** possui diversos módulos para trabalhar com números romanos. Escolha um módulo **CPAN**, instale e faça um programa que converta um inteiro informado pelo usuário em um número romano.

2. Temos diversas **APIs** no **CPAN** para o Google. Escolha uma e monte um script que realiza pesquisas no Google.

3. Para melhorar o seu perldoc-fu pesquise a documentação do módulo ***CGI.pm*** e tente montar um script que gera o html de uma página web simples.

4. Pesquise no **CPAN** o módulo ***Net::SMTP***. Se você tiver acesso a um servidor **SMTP** instale este módulo e crie um script que envie e-mails utilizando a interface **OOP** do módulo.

5. Monte um script que imprima na tela um arquivo informado pelo usuário. Faça duas versões, uma usando o módulo Filehandle e outra usando a função **open.**

Capítulo 3. Variáveis Simples: Strings e Expressões Regulares Simples

Discutimos anteriormente que as variáveis do Perl podem ser ***escalares, arrays e hashes.*** Vimos também que o tipo mais básico de variável no Perl é o escalar e que os nomes das variáveis deste tipo iniciam com o caractere ***$***. Neste capítulo vamos fazer um estudo mais a fundo do Perl. Já demos um giro completo pelas suas características fundamentais mas precisamos entrar no detalhe se quisermos ser profissionais e escrever programas realmente robustos.

3.1 Regras para nomes

Falamos brevemente sobre os prefixos para os nomes das variáveis e funções, mas não falamos nada sobre as regras para o nome das variáveis e funções. A grande regra de ouro para nomes de funções e variáveis é **"OS NOMES DEVEM FAZER SENTIDO!!!"**. Evite nomes como ***$n, $alpha, $numero1, $f , $x3po*** a não ser que eles tenham algum sentido no seu programa. Exemplos de nomes que fazem sentido são ***$produto_final, $retorno, $soma_x_y.*** Veja que este livro é um material didático com o intuito de ensiná-lo a programar, mas como livros têm espaço limitado para apresentar o conteúdo proposto, me reservo o direito de usar nomes curtos para variáveis e funções.

A segunda regra diz respeito à construção interna do Perl. Os nomes de variáveis podem ser formados por letras, números e o caractere sublinhado **'_'**. Isso quer dizer que, para dar significado ao nome da minha variável, eu posso usar qualquer combinação de letras e números e sublinhados desde que não comece com números. Caso o nome da variável inicie com números, poderá conter apenas números. Exemplo: ***$153, $675***.

NOTA: O Perl reserva variáveis cujo nome é composto apenas por números para casos especiais. Veja o erro gerado pelo comando a seguir.

A extensão do nome vai de 1 até 251 caracteres, que permite que usemos nomes bem significativos. Mas não é uma boa ideia usar nomes muito longos. Isso é ruim para você codificar.

Caracteres maiúsculos e minúsculos: Perl diferencia as letras maiúsculas das minúsculas. Então ***$thiago*** é, para o interpretador, diferente de ***$Thiago*** ou ainda ***$ThiaGo***. É importante frisarmos este ponto, pois linguagens nascidas no mundo Windows não são sensíveis à caixa da letra.

Na verdade o Perl aceita caracteres especiais como asterisco (*), sustenido (#) ou qualquer outro que quiser. Estes podem ser utilizados apenas no primeiro caractere do nome, e este nome fica limitado a apenas um caractere. Porém, como as variáveis internas do Perl utilizam estes caracteres, **o compilador não permitirá novas variáveis com estes caracteres.**

Além disso, o tipo da variável está sempre associado ao nome pelo prefixo. Então o Perl sempre sabe o que é um array, um escalar e um hash. Da mesma forma, o programador. Isto facilita muito a leitura do programa, uma vez que você sabe exatamente o que foi criado pelo programador e o que é nativo da linguagem. Essa característica permite outra funcionalidade interessante. Variáveis de tipos diferentes podem ter o mesmo nome. Assim, ***$img*** é uma variável diferente de ***@img*** ou ***%img***. Mas eu não recomendo que você utilize variáveis de nome parecido dentro de uma mesma função porque pode confundir um terceiro programador que venha a fazer manutenção no sistema.

3.2 Palavras reservadas

O Perl, como a maioria das linguagens de programação, tem palavras reservadas, que são os nomes de suas funções internas. Mas como as variáveis e funções criadas pelos usuários têm prefixos próprios **($, %,@,&),** elas nunca vão conflitar com as palavras internas do Perl. Entendeu por que Larry Wall decidiu usar esta forma de nomes no Perl? Imagine que você crie uma variável com nome ***img*** em uma linguagem qualquer e seu programa roda perfeitamente. Até que após uma atualização dos sistemas seu programa para de funcionar. E você descobre que isso ocorreu porque a nova versão da linguagem que você está usando adicionou ***img*** como palavra reservada. Você terá que refazer todos os seus programas que usam alguma função ou variável com o nome ***img*** para outro nome e rezar para nunca mais acontecer com outro nome que você escolheu. Com a metodologia de nomes utilizada no Perl isto nunca irá acontecer. Veja um pequeno exemplo, adicionado diretamente à linha de comando com ***perl*** -e:

```
$ perl -e '
>print "Ola\n";
>print &print,"\n";
>sub print { #crio uma função print interna
 >return 5 + 5;
>}'
Ola
10
$
```

Repare que, neste exemplo, eu crio uma função chamada **print**. Na maioria das linguagens essa nova função iria gerar um erro de conflito de nomes. Porém o prefixo **&** informa ao compilador que esta é uma função criada pelo usuário, permitindo que ela seja executada sem erros.

3.3 Strings

Diferentemente de outras linguagens, o Perl possui como tipo fundamental os escalares. As variáveis escalares podem conter apenas um valor. Este valor pode ser uma **string** ou um **número**. Num terceiro caso os escalares podem ser **undef,** o que significa "valor não atribuído". **Strings nada mais são que sequências de nenhum ou mais caracteres.** Ainda mais diferente das outras linguagens, as **strings** têm uma estrutura de muito alto nível. O programador não tem que se aborrecer definindo o provável tamanho da string, a quantidade de caracteres ou o quanto de memória devemos armazenar para esta. Basta associar a string a um escalar e podemos usá-la. Todas as conversões de subtipos, tamanho e alocação de memória ocorrem internamente, no interpretador do Perl.

Aprender a manusear as strings é importante para que possamos criar saídas formatadas para os nossos usuários, gerar relatórios ou ainda buscar por padrões em arquivos e na internet.

3.4 String literais

Strings literais são todas as strings que estão entre aspas no programa. Elas têm seu valor definido no código. Exemplo:

```
$meu_string = "Este é um string literal";
```

Para facilitar nosso trabalho com as strings, existem os caracteres de escape que representam os caracteres de controle, nova linha, tabulação etc. Os caracteres de escape são definidos por um caractere qualquer antecedido por uma barra invertida **(\ )**.

Código	**Significado:**
\\	Imprime o caractere \
\$	Imprime o caractere $ em vez de reconhecê-lo como identificador de variável
\@	Imprime o caractere @ em vez de reconhecê-lo como identificador de variável
\n	Nova linha (equivale a \f\r)
\r	Retorna o cursor para o início da linha
\f	Descer uma linha
\t	Adicionar uma tabulação, caractere de tabulação
\a	Campainha ou beep
\e	Tecla de escape (tecla esc)
\b	Backspace

3.5 Concatenando strings

Em muitos exemplos utilizamos variáveis dentro de uma string literal, e o Perl transformava automaticamente o valor da variável em sua string equivalente. Mas existem outros meios de concatenar strings. No Perl sempre existem vários meios de fazermos a mesma coisa.

Para isso você pode usar o operador "." – sim, é um ponto!

Exemplo:

```
  1 #!/usr/local/bin/perl
  2 #exemplo3.1
  3 $str1 = "Um string pode ser" . " " . "concatenado de
várias formas!\n";
```

```
  4 $str2 = "$str1 com o operador ponto: . ou com o uso de
uma variável" .
  5          " dentro da string!\n";
  6 print "A \$str2 = $str2";

saída:

A $str2 = Um string pode ser concatenado de várias formas!
 com o operador ponto: . ou com o uso de uma variável den-
tro da string!
```

3.5.1 As muitas cotas do Perl

O Perl reconhece uma string literal em meio ao código pelas aspas. Mas, na filosofia do Perl, existem muitas formas de fazer a mesma coisa. O mesmo acontece com as strings literais. Para facilitar a vida do programador (e dificultar a vida dos novatos), uma string literal pode ser declarada com **aspas, apóstrofo** ou com as funções ***q, qq, qr e qx***. Para que esta bagunça? Na verdade, para organizar!

As strings literais declaradas com apóstrofos não sofrem concatenações internas. Então, quando você precisar declarar uma string que contenha caracteres como $, @ ou % você pode fazer de uma das maneiras a seguir:

```
print "meu \@array e meu \%hash";
```

ou

```
print 'meu @array e meu %hash';
```

Ambos geram a **saída**:

```
meu @array e meu %hash
```

As strings literais usam a barra "\" como caractere de escape para o próprio apóstrofo ou ainda para o caractere barra em si.

```
print 'Então ele disse: \'Não! Não Pode ser!\ '. Mas já era
tarde! ';
print 'o caminho do arquivo é: c:\\meus documentos\\proje-
tos';
```

Os operadores ***q*** e ***qq*** podem nos oferecer um pouco mais de economia na digitação, definindo qual caractere queremos utilizar como delimitador da string literal. O operador ***qq*** fará o caractere escolhido funcionar como aspas e ***q*** o fará funcionar como apóstrofo.

Exemplo:

```
$perl -e "print q(Então ele disse: 'Não, Não pode ser'\n);
print q{O caminho do arquivo do arquivo é: c:\meus documen-
tos\projetos};"
Então ele disse: 'Não, Não pode ser'
O caminho do arquivo do arquivo é: c:\meus documentos\pro-
jetos
```

3.6 Digitando menos com a função say

A função ***'say'*** é uma melhoria adicionada ao Perl na versão 5.10. Ela funciona de forma idêntica à função ***'print',*** mas adiciona o caractere de fim de linha automaticamente ao final. Para utilizar a função ***say***, entretanto, precisamos dizer ao Perl que não estamos interessados na compatibilidade com as versões anteriores do compilador com a diretiva ***use 5.12.0***.

```
  1 #!/usr/local/bin/perl
  2 #exemplo3.2
  3 use 5.12.0;
  4
  5 say 'meu @array e meu %hash';
  6 say "meu \@array e meu \%hash";
  7 say q(Então ele disse: 'Não, Não pode ser');
  8 say q{O caminho do arquivo do arquivo é: c:\meus docu-
mentos\projetos};
```

```
saída:

meu @array e meu %hash
meu @array e meu %hash
Então ele disse: 'Não, Não pode ser'
O caminho do arquivo do arquivo é: c:\meus documentos\pro-
jetos
```

A função ***say*** não é nada revolucionária, mas alivia um bocado de digitação. Se você estiver usando a versão 5.8, quando vir a função ***say***, você deve usar a função ***print*** e terminar a string com ***"\n"***;

3.7 Textos grandes com os documentos HERE

O Perl foi, e ainda é, muito utilizado para programação de internet e de programas de linha de comando. Se você não for iniciante na área de TIC deve saber que o coração da internet, o http, realiza essencialmente o envio e recebimento de texto. E se você utiliza Linux ou algum unix-like com certeza trabalha muito com linhas de comando e deve ter visto como os documentos de ajuda e de apresentação destas aplicações têm grandes volumes de texto e até mesmo arte **ASCII**, que são desenhos formados a partir de caracteres de texto.

Para enfatizar a preguiça que permeia o ser do programador Perl, implementaram na linguagem os documentos **HERE**, que nada mais são que grandes volumes de texto embutidos no seu programa.

Exemplo:

```
1 #!/usr/local/bin/perl
2 #exemplo3.3
3 use 5.12.0;
4
5 say <<"Abertura";
```

```
  6      _______\\__
  7     (_. _ ._  _/   Bem vindo ao mundo da programação com Perl
  8      `-' \__. /  Aqui você irá descobrir que programação
  9           /  /  é bem mais que digitar códigos estranhos!!!
 10          /  /    .--.  .--.
 11         (  (    / `' \/ `' \   "
 12          \  \_.'            \   )
 13          ||                _  `./
 14           |\   \     ___.'\  /
 15             `-./   .'    \ |/
 16                \| /       )|\
 17                 |/       // \\
 18                 |\    __//   \\__
 19                //\\  /__/     \__|
 20            .--_/  \_--.
 21           /__/      \__\
 22
 23 Tecle enter para continuar!!!
 24 Abertura
 25 <STDIN> #STDIN força o usuário a digitar algo para con-
tinuar
```

saída:

```
    _______\__
   (_. _ ._  _/  Bem vindo ao mundo da programação com Perl
    `-' __. /    Aqui você irá descobrir que programação
         /  /    é bem mais que digitar códigos estranhos!!!
        /  /    .--.  .--.
       (  (    / `' / `'    "
         _.'                )
        ||                _  `./
         |         ___.'  /
           `-./   .'      |/
              | /        )|
               |/        // \
               |    __//   \__
              //\  /__/     __|
          .--_/  _--.
         /__/      __

Tecle enter para continuar!!!
```

A partir deste exemplo é possível imaginar o quanto de digitação seria necessário para acertar a posição dos caracteres na imagem apenas com as funções **say** e **print.**

Mas como exatamente funcionam os documentos **HERE**? Eles são identificados pelo operador **<<** seguido da string de delimitação, no caso ABERTURA. Então o meu documento HERE inicia na linha logo abaixo de ***say <<"Abertura"*** e finaliza no último caractere antes da string **Abertura** ao final do documento.

Se a string de demarcação do documento **HERE** estiver entre apóstrofos, então não ocorrerá concatenação de string por meio de variáveis adicionadas ao texto do documento. Se estiver entre aspas poderemos utilizar a concatenação de strings normalmente, ou seja, irá substituir as variáveis por seus valores na saída.

Veja mais exemplos:

```
  1 #!/usr/local/bin/perl
  2 #exemplo3.4
  3 use 5.12.0;
  4
  5 my $var = "Ei, eu concatenei!!!\n";
  6 say <<"CONCATENA";
  7 Então, as strings se reuniram
  8         para decidir como fariam para se reunir às va-
riáveis
  9                até que uma disse:
 10
 11                $var
 12 CONCATENA
 13
 14 say <<'CONTINUA';
 15 Porém aqui a variável é apenas $var.
 16 CONTINUA

saída:

Então, as strings se reuniram
para decidir como fariam para se reunir às variáveis
     até que uma disse:


           Ei, eu concatenei!!!


Porém aqui a variável é apenas $var.
```

Os documentos **HERE** também são úteis para programação de scripts **CGI** à moda antiga:

```
 1 #!/usr/local/bin/perl
 2 #exemplo3.5
 3
 4 use warnings;
 5
 6 $data = localtime;
 7 print <<"INDEX";
 8 Content-Type: text/html; charset=ISO-8859-1
 9
10 <!DOCTYPE html
11         PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
12          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transi-
tional.dtd">
13 <html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
14 <head>
15 <title>Untitled Document</title>
16 <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
17 </head>
18 <body>
19      <h3>Bem vindo ao nosso site</h3>
20      <p>a data atual Ú: $data</p>
21
22 </body>
23 </html>
24
25 INDEX
26 exit;
```

saída:

```
Content-Type: text/html; charset=ISO-8859-1
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transi-
tional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
     <h3>Bem vindo ao nosso site</h3>
     <p>a data atual é: Sat May 29 02:03:59 2010</p>
</body>
</html>
```

Perceba que a saída deste script gera uma página **HTML** perfeita, onde a variável $data apresenta a hora exata do acesso à página. Isso permite a programação de scripts **CGI** que geram páginas com conteúdo dinâmico. Mais adiante iremos aprender a gerar páginas **CGI**. Até lá você pode ler as saídas **HTML** na linha de comando – ler o html na saída é uma forma de debug básico de programas **CGI**. Se souber instalar e configurar um servidor web, adicione o shebang correto em relação ao seu sistema na primeira linha do script e coloque o script na pasta de scripts **CGI** do seu servidor.

Você pode ter quantos documentos ***HERE*** precisar no seu programa, mas há um detalhe importante: a string de terminação do documento ***HERE*** não pode ser a última linha do arquivo; para isso adicione uma linha em branco ou termine o arquivo de forma elegante com a função ***exit***.

3.8 O operador x

Larry Wall defendia a preguiça como uma das características mais marcantes do programador Perl. Por quê? Porque a preguiça leva à otimização, quando bem direcionada. Outro operador que nos traz essa característica na linguagem é o operador ***'x',*** que pode repetir inúmeras vezes um string.

```
 1 #!/usr/local/bin/perl
 2 #exemplo3.6
 3 use warnings;
 4 use feature "say";
 5
 6 say "#" x 40;
 7 say "#" x 15, "Bem Vindo", "#" x 16;

saída:

########################################
###############Bem Vindo################
```

Neste exemplo economizamos a digitação de dezenas de caracteres.

3.9 As funções printf e sprintf

Usando o princípio da simplicidade: **não use garfo e faca se a colher resolver o problema.** Mas às vezes precisamos usar garfos, facas e talheres de sobremesa. As funções print e say produzem saídas interessantes para os usuários, podemos manusear a saída por meio dos caracteres de escape como \a, \n, \t e podemos ainda complementar as coisas com o operador ***x,*** mas muitas vezes precisamos produzir saídas mais sofisticadas que demandariam muito trabalho com as funções print, say ou mesmo os documentos HERE. Para essas saídas mais incrementadas foram implementadas no Perl as funções ***printf*** e ***sprintf.*** Elas introduzem, além dos já tradicionais caracteres de escape, os controles de conversão que permitem formatar a string de saída, por exemplo, definindo quantos dígitos um número deve ser mostrado ou mesmo alinhar os caracteres na tela.

A lista adiante apresenta as sequências de controle:

%%	Imprime o caractere %
%c	Converte o código ASCII em caractere
%s	Imprime uma string relacionada
%d	Imprime um número decimal com sinal
%u	Imprime um número decimal sem sinal
%o	Imprime um número em octal sem sinal
%x	Imprime um inteiro sem sinal em Hexadecimal

%e Imprime um número real em notação científica
%f Imprime um número real
%g Imprime um número real. A notação pode ser em ponto flutuante ou em notação científica
%X mesmo que %x mas com letras maiúsculas
%E mesmo que %e mas com letras maiúsculas
%G mesmo que %g mas com letras maiúsculas
%b Imprime um número binário sem sinal
%B O mesmo que %b mas usa a flag de binário em maiúsculas
%p Imprime o endereço de memória de uma referência

A função ***printf*** tem o seguinte formato:

```
printf <string de controle de impressão>, <@array_de_valo-
res>;
```

Veja um exemplo:

```
 1  #!/usr/local/bin/perl
 2  #exemplo3.7
 3  use warnings;
 4
 5 #Remove a parte fracionária do número
 6  printf "%d\n", 5010.956;
 7
 8 #Mantém a parte fracionária do número
 9  printf "%f\n", 5010.956;
10
11 #Remove a parte fracionária do número
12  printf "%d\n", +5010.956;
13
14 #Remove a parte fracionária do número e mantém o sinal
15  printf "%d\n", -5010.956;
16
17 #Mantém a parte fracionária do número e mantém o sinal
18  printf "%f\n", -5010.956;
19
```

```
 20 #Mantém a parte fracionária do número e mantém o sinal.
Notação científica
 21  printf “%e\n”, 5010.956;
 22
 23 #Mantém a parte fracionária do número e mantém o sinal
 24  printf “%e\n”, -5010.956;
 25
 26 #Mantém a parte fracionária do número e mantém o sinal
 27  printf “%E\n”, 5010.956;
 28
 29 #Mantém a parte fracionária do número e mantém o sinal

 30  printf “%E\n”, -5010.956;
 31
 32 #Mantém a parte fracionária do número e mantém o sinal
 33  printf “%g\n”, 5010.956;
 34
 35 #Mantém a parte fracionária do número e mantém o sinal
 36  printf “%g\n”, -5010.956;
 37
 38 #Mantém a parte fracionária do número e mantém o sinal
 39  printf “%G\n”, 5010.956;
 40
 41 #Mantém a parte fracionária do número e mantém o sinal
 42  printf “%G\n”, -5010.956;
 43
 44 #Remove a parte fracionária do número
 45  printf “%u\n”, +5010.956;
 46
 47 #Remove a parte fracionária do número e converte para
um positivo
 48  printf “%u\n”, -5010.956;
 49
 50 #Remove a parte fracionária do número e exibe como he-
xadecimal
 51  printf “%x\n”, 5010.956;
 52
 53 #Remove a parte fracionária do número e exibe como he-
xadecimal positivo
```

```
 54  printf "%x\n", -5010.956;
 55
 56 #Remove a parte fracionária do número
 57  printf "%X\n", 5010.956;
 58
 59 #Remove a parte fracionária do número e exibe como he-
xadecimal positivo
 60  printf "%X\n", -5010.956;
 61
 62 #Remove a parte fracionária do número e exibe como oc-
tal
 63  printf "%o\n", 5010.956;
 64
 65 #Remove a parte fracionária do número e exibe como oc-
tal positivo
 66  printf "%o\n", -5010.956;
 67
 68 #Remove a parte fracionária do número e exibe como oc-
tal positivo
 69  printf "%O\n", 5010.956;
 70
 71 #Remove a parte fracionária do número e exibe como oc-
tal positivo equivalente
 72  printf "%O\n", -5010.956;
 73
 74 #Remove a parte fracionária do número e exibe como bi-
nário
 75  printf "%b\n", 5010.956;
 76
 77 #Remove a parte fracionária do número e exibe como bi-
nário positivo
 78  printf "%b\n", -5010.956;
 79
 80 #Remove a parte fracionária do número e exibe como bi-
nário
 81  printf "%B\n", 5010.956;
 82
```

```
 83 #Remove a parte fracionária do número e exibe como bi-
nário positivo
 84 printf "%B\n", -5010.956;

saída:

5010
5010.956000
5010
-5010
-5010.956000
5.010956e+003
-5.010956e+003
5.010956E+003
-5.010956E+003
5010.96
-5010.96
5010.96
-5010.96
5010
4294962286
1392
ffffec6e
1392
FFFFEC6E
11622
37777766156
11622
37777766156
1001110010010
11111111111111111110110001101110
1001110010010
11111111111111111110110001101110
```

Você reparou numa coisa: **%b** e **%B**, assim como **%x** e **%X**, mostraram a mesma saída. Então qual a vantagem de termos operadores diversos fazendo a mesma coisa? Vamos perceber a vantagem com a bandeira **'#'**. Basta adicionar **'#'** entre a porcentagem e os caracteres de controle b, B, o, O, x e X e vamos ter um indicador de base numérica na string de saída:

```
 1 #!/usr/local/bin/perl
 2 #exemplo3.8
 3
 4 printf "479 em hexa é: %#x\n", 479;
 5 printf "479 em hexa é: %#X\n", 479;
 6 printf "479 em octal é: %#o\n", 479;
 7 printf "479 em octal é: %#O\n", 479;
 8 printf "479 em binário é: %#b\n", 479;
 9 printf "479 em binário é: %#B\n", 479;

saída:

479 em hexa é: 0x1df
479 em hexa é: 0X1DF
479 em octal é: 0737
479 em octal é: 0737
479 em binário é: 0b111011111
479 em binário é: 0B111011111
```

Agora um exemplo com %s e %c:

```
 1 #!/usr/local/bin/perl
 2 #exemplo3.9
 3
 4 use warnings;
 5 use feature "say";
 6
 7 $elemento1 =
 8 "Os caracteres têm valores ASCII numéricos que são con-
vertidos para " .
 9 "letras.\n";
10 $elemento2 =
11 "Se quiser ver os valores numéricos de todas as letras
execute o loop" .
12 "abaixo\n";
13 printf qq/O número 65 têm o valor %c enquanto que número
66 / .
```

```
14        qq/têm o valor %c e o 67 %c\n/, 65,66, 67;
15
16 printf "%s%s", $elemento1, $elemento2;
17 $ascii = 33;
18 while($ascii <= 122){
19   printf "O número $ascii vale: %c\n", $ascii;
20   $ascii = $ascii + 1;
21 }
```

saída:

O número 65 têm o valor A enquanto que número 66 têm o valor B e o 67 C
Os caracteres têm valores ASCII numéricos que são convertidos para letras.
Se quiser ver os valores numéricos de todas as letras execute o loop abaixo
O número 33 vale: !
O número 34 vale: "
O número 35 vale: #
O número 36 vale: $
O número 37 vale: %
O número 38 vale: &
O número 39 vale: '
O número 40 vale: (
*o restante da saída foi omitido por questão de espaço e por ser desnecessária.

Bem, nós já percebemos que podemos utilizar strings longas, formatar números em decimais, reais, hexadecimais e converter valores numéricos em caracteres. Mas a função ***printf*** vai além disto.

Agora vamos ver como usar a função printf para definir os caracteres de alinhamento ou para definir quantos caracteres de tamanho, no mínimo, deve ter um elemento na saída controlada. Basta adicionar o número de strings que deseja utilizar entre o símbolo de porcentagem e o caractere de controle em si, como **%5s** ou **%9d**.

Exemplo:

```
1 #!/usr/local/bin/perl
2 #exemplo3.10
3
4 use warnings;
5 use feature "say";
6
7 $str1 = "Caso eu queira alinhar para a direita:\n";
8 @array = ($str1, 500, 1000, 10000, 100000);
9 printf "%7s%7d\n%7d\n%7d\n%7d\n", @array;

saída:

Caso eu queira alinhar para a direita:
    500
   1000
  10000
 100000
```

Perceba que todos os elementos estão alinhados sete caracteres para a direita. Menos a primeira string, pois ela supera os sete caracteres de tamanho, então a função completa os caracteres para saída, transbordando a formatação. Mas perceba que os sete primeiros caracteres estão alinhados. Se utilizar 38 como tamanho mínimo, em vez do sete, aí teremos a saída toda alinhada para a direita:

Claro que você pode aumentar sua flexibilidade usando uma variável para determinar o alinhamento dos elementos:

```
1 #!/usr/local/bin/perl
2 #exemplo3.11
3
4 use warnings;
5 use feature "say";
6
7 $al = 38;
8 $str1 = "Caso eu queira alinhar para a direita:\n";
9 @array = ($str1, 500, 1000, 10000, 100000);
```

```
10 printf "%${al}s%${al}d\n%${al}d\n%${al}d\n%${al}d\n", @
array;
saída:

Caso eu queira alinhar para a direita:
                                   500
                                  1000
                                 10000
                                100000
```

Mas... e se eu quiser determinar o tamanho da parte de ponto flutuante de um número da saída? Para dizer que queremos uma saída com cinco caracteres na parte flutuante do número usamos %.5f (repare no ponto antes do número cinco).

Exemplo:

```
 1 #!/usr/local/bin/perl
 2 #exemplo3.12
 3 use warnings;
 4
 5 $str1 =
 6 "Notação monetária costuma usar precisão de dois dígitos
decimais:\n";
 7 $al = length($str1) - 2;
 8 #A função length retorna o tamanho de um string em núme-
ro de caracteres
 9 @valores = ($str1, 45.00, 50.98, 197.0018, 550.9901);
10 printf "%${al}s %${al}.2f\n %${al}.2f\n %${al}.2f\n
%${al}.2f\n", @valores;

saída:

Notação monetária costuma usar precisão de dois dígitos de-
cimais:
                                                      45.00
                                                      50.98
                                                     197.00
                                                     550.99
```

E para finalizar a função ***printf*** vamos ver os últimos sinais de controle:

-	justifica a saída à esquerda.
+	mostra o sinal de um elemento numérico como em %+f ou %+d.
<espaço>	adiciona um espaço à string de saída.
#	mostra o indicador da base do número.
0	Preenche campos vazios à esquerda com zeros em vez de espaços.

```
  1 #!/usr/local/bin/perl
  2 #exemplo3.13
  3
  4 use warnings;
  5
  6 $al = 5;
  7 printf "veja a saída preenchida com zeros: %0${al}d\n",
100;
  8 printf "veja a saída preenchida com espaço adicional: %
${al}d\n", 100;
  9 printf "veja a saída preenchida com espaço adicional e
zeros: %0${al}d\n",
 10        100;
 11 printf "Agora os sinais:\n %+f\n %+f\n %+d\n", 100.56,
-10.0001, -10000;

saída:

veja a saída preenchida com zeros: 00100
veja a saída preenchida com espaço adicional:   100
veja a saída preenchida com espaço adicional e zeros:  0100
Agora os sinais:
 +100.560000
 -10.000100
 -10000
```

Bem, acho que vimos o bastante de printf, print, say, documentos HERE e todas as formas como você pode manipular a saída com estas funções. Por outro lado, ainda não falei do ***sprintf***. Esta função retorna uma string formatada, como a uti-

lizada pela função ***printf***, de forma que eu possa usar essa string já formatada a qualquer hora em meu programa:

```
  1 #!/usr/local/bin/perl
  2 #exemplo3.14
  3
  4 use warnings;
  5 use feature "say";
  6
  7 $distancia = 384405000;
  8 $str_formatada = sprintf
  9 "A distância entre a Terra e a Lua é cerca de %E me-
tros", $distancia;
 10 say $str_formatada;

saída:

A distância entre a Terra e a Lua é cerca de 3.844050E+008
metros
```

3.10 Controlando a caixa de uma string

Às vezes podemos querer uniformidade na caixa de strings para fins de normatização e padronização de saída ou armazenamento. Normalmente estas normatizações de caixa são: todas as letras minúsculas, todas as letras maiúsculas ou todas as primeiras maiúsculas ou ainda todas as primeiras minúsculas. Para isto temos as funções irmãs: ***lc, uc, ucfirst*** e ***lcfirst***.

Todas têm a implementação semelhante. ***uc*** retorna uma string idêntica à fornecida com todos os caracteres em maiúsculas (upper case), ***lc*** retorna uma string idêntica à fornecida com todos os caracteres em minúsculas (lower case). ***ucfirst*** e ***lcfirst*** modificam apenas o primeiro caractere.

Exemplo:

```
  1 #!/usr/local/bin/perl
  2 #exemplo3.15
  3
  4 use warnings;
```

```
 5 use feature "say";
 6
 7 $str1 = "TODAS AS MINHAS LETRAS SÃO MAIÚSCULAS";
 8 $str2 = "todas as minhas letras são minúsculas";
 9 $convertida1 = lc $str1;
10 $convertida2 = lcfirst $str1;
11 $convertida3 = ucfirst $str1;
12 $convertida4 = uc $str2;
13 $convertida5 = ucfirst $str2;
14 $convertida6 = lcfirst $str2;
15 @convertidas = ($convertida1,$convertida2,$convertida3,
16                $convertida4,$convertida5,$convertida6);
17
18 $i = 0;
19 $str_controle = "";
20 while($i<=5){
21         $str_controle = $str_controle . "%s\n";
22         $i = $i + 1;
23 }
24 printf $str_controle, @convertidas;
```

saída:

```
todas as minhas letras sÃo maiÚsculas
tODAS AS MINHAS LETRAS SÃO MAIÚSCULAS
TODAS AS MINHAS LETRAS SÃO MAIÚSCULAS
TODAS AS MINHAS LETRAS SãO MINúSCULAS
Todas as minhas letras são minúsculas
todas as minhas letras são minúsculas
```

Perceba que ***lcfirst*** não tem efeito se todas as letras forem minúsculas. Ou seja, ele apenas garante que a primeira letra seja minúscula, as outras mantêm a caixa da string original. O mesmo vale para ***ucfirst***. Observe ainda que neste aplicativo utilizamos um loop, que gerou dinamicamente a string de controle da função ***printf,*** aumentando mais ainda a flexibilidade da função.

3.11 Transformando uma string em um array de caracteres

Algumas linguagens como o C ou o C++ não possuem **strings.** Temos arrays de caracteres. Como as strings são tipos básicos do Perl, você pode querer rodar um algoritmo de outra linguagem ou mesmo fazer uma pesquisa detalhada em todos os caracteres de uma string. Para isso você precisa transformar sua string em um array de caracteres. Veja no código adiante um exemplo que transforma uma string informada pelo usuário em uma sequência de numeração ASCII equivalente para cada caractere.

Exemplo:

```
  1 use feature 'say';
  2 #exemplo3.16
  3 #transformar uma string num array de caracteres:
  4 chomp ($str = <STDIN>);
  5 @array_char = split //, $str;
  6 map { say ord $_; } @array_char;

saída:

testando
116
101
115
116
97
110
100
111
```

Analisando o código, você vai se deparar com três funções desconhecidas ainda: ***map, ord*** **e** ***split***. A função **map** recebe como argumento cada elemento de um array informado, passando este elemento para a variável interna do Perl ***$_*****.** Então o código informado entre chaves **{}** é executado para cada elemento do array. Vamos ver mais detalhes sobre essa e outras funções no capítulo sobre arrays e hashes.

A função ***ord*** converte cada caractere informado para seu decimal equivalente do código ASCII. A função ***chr*** executa uma tarefa oposta a ***ord***, transformando um número dado em seu caractere ASCII equivalente.

```
  1 #exemplo3.17
  2 @num = (30..126);
  3 map {print chr $_, " "} @num;

saída:

- ! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < =
> ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [
\ ] ^ _ ` a b c d e f g h i j k l m n o p q r s t u v w x y
z { | } ~
```

A função ***split*** divide uma string em partes, conforme um padrão informado entre //.

```
  1 #exemplo3.18
  2 use feature "say";
  3 @array = split /;/, "este;é;um;exemplo;de;csv;";
  4 say $array[3];
  5 say $array[1],"\n";
  6 map {say $_} @array;

saída:

exemplo
é
este
é
um
exemplo
de
csv
```

Os arquivos "csv", onde os dados são separados por "ponto-e-vírgula", são muito comuns na informática. A função ***split*** também é útil nesses casos, pois pode retornar os dados úteis e descartar os divisores.

No primeiro exemplo o padrão de separação de *split* estava em branco: ***split //, $str*** logo a função separa todos os caracteres.

3.12 Expressões regulares

Uma das mais famosas características do Perl é o seu mecanismo de ***expressões regulares***, um dos mais poderosos dentre as linguagens de programação. Tanto que diversas linguagens preferem embutir o mecanismo do Perl a criar um próprio. Ou ainda, alguns programadores C e Java **embarcam código Perl** no sistema para economizar linhas e tempo.

Mas o que vem a ser uma expressão regular, ou ***regexp***, ou ainda ***regex***? Uma expressão regular é um padrão. Então o Perl irá verificar se uma string atende ou não a um padrão. Quando ele está dentro do padrão dizem, no jargão, que realizou o ***match*** ou que **a string combina/atende** o padrão. Vamos supor a string "ao". Se eu quiser verificar se uma determinada string atende o padrão basta executar ***$string =~ /ao/;*** Se houver na string a sequência **ao**, em qualquer parte da string, ela irá atender o padrão e a operação irá retornar true.

O operador que verifica um padrão é ***=~ (pattern-binding)***

Veja:

```
 1 #exemplo3.19
 2  $str1 = "pao";
 3  $str2 = "macarrao";
 4  $str3 = "mamaozinho";
 5  $str4 = "Cabana";
 6  $str5 = "Um paozinho por favor!";
 7  $match = "ao";
 8
 9 if ($str1 =~ $match){
10   say $str1
11 }
12 if ($str2 =~ $match){
13   say $str2
14 }
15 if ($str3 =~ $match){
16   say $str3
17 }
18 if ($str4 =~ $match){
19   say $str4 #Cabana não atende ao padrão
20 }
```

```
 21 if ($str5 =~ $match){
 22    say $str5 #O padrão pode estar em qualquer parte da
string
 23 }


saída:

pao
macarrao
mamaozinho
Um paozinho por favor!
```

Agora vamos fazer um exemplo mais prático. Supondo que eu esteja trabalhando num sistema para configurar um servidor Apache, e uma das funcionalidades será verificar determinados itens de configuração. Vamos supor o programa:

```
  1 #exemplo3.20
  2 use feature "say";
  3 #O path dos arquivos do Apache varia conforme a insta-
lação e versão
  4 use constant HTTPD => 'C:\Program Files\Apache Software
' .
  5 'Foundation\Apache2.2\conf\httpd.conf';
  6 #Módulo para abrir e ler arquivos e substitui as fun-
ções open e readline.
  7 use FileHandle;
  8
  9 my $conf_file = FileHandle->new( HTTPD, "r" ); #funcio-
na como a função open
 10 say "erro" if not defined $conf_file; #gera um erro se
não abrir o arquivo
 11 my $match = "ScriptAlias"; #string para o match
 12
 13 while ($_ = $conf_file->getline()){
 14 if (/$match/){
 15 print;
```

```
 16 }
 17 }
 18
 19 $conf_file->close();

saída:

# ScriptAlias: This controls which directories contain
server scripts.
# ScriptAliases are essentially the same as Aliases, except
that
# client. The same rules about trailing "/" apply to Script-
tAlias

ScriptAlias /cgi-bin/ "C:/Program Files/Apache Software
Foundation/Apache2.2/cgi-bin/"

# "C:/Program Files/Apache Software Foundation/Apache2.2/
cgi-bin" should be changed to whatever your ScriptAliased
# To use CGI scripts outside of ScriptAliased directories:
```

Essa saída ainda tem muito o que melhorar para mostrar apenas a informação que queremos, mas já tem o ponto essencial. A nossa saída tem toda e qualquer string que contenha a sequência ScriptAlias. E nosso programa merece uma explicação extra sobre $_:

Nota Importante: A variável padrão $_.

Você já percebeu que os programadores Perl abominam digitar (acho que herdaram isso do comportamento americano, do not... don't, I will... I'll. Aliás, eu já falei que Larry Wall era formado em Letras?). Então programas escritos por programadores experientes costumam ser abreviados ao máximo! E a variável $_ está aí para isso. Ou melhor, não está aí... quero dizer... Está aí, mas não está visível!

O perldoc define esta variável como a variável padrão para pesquisa de padrões e entrada de dados.

Ela pode ser usada em verificação de padrões da seguinte forma:

$_ =~ $padrao;

seria o mesmo que:

/$padrao/

Ou

say $_;

seria o mesmo que:

say;

Isso é que é não gostar de digitar!!!
E vai além. Qualquer função dentre as listadas a seguir terá $_ como argumento se não for especificado de outra forma:

abs, alarm, chomp, chop, chr, chroot, cos, defined, eval, exp, glob, hex, int, lc, lcfirst, length, log, lstat, mkdir, oct, ord, pos, print, quotemeta, readlink, readpipe, ref, require, reverse (apenas em contexto escalar), rmdir, sin, split (on its second argument), sqrt, stat, study, uc, ucfirst, unlink, unpack.

Mais um detalhe é que $_ é uma variável global. Então mudanças na variável podem afetar de forma inesperada seu programa. Porém eu posso especificar instâncias locais da minha variável com o comando my $_ .

Depois desta explicação extra, fica claro que ***if (/$match/)*** é o mesmo que ***if ($_ =~ $match)*** e ***while ($_ = $conf_file->getline())*** é o mesmo que ***while (defined ($_ = $conf_file->getline()))***, com muito menos esforço de digitação.

De qualquer forma, nossa pesquisa ainda não ficou boa o bastante. Queremos apenas a linha que começa com ScriptAlias. Para tanto devemos adicionar à nossa pesquisa o caractere **^ (acento circunflexo)**. Esse caractere indica ao mecanismo de pesquisa que o padrão buscado deve estar no início da string. Mude a linha 10 do exemplo3.19 para ***$match = '^ScriptAlias'*** . Porém, no meu arquivo do Apache a string ***ScriptAlias*** não é o começo de nenhuma linha. Elas estão tabuladas. Então ainda vou precisar adicionar o ***\s***, que indica que esperamos um espaço, seguido de **+.** O sinal + indica que o caractere anterior pode aparecer uma ou mais vezes. Então a linha ficará ***$match = '^\s+ScriptAlias',*** que significa: No início da string, um ou mais espaços seguidos por ScriptAlias.

Veja a nova saída:

```
ScriptAlias /cgi-bin/ "C:/Program Files/Apache Software
Foundation/Apache2.2/cgi-bin/"
```

Ou podemos ainda solicitar todas as linhas não comentadas. Se para indicar um "combina" usamos =~ para um "não combina" usamos !~.

Veja:

```
  1 #exemplo3.21
  2 use feature "say";
  3 use constant HTTPD =>
  4 'C:\Program Files\Apache Software Foundation\Apache2.2\
conf\httpd.conf';
  5 use FileHandle; #Módulo para manusear filehandles
  6
  7 $conf_file = FileHandle->new( HTTPD, "r" );
  8 say "erro" if not defined $conf_file;
  9 $match = '#';
 10
 11 while ($_ = $conf_file->getline()){
 12         if ($_ !~ /$match/){ # Se não fizer match
 13                 say;
 14         }
 15 }
```

```
 16 $conf_file->close();

saída:

ServerRoot "C:/Program Files/Apache Software Foundation/
Apache2.2"
Listen 8080
LoadModule actions_module modules/mod_actions.so
LoadModule alias_module modules/mod_alias.so
LoadModule asis_module modules/mod_asis.so
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule authn_default_module modules/mod_authn_default.
so
...continua...
```

Vamos continuar com os principais pontos do regex. Vimos que podemos procurar um determinado padrão no início de uma string com o caractere ^ no início do padrão. E se quisermos procurar no final da string? Podemos adicionar ao final da regex o caractere $. Voltando ao primeiro exemplo, vamos mudar a linha **my $match = "ao"** para **my $match = 'ao$'** com apóstrofo (cota simples) ou o Perl vai tentar concatenar a string. E veja a nova saída:

```
pao
macarrao
```

As strings paozinho e mamaozinho foram descartadas da saída pois, embora possuam a sequência 'ao', elas não estão no fim da string.

O Perl provê ainda três variáveis internas que informam qual a string que efetuou o match com o seu padrão, o que havia antes e o que havia depois. Essas variáveis são ***$&, $` e $'.***

Exemplo:

```
 1 #exemplo3.22
 2 #use feature ("say");
 3
```

```
  4 $str1 = "pao";
  5 $str2 = "macarrao";
  6 $str3 = "mamaozinho";
  7 $str4 = "Cabana";
  8 $str5 = "Um paozinho por favor!";
  9 $match = 'ao';
 10 if ($str5 =~ $match){
 11         say $str5; #O padrão pode estar em qualquer
parte da string
 12         printf "Match: %s\nAntes: %s\nDepois: %s\n",
$&, $`,$';
 13 }

saída:

Um paozinho por favor!
Match: ao
Antes: Um p
Depois: zinho por favor!
```

A variável $& é interessante, pois o Perl permite pesquisas não apenas por uma sequência de caracteres definidos, mas também por uma classe de caracteres. Com esta variável eu posso saber o valor exato que foi encontrado pelo programa. As classes principais são:

Espaços	\t \n \r \f	\s
Caracteres do alfabeto e dígitos	a-z A-Z _ 0-9	\w
Dígitos	0 – 9	\d
Tudo exceto caracteres do alfabeto e dígitos	Exceto: a-z A-Z _ 0-9	\W
Tudo exceto espaços	Exceto: \t \n \r \f	\S
Tudo exceto dígitos	Exceto: 0 – 9	\D

E também os metacaracteres:

+	Uma sequência com uma ou mais vezes o caractere anterior.
.	Qualquer caractere, exceto \n.
*	Uma sequência com nenhuma ou mais vezes o caractere anterior.
{n,m}	n=no mínimo "n" no máximo "m".
?	Uma sequência com uma ou nenhuma vez o caractere anterior.
^	A sequência a seguir deve ser a primeira da string.
$	A sequência anterior deve ser a última da string.

Como exemplo, vamos montar uma máscara para realizar um match em um endereço IP. Um endereço IP é uma sequência de dígitos seguidos por pontos. Porém, o ponto é um metacaractere dentro da regex. Logo, precisamos de um caractere \ (barra invertida) antes do ponto para indicar que ele não será usado como metacaractere. Isso vale para todos os metacaracteres. A barra invertida os torna caracteres comuns.

Então nossa primeira máscara será **\d+\.** , ou seja, uma sequência de um ou mais dígitos seguidos por ponto. Veja:

```
 1 use feature "say";
 2 #exemplo3.23
 3
 4 $ip1 = "192.168.4.10";
 5 $ip2 = "10.10.4.5";
 6 $ip_errado = "1010.5.3"; #erro de digitação;
 7 $var = "Casa.Carro";
 8 if ($ip1 =~ /\d+\./){
 9         say $&;
10 }
11 if ($ip2 =~ /\d+\./){
12         say $&;
13 }
14 if ($ip_errado =~ /\d+\./){
15         say $&; #Faz o Match, mas não é IP.
16 }
17 if ($var =~ /\d+\./){
```

```
 18         say $&; #nunca executa
 19 }

saída:

192.
10.
1010.
```

Nossa máscara nos protege de erros grotescos, impedindo que algo sem dígitos seja aceito. Porém, ainda passam erros. Na anterior vimos o quantificador ***{n,m}***. Os endereços IP podem ter uma sequência de um a três dígitos em cada classe do endereço.

Então podemos melhorar nossa máscara para:

```
\d{1,3}\.
```

Mas ainda haverá o erro: 1010. tem uma sequência de três caracteres seguidos por ponto, que é 010. Mude a **regex** e execute o programa para ver. Então vamos adicionar um recurso – agrupar os caracteres:

```
(\d{1,3}\.)+
```

Os novos **matches** serão:

```
192.168.4.
10.10.4.
010.5.
```

Ainda não é exatamente o que queremos. Em um endereço IP, três sequências de dígitos são seguidos por ponto. Então vamos exigir três repetições do nosso agrupamento:

```
(\d{1,3}\.){3}
```

E teremos na saída:

```
192.168.4.
10.10.4.
```

Mas ainda temos limitações: a última sequência de caracteres não está protegida. Para protegê-la vamos adicionar \d{1,3} ao final da máscara. Assim:

```
(\d{1,3}\.){3}\d{1,3}
```

E a saída será:

```
192.168.4.10
10.10.4.5
```

Agora vamos usar essa regex em um arquivo de logs do Apache para ver quais IPs tentaram se conectar ao meu servidor:

```
  1 #exemplo3.24
  2 use FileHandle; #Módulo para abrir arquivos de texto
  3 use constant {FILE => "logs.txt",
  4                   MASK => '(\d{1,3}\.){3}\d{1,3}'
  5                   };
  6 use feature ("say");
  7
  8 my $log = FileHandle->new(FILE, 'r'); #Abre o arquivo
para leitura
  9
 10 say "Os IPs abaixo acessaram meu servidor:";
 11 while($_ = $log->getline()){ #mesmo que while($_ =
<$log>){
 12         if ($_ =~ MASK){ #verifica se $_ efetua o match
com a máscara MASK
 13                 say $&
 14         }
 15 }
```

O conteúdo do meu arquivo é:

```
[Thu Jun 24 09:32:51 2010] [error] [client 189.85.128.10]
File does not exist: /home/thiago/public_html/favicon.ico
[Thu Jun 24 09:32:51 2010] [error] [client 189.85.128.10]
File does not exist: /home/thiago/public_html/favicon.ico
[Thu Jun 24 09:32:51 2010] [error] [client 189.85.128.10]
File does not exist: /home/thiago/public_html/favicon.ico
[Thu Jun 24 08:52:28 2010] [error] [client 189.77.212.142]
File does not exist: /home/thiago/public_html/favicon.ico
[Thu Jun 24 08:52:25 2010] [error] [client 189.77.212.142]
File does not exist: /home/thiago/public_html/favicon.ico
[Thu Jun 24 08:23:23 2010] [error] [client 189.15.121.202]
File does not exist: /home/thiago/public_html/favicon.ico
```

Veja que a máscara que utilizamos isola muito bem o endereço IP para esse caso em questão. Mas ela pode ser melhorada para outras situações.

A saída gerada pelo script é:

```
Os IPs abaixo acessaram meu servidor:
189.85.128.10
189.85.128.10
189.85.128.10
189.77.212.142
189.77.212.142
189.15.121.202
```

Exercícios

1. Monte um script que transforme todas as letras de um arquivo em maiúsculas.

2. Desenvolva um script que conte quantos espaços existem em um arquivo.

3. Faça um pequeno programa que solicite ao usuário sua data de nascimento. Monte uma expressão regular para verificar se a data informada é válida no formato dd/mm/aa ou dd-mm-aa.

4. Monte uma abertura de programa de linha de comando com um "HERE Document" apresentando a sua empresa ou uma empresa fictícia.

Capítulo 4. Variáveis Simples: Números e Operações Matemáticas

Os números, assim como as strings, se enquadram no tipo mais básico do Perl, os escalares. Números estão entre os elementos mais importantes das linguagens de programação, pois internamente ao seu computador todos os dados são representados como números binários. Além desta característica física dos processadores e memórias, trabalhar com números é algo extremamente importante em aplicações científicas, de engenharia, financeiras e matemáticas.

Como vimos anteriormente, as strings também compõem os tipos escalares. E damos ao interpretador Perl a responsabilidade de lidar devidamente com os números e strings e usá-los da forma devida. Assim, quando seu usuário enviar a string '5', o Perl irá converter o valor ASCII em número na operação 5 + 5 e vai convertê-lo novamente em seu valor ASCII para exibir a saída na tela. Podemos comparar esta inteligência com a nossa linguagem, que automaticamente percebemos pelo contexto do assunto quando o símbolo 5 deve ser usado para uma operação matemática, como 5x3 ou 30/5, ou quando queremos usar o símbolo de forma a enfatizar um outro significado. Exemplo: paragrafo 5; a distância de 5 metros; 5 + 55 = 60.

Para mais detalhes você pode ler as páginas de manual perlvar e perlnumber, com os comandos:

```
$ perldoc perlvar e $ perldoc perlnumber
```

4.1 Formas de representar um número no Perl

Assim como as strings, os números podem ser introduzidos no programa por meio de filehandles ou podem ser atribuídos diretamente no código, como as strings literais.

4.2 Literais inteiros

1	#significa 1
25	#significa 25
0x2f	#significa 47, representado em hexa
0b0010	#significa 2, representado em binário
073	#significa 59, representado em octal

Literais inteiros longos: para organizar a visualização de números literais longos introduzidos no programa, podemos utilizar o caractere sublinhado '_' para separar os milhares. Por exemplo:

150000530 pode ser representado como 150_000_530 sem influenciar o resultado da operação.

4.3 Literais reais ou de ponto flutuante

Como o Perl foi desenvolvido nos EUA e lá o indicador decimal é o ponto e não a vírgula, o Perl usa o ponto para identificar a parte racional dos números da parte inteira:

150.001	#é o mesmo que 150,001 no Brasil
29.4396	#é o mesmo que 29,4396 na notação brasileira
18.082e8	#é o mesmo que 18,082 x 10^8 ou 1_808_200_000

4.4 Quando é string e quando é número?

Quando o Perl foi criado, uma das preocupações do Larry era tornar a linguagem parecida com a linguagem humana, que é influenciada pelo contexto. O contexto é algo que deve estar subentendido pelas duas partes do diálogo.

Exemplo de contexto: "Chovem canivetes." ou "Ele me cortou com um canivete."

A palavra canivete varia conforme o contexto. E você sabe disso intuitivamente, culturalmente. É impossível chover canivetes. Mas você sabe que canivetes são perigosos. Então se alguém disser que choveram canivetes, você associa o canivete a uma situação de chuva perigosa, ou de muita chuva.

Agora, vamos levar isto para o Perl.

```
$myVar = 1;
$myOtherVar = 2;
$sum = $myVar + $myOtherVar;
```

Para o Perl **$myVar** pode significar a string '1' ou o valor numérico inteiro '1'. O mesmo vale para **$myOtherVar**. A partir do momento em que você quer somar os valores, o Perl para e pensa: eu não posso somar strings com o operador '+'. Logo, o programador quer dizer que os valores são numéricos. Isso é contexto dentro de uma visão linguística. Então o Perl guarda o valor numérico da soma em ***$sum***. Seu código continua...

```
print $sum, "\n";
```

E o compilador sabe, pelo contexto, que agora você quer tratar ***$sum*** como uma string de texto, e não mais como um número... o contexto, dentro do Perl, é como o contexto da linguagem humana. Você deve verificar a situação que rodeia o assunto para saber o real significado das coisas. É estranho no começo, mas intuitivamente você faz isso todo o tempo. Com a experiência você percebe que é muito mais prático que se dar ao trabalho de explicar ao compilador: ei, transforme isso em número... transforme isso em string..., como em outras linguagens, que obrigam o programador a toda hora converter os valores, mesmo quando isto poderia estar implícito.

4.5 Atividades comuns com os números

Além das operações comuns já vistas anteriormente de comparação entre valores, soma, subtração e multiplicação, um programador deve conhecer técnicas para:

- Verificar quando um escalar é válida como número;
- Arredondar números reais com parte decimal;
- Realizar comparações efetivas com números reais;
- Realizar operações trigonométricas, logarítmicas e com números complexos.

4.5.1 Revendo os operadores

Operação	Nome
`$a + $b`	Soma
`$a * $b`	Multiplicação
`$a % $b`	Módulo
`$a ** $b`	Exponencial

Operação	Nome	Resultado
`++$a, $a++`	Operador de auto incremento	`$a = $a + 1`
`--$a, $a--`	Operador de auto incremento	`$a = $a -1`

Operação	Numérico	String
igualdade	`==`	`eq`
diferença	`!=`	`ne`
menor	`<`	`lt`
maior	`>`	`gt`
menor ou igual	`<=`	`le`
maior ou igual	`>=`	ge
maior ou igual	`<=>`	`cmp`

4.5.2 Comparando números reais com ponto flutua te

Internamente todos os números do Perl são números de ponto flutuante baseados no tipo double do C. Essa forma interna de armazenamento algumas vezes gera erros de precisão em números de ponto-flutuante, de forma que os comparadores de igualdade, diferença, maior ou menor podem retornar uma comparação inesperada.

Veja:

```
1 #exemplo4.1
2 $a = 0.625; # 1/2 + 1/8
```

```
3 $b = 0.725; # 725/1000, or 29/40
4 printf "$a internamente é %.30g\n", $a;
5 printf "$b internamente é %.30g\n", $b;

saída:

0.625 internamente é 0.625
0.725 internamente é 0.72499999999999998
```

Então, quando for realizar comparações entre números de ponto flutuante, utilize a lógica a seguir:

```
sub sao_iguais {
     my ($num1, $num2, $precisao) = @_;
     return sprintf("%.${precisao}f", $num1) eq
sprintf("%.${precisao}f", $num2);
}
```

Esse código utiliza a flexibilidade do Perl em converter números em strings. Assim, com a função ***sprintf*** podemos definir facilmente a precisão desejada e comparar se duas strings têm valor numérico equivalente.

4.5.3 Verificando se um escalar é um núme o

Em diversas situações precisamos confirmar se os dados fornecidos pelo usuário ou armazenados em um arquivo podem ser utilizados como número. Para isso o módulo ***Scalar::Util*** contém a função ***looks_like_number***.

```
1 #!/usr/local/bin/perl
2 #exemplo4.2
3 use warnings;
4 use feature "say";
5 use Scalar::Util "looks_like_number";
6 print "Digite um número: ";
7 chomp($num = <STDIN>);
8 while($num){
9         if(!looks_like_number($num)){
```

```
 10                 say "Você deve informar um número: ";
 11                 chomp($num = <STDIN>);
 12                 next;
 13         }
 14         say "OK, $num é um número!!";
 15         last;
 16 }

saída:

Digite um número: p3c
Você deve informar um número:
36.5
OK, 36.5 é um número!!
```

NOTA: O pragma ***use warnings*** instrui o Perl a fornecer maior quantidade e detalhe nas mensagens de debug. Utilizaremos este pragma em nossos códigos daqui em diante.

4.5.4 Gerando números randômicos

```
$random = int( rand( $Y-$X+1 ) ) + $X;
```

A função ***rand*** gera um número de ponto flutuante maior que 0 e menor que o número informado no argumento. Então, se usarmos ***rand(100)***, teremos saída entre 0.0001 e 99.999... a função ***int*** remove a parte decimal do número, ficando apenas a parte inteira.

4.6 Arredondando números

No exemplo sobre comparação entre números reais eu utilizei o exemplo a seguir:

```
  1 #exemplo4.3
  2 $a = 0.625;
  3 $b = 0.725;
```

```
  4 printf "$a internamente é %.30g\n", $a;
  5 printf "$b internamente é %.30g\n", $b;

saída:

0.625 internamente é 0.625
0.725 internamente é 0.72499999999999998
```

Já dissemos que essa imprecisão é resultado da forma como o Perl armazena os números internamente. Vamos explorar um pouco mais isso. Se pudermos representar o número como uma fração cuja base seja um exponencial de 2, como 2, 4, 8, 16, podemos representar a parte decimal como um número finito. 0,625 = ½ + 1/8, que são potências de 2 e podem ser representados de forma finita internamente.

Qualquer número real cuja representação fracionária não tenha na raiz uma potência de 2 não terá uma representação finita. Mas essa diferença, em casos práticos, só é notada em aplicações que exijam acuracidade dos valores numéricos, como aplicações científicas ou financeiras.

O Perl possui métodos internos para arredondar números, veja:

```
  1 #exemplo4.4
  2 $a = 0.625; # 1/2 + 1/8
  3 $b = 0.725; # 725/1000, or 29/40
  4 printf "$a com 30 dígitos é %.30f\n", $a; #precisão de
30 dígitos
  5 printf "$b com 30 dígitos é %.30f\n\n", $b;
  6 printf "$a com 3 dígitos é %.3f\n", $a; #precisão de 3
dígitos
  7 printf "$b com 3 dígitos é é %.3f\n\n", $b;
  8 printf "$a com 4 dígitos é %.4f\n", $a; #precisão de 4
dígitos
  9 printf "$b com 4 dígitos é %.4f\n\n", $b;
 10 printf "$a com 2 dígitos é %.2f\n", $a; #precisão de 2
dígitos
 11 printf "$b com 2 dígitos é %.2f\n\n", $b;
```

```
saída:

0.625 com 30 dígitos é 0.625000000000000000000000000000
0.725 com 30 dígitos é é 0.724999999999999980000000000000

0.625 com 3 dígitos é 0.625
0.725 com 3 dígitos é 0.725

0.625 com 4 dígitos é 0.6250
0.725 com 4 dígitos é 0.7250

0.625 com 2 dígitos é 0.63
0.725 com 2 dígitos é 0.72
```

O aparente defeito da técnica de arredondamento do Perl não é um defeito, mas resultado do padrão **IEEE-754**. Perceba, por exemplo, que, ao solicitarmos ao Perl que imprima 0,725 com 4 dígitos de precisão, ele apresenta 0,7250. Mas se solicitarmos com 2 dígitos ele apresenta 0,72, enquanto que o valor 0,625 tornou-se 0,63. Isso porque internamente 0,725 é 0,7249999... e quando pedimos que ele represente com 2 dígitos, ele verifica o terceiro dígito que é 4, portanto arredonda para menor, ficando 0,72. Enquanto que o terceiro digito de 0,625 é 5, arredondando para maior, tornando-se 0,63.

NOTA: Especificações de computadores com 16, 32 ou 64 bits ou mesmo sistemas compilados em diferentes especificações podem ter saídas levemente diferentes. Por exemplo:
0.725 com 30 dígitos é 0.724999999999999977795539507497 no Fedora 15, 32 bits.

Mas o que fazer quando a acuracidade do valor é importante na minha aplicação? Você pede ajuda ao CPAN. Digite na sua linha de comando ***cpan install Math::Round*** e deixe o CPAN trabalhar. Ele irá instalar o módulo Math::Round na sua biblioteca do Perl. Essa é nossa primeira instalação de módulos CPAN no Perl.

```
  1 #exemplo4.5
  2 use feature "say";
  3 use Math::Round "nearest"; #Informa que meu programa
irá usar o Math::Round
```

```
  4 #Assim o Perl irá carregar o módulo e disponibilizar a
função nearest
  5
  6 $a = 0.625;
  7 $b = 0.725;
  8 $c = 0.004;
  9 $d = 0.006;
 10
 11 printf "$a arredondado é %.2f\n", nearest(.01, $a);
#Precisão de 2 dígitos
 12 printf "$b arredondado é %.2f\n", nearest(.01, $b);
#Precisão de 2 dígitos
 13 printf "$c arredondado é %.2f\n", nearest(.01, $c);
#Precisão de 2 dígitos
 14 printf "$c arredondado é %.3f\n", nearest(.001, $c);
#Precisão de 3 dígitos
 15 printf "$d arredondado é %.2f\n", nearest(.001, $d);
#Precisão de 2 dígitos
 16 printf "$d arredondado é %.3f\n", nearest(.001, $d);
#Precisão de 3 dígitos
 17 say round $a;
 18 say round $b;
 19 say round $c;
 20 say round $d;
```

saída:

```
0.625 arredondado é 0.63
0.725 arredondado é 0.73
0.004 arredondado é 0.00
0.004 arredondado é 0.004
0.006 arredondado é 0.01
0.006 arredondado é 0.006
1
1
0
0
```

Veja que a função ***nearest*** do módulo Math::Round possui dois argumentos. O primeiro define a acuracidade, onde o valor deve ser arredondado. O segundo é o valor a ser manipulado. Assim, podemos controlar efeitos indesejados do padrão IEEE-754 quando precisarmos definir a acuracidade dos dados. Este módulo também possui a função ***round,*** que arredonda o número para o inteiro mais próximo.

4.7 Números complexos com Math::Complex

```
  1 #exemplo4.6
  2 use warnings; #Ativa warnings
  3 use feature "say"; #ativa say
  4 use Math::Complex; #Carrega o módulo Math::Complex
  5
  6 $num1 = cplx (5,4); #cria o número complexo $num1, 5
+4i
  7 $num2 = cplx (20,-8); #cria o número complexo $num2, 20
-8i
  8 say '$num1 é: ',$num1;
  9 say '$num2 é: ',$num2;
 10 say '$num1 + $num2 = ',$num1 + $num2; #soma complexos
 11 say '$num1 x $num2 = ', $num1 * $num2; #multiplica com-
plexos
 12 say '$num1 - $num2 = ', $num1 - $num2; #subtrai comple-
xos

saída:

$num1 é: 5+4i
$num2 é: 20-8i
$num1 + $num2 = 25-4i
$num1 x $num2 = 132+40i
$num1 - $num2 = -15+12i
```

Para trabalhar com números complexos dentro do Perl o processo é bem simples. Utilizamos o módulo Math::Complex, que já vem com o Perl, e podemos criar números complexos com a função ***cplx(REAL, IMAGINÁRIO)***. Após criar o número complexo podemos realizar as operações que desejarmos.

Bem, até aqui vimos muito sobre strings e números. Agora, para que nosso Perl--fu evolua, precisamos aprofundar nossos conhecimentos em funções, loops e estruturas de controle de fluxo.

Exercícios

1. Monte duas versões de um programa que gere na saída uma tabela de conversão de graus Celsius para Fahrenheit. Uma que não mostre as casas decimais da saída, apenas os valores inteiros arredondados. Outra que tenha precisão de três casas decimais.

2. Faça um pequeno programa de operações com números complexos fornecidos pelo usuário.

Capítulo 5. Operadores, Decisão e Controle de Fluxo

O Perl é uma linguagem muito otimizada para facilitar a vida do programador. Essa facilidade é parte de sua filosofia: "Devem existir várias maneiras de fazer isso com Perl". Por isso a linguagem tem muitos operadores. Na verdade, beirando os 70. Isso porque estamos contando apenas os que não são considerados funções: operadores matemáticos, lógicos, match etc. Em algumas literaturas sobre Perl as funções internas são consideradas operadores. A bem da verdade, tanto para mim quanto para Larry Wall **não importa o nome que você prefira dar às coisas, desde que você entenda o que elas fazem**. Apesar de tantos operadores, verbos, módulos e substantivos, a linguagem internamente é bem enxuta. O Perl 5.8 tinha cerca de 350 opcodes – o que é relativamente pouco. Isso leva à conclusão de que diversos operadores e funções trazem a mesma sequência de opcodes. Mas geram uma linguagem muito expressiva tal qual a linguagem humana.

Outra consideração importante diz respeito ao controle de fluxo. Os programas seguem sequencialmente, executando comando após comando, linha após linha. O controle de fluxo gera desvios e repetições na execução linear do código, após avaliar uma expressão, tornando a avaliação de expressões, a decisão, o controle de fluxo e os operadores temas intimamente ligados, e é disso que vamos tratar a fundo agora. Você pode, durante a leitura deste capítulo ou após, se aprofundar nestes assuntos estudando a página de manual do Perl perlop, no site http://perldoc.perl.org ou com o comando:

```
$perldoc perlop
```

Bem... dissemos anteriormente que os operadores devem realizar certos processamentos fundamentais nos dados. Nós vimos superficialmente alguns operadores como operador de atribuição, operadores de comparação, etc. Mas você, a esta altura, já está pronto para estudos detalhados.

5.1 O operador de atribuição '='

O operador de atribuição informa que o valor à sua direita deve ser atribuído ao substantivo à sua esquerda. Mas existem alguns truques que todo programador Perl deve conhecer.

Vamos lá:

```
$var = 1;
$var2 = 2;
```

Para atribuir o valor de uma variável a outra:

```
$var1 = $var2.
```

Aqui um grande truque, para intercambiar o valor de duas variáveis:

```
($var1, $var2) = ($var2, $var1);
```

Atribuir um mesmo valor a diversas variáveis:

```
$i = $j = $k = $y = 0;
```

Ou diversos valores a diversas variáveis:

```
($var1, $var2, $var3) = (1, 2, 3);
```

Atribuir valores a arrays:

```
@array = (1, 2, 3, 4, 5, 6, 7);
```

Ou apenas a um índice:

```
$array[2] = 3;
```

Ou ainda a uma porção do array:

```
$array[ 2 .. 5 ] = (23, 11, 30, 18);
```

Criar um hash:

```
%hash = (str1 => 'casa', str2 => 'carro', str3 => 'tv');
```

Ou:

```
@array = ('str', 'casa', 'str2', 'carro', 'str3', 'tv');
```

%hash = @array #obs: array precisa ter um número par de elementos para ser convertido em hash.

5.2 Operadores aritméticos

O Perl provê operadores aritméticos para todas as operações básicas. Soma (+), subtração (-), multiplicação (*), divisão (/), módulo (%) e exponencial (**). Além destes, também podemos usar os parênteses ( ). Esses operadores seguem a precedência comum da matemática. Primeiro são executados os exponenciais, depois as multiplicações e divisões. Por último as somas e subtrações. Elementos entre parênteses têm precedência sobre os outros. Parênteses podem ser internos a outros parênteses ou, como se diz formalmente, aninhados.

```
$var =  3**2*2;      # produz 18 => 3**2 = 9, 9*2 = 18
$var = 2*3**2;          # produz 18 => 3**2 = 9, 2*9 = 18.
                          Exponenciais executam primeiro.

$var = 3*2 + 4*4;    #produz 22
$var = 3*(2+4)*4;    #produz 72
```

5.3 Operadores de deslocamento binários (shift)

Os operadores de deslocamento binários são importantes em programação de baixo nível e para o desenvolvimento de algoritmos de criptografia. Para quem deseja, por exemplo, controlar outros elementos via porta serial ou USB pode achar o site http://www.easysw.com/~mike/serial/serial.html uma fonte interessante para aprender os princípios de como programar em baixo nível. No CPAN existem módulos que realizam a interface entre Perl e certos elementos de hardware como portas USB, joysticks e portas seriais. Mas o uso mais comum nessas situações é uma mistura de Perl e C.

Veja o resultado das operações com o operador ***shift***:

```
perl -E "say 1 << 1;"
```

Mostra 2. Por quê? A representação binária de 1 é 00000001. Efetuando um deslocamento para a esquerda, teremos 00000010, que é a representação binária para 2.

```
perl -E "say 3 << 4 ;" #Mostra 48 00000011 torna-se 00110000
```

Da mesma forma:

```
perl -E "say 48 >> 4 ; " #Mostra 3, pois efetua o desloca-
mento binário para a direita.
```

Um bom truque, se você olhar com cuidado:

```
x << n sempre resulta em x*2**n
```

Exemplo:

```
perl -E "say 3 << 2; " #retorna 12. 2**2 = 4, 4*3= 12;
perl -E "say 3 << 4; " #retorna 48. 2**4 = 16, 16*3 = 48.
```

NOTA: Esse truque eu aprendi programando Assembler para microcontroladores.

5.4 Operadores de strings

Os operadores de string já foram vistos. Eles são **x** e o operador ponto "." . Mas ainda temos alguns truques para tirar da manga:

```
@array = (1, 2, 3) x 3;          # @array = (1, 2, 3, 1, 2,
@array1 = (1, 2, 3);             3, 1, 2, 3)
@array2 = @array1 x 3;           # @array2 contém (3, 3, 3)
                                 pois @array1 tem 3 elemen-
                                 tos
@array2 = (@array1) x 3;         # @array2 contém (1, 2, 3,
                                 1, 2, 3, 1, 2, 3)
print "string" x (1, 2, 3);      # "stringstringstring"
```

5.5 Operadores lógicos

Veja a tabela a seguir com os operadores booleanos mais comuns:

Operador	Retorna
and &&	Verdadeiro se todos os termos forem verdadeiros
or ||	Verdadeiro se ao menos um termo for verdadeiro
xor	Verdadeiro se um for verdadeiro e o outro falso
not !	Verdadeiro se o elemento for falso

Vamos ver um exemplo:

```
  1 #exemplo5.1
  2 use feature qw(say);
  3
  4 #Zero é falso. && é verdadeiro se todos elementos forem
verdadeiros
  5 say (1 && 0);
  6 say (1 && 1);   #1 && 1 retorna verdadeiro
```

```
  7 say (1 && undef); #undef é falso
  8 say ('e' && ''); #falso
  9 say ('e' && 'i'); #verdadeiro
 10 say ('e' && '0'); #falso
 11 say ('1' or ''); #verdadeiro. or é verdadeiro se um
elemento for verdadeiro
 12 say ('' or '0'); #falso
 13 say ('casa' or undef); #verdadeiro
 14 say ('0' or ''); #falso
 15 say (0 xor 1); #verdadeiro
 16 say (0 xor 0); #falso
 17 say (0 or undef or '' or "volta 1"); #verdadeiro, volta
1 é verdadeiro
 18 say (0 xor 1 xor 0); #verdadeiro
saída:

0
1


i
0
1
0
casa

1

volta 1
1
```

É importante saber que no Perl os valores para falso são: 0, "0", "" e undef. Todos os outros valores são considerados verdadeiros. Assim, qualquer um destes valores retornados será interpretado no contexto booleano como falso. E todos os outros valores são considerados verdadeiros neste contexto.

O Perl precisa retornar apenas o valor da última expressão executada que define o resultado. Então para retornar falso, o Perl retorna o valor da última expressão falsa - exemplo: ***'1' && '0'*** retorna **0,** já que a expressão é falsa e o último valor

falso lido foi **0**. Por outro lado, **0 or undef** retorna **undef** pois a expressão é falsa e **undef** é o último valor falso lido. O mesmo para expressões verdadeiras: '***e' && 'i'*** retorna 'i', pois a expressão é verdadeira e 'i' é o último valor verdadeiro lido.

A principal finalidade dos operadores de comparação e dos operadores lógicos booleanos é **controlar o fluxo de execução do programa** juntamente com os operadores de controle de fluxo e de loop. Veja os operadores lógicos juntamente com as estruturas de controle e loop:

```
 if (EXPRESSAO1 AND EXPRESSAO2){
     SEU CÓDIGO...
 }

 if(EXPRESSAO1 AND EXPRESSAO2 AND … AND EXPRESSAON){
     SEU CÓDIGO...
}

while (EXPRESSAO1 OR EXPRESSAO2 OR … OR EXPRESSAON){
     SEU CÓDIGO...
}

unless (EXPRESSAO1 XOR EXPRESSAO2){
     SEU CÓDIGO...
}

if ( not (EXPRESSAO1 AND EXPRESSAO2)){
     SEU CÓDIGO...
}
```

Exemplo:

```
1 #exemplo5.2
2 use 5.12.0;
3
4 my $var1 = 0;
5 my $var2 = 5;
```

```
  6 my $var3 = $var2;
  7 my ($str1, $str2, $str3) = ('casa', 'carro', 'casa');
  8 if ($var1 == $var2 or $str1 eq $str3){
  9  say "basta uma expressão verdadeira para que or retor-
ne verdadeiro";
 10 }
 11 if ((say "Hello") and (say "bye")){
 12  say "Você pode usar funções nas expressões!!"
 13 }
 14 if ((say "Hello") or (say "Esse nunca vai ser executa-
do!!!")){
 15  say "\n\nO Perl executa/verifica as expressões apenas
enquanto " ,
 16        "for necessário\n Como basta um elemento verda-
deiro para definir ",
 16   "uma operação or\n O segundo elemento nunca será ",
 17   "executado/verificado se o primeiro for verdadei-
ro!!!"
 18 }
```

saída:

```
basta uma expressão verdadeira para que or retorne verda-
deiro
Hello
bye
Você pode usar funções nas expressões!!
Hello


O Perl executa/verifica as expressões apenas enquanto for
necessário
Como basta um elemento verdadeiro para definir uma operação
or
O segundo elemento nunca será executado/verificado se o
primeiro for verdadeiro!!!
```

Os operadores lógicos do Perl têm uma característica chamada **curto circuito**. Isso significa que, ao fazer a verificação lógica, assim que for possível definir o resultado, o operador retorna o valor da expressão sem verificar/executar os outros elementos (basta que um elemento seja verdadeiro em uma operação **or** ou que apenas um seja falso em uma operação **and**).

Assim é possível usar os operadores lógicos como controladores de fluxo simples, veja:

```
  1 #exemplo5.3
  2 use 5.12.0;
  3 use FileHandle;
  4
  5 (say "hello") && (say " Olá!!!") && (say "Qualquer coi-
sa mais");
  6
  7 (0 + 0) && (say "Nunca executa");
  8
  9 my $fh = FileHandle->new("arquivo_inexistente", 'r')
 10  or die "Erro ao abrir arquivo: $!";

saída:

hello
 Olá!!!
Qualquer coisa mais
Erro ao abrir arquivo: No such file or directory at C:/Pro-
gram Files
```

NOTA: A variável interna $! armazena a mensagem de erro, que pode ser exibida com a função **die.** A função **die** interrompe a execução do programa bruscamente.

5.6 Operadores bitwise

Ainda há pouco, neste capítulo, falamos dos "shift operators" (operadores de deslocamento), que atuam em baixo nível, ou seja, na parte binária do dado. Os operadores bitwise também atuam no nível do bit, realizando operações aritméticas e lógicas bit a bit. Por exemplo, supondo uma representação de 8 bits, o número 7 em binário é: 00000111. O número 9 é 00001001. Adiante temos um exemplo das operações de bitwise em cada um dos bits do número.

NOTA: O exemplo adiante supõe uma representação de 8 bits para os dados. A representação é meramente didática, pois seu computador deve ter de 16 a 64 bits para representar os dados.

```
        00000111
and     00001001
        00000001  (1)

        00000111
or      00001001
        00001111 (15)

        00000111
xor     00001001
        00001110 (14)
```

O operador not (~) é unário, ou seja, trabalha apenas com um elemento por vez:

```
not     00000111 (~7)
        11111000 (248)
```

Operador	Função
&	Bitwise and
|	Bitwise or
^	Bitwise xor
~	Bitwise not

```
  1 #exemplo5.4
  2 use warnings;
  3 use feature ("say");
  4 $vigia = 15;
  5
  6 while ($vigia){
  7     printf "$vigia em binário = %04b\n", $vigia;
  8     printf "Um and binário: %04b\t\%d\n", $vigia & 2,
$vigia & 2;
  9     printf "Um or binário: %04b\t\%d\n", $vigia | 2,
$vigia | 2;
 10     printf "Um xor binário: %04b\t\%d\n", $vigia ^ 2,
$vigia ^ 2;
 11     printf "Um not binário: %04b\n", ~$vigia;
 12     say "#" x 60;
 13     say "";
 14     $vigia = $vigia - 1;
 15 }
```

saída:

```
15 em binário = 1111
Um and binário: 0010   2
Um or binário: 1111    15
Um xor binário: 1101   13
Um not binário: 11111111111111111111111111110000
############################################################
#

14 em binário = 1110
Um and binário: 0010   2
Um or binário: 1110    14
Um xor binário: 1100   12
Um not binário: 11111111111111111111111111110001
############################################################
#

13 em binário = 1101
```

```
Um and binário: 0000   0
Um or binário: 1111    15
Um xor binário: 1111   15
Um not binário: 11111111111111111111111111110010
##########################################################
#

12 em binário = 1100
Um and binário: 0000   0
Um or binário: 1110    14
Um xor binário: 1110   14
Um not binário: 11111111111111111111111111110011
##########################################################
#

11 em binário = 1011
Um and binário: 0010   2
Um or binário: 1011    11
Um xor binário: 1001   9
Um not binário: 11111111111111111111111111110100
##########################################################
#

...saída muito longa omitida...
```

5.7 Operadores de incremento e decremento

Os operadores ++ e - - realizam a operação de incremento ou decremento respectivamente, que seria adicionar ou subtrair 1 ao valor da variável. Então: $var ++ é o mesmo que $var = $ var + 1 e $var-- é o mesmo que $var = $var – 1. Um detalhe importante sobre os operadores de incremento e decremento é o pós-decremento/incremento e o pré-incremento/decremento.

Veja:

```
1 #exemplo5.5
2 use 5.12.0;
3 my $var = 1;
```

```
4 my $var2 = 1;
5
6 say $var++; #pós-incremento
7 say $var;
8
9 say ++$var2; #pré-incremento
```

saída:

```
1
2
2
```

Na configuração de pré-incremento, a variável é incrementada antes da expressão ser avaliada. No pós-incremento, a variável é incrementada após a expressão ser avaliada.

Isso também vale para caracteres e strings:

```
1 #exemplo5.6
2 use 5.12.0;
3
4 my $var = 'a';
5 my $var2 = 'J';
6 say $var++; #pós-incremento
7 say $var;
8 say ++$var2; #pré-incremento
```

saída:

```
a
b
K
```

5.8 Combinando os operadores

Matemáticos:	String	Shift	Logical	Bitwise
+=	x=	<<=	||=	|=
-=	.=	>>=	&&=	&=
*=				^=
/=				
**=				
%=				

Os operadores combinados são mais uma forma de abreviação, utilizando a variável à esquerda como parte da expressão e ao mesmo tempo têm o resultado da expressão atribuído a ela:

```
 1 #exemplo5.7
 2 use feature "say";
 3 use warnings;
 4
 5 $num1 = 50;
 6 $num2 = 5;
 7 $num1 += $num2; #mesmo que $num1 = $num1 + $num2
 8 say $num1;
 9 $num2 *= $num1; #mesmo que $num1 = $num1 * $num2
10 say $num2;
11 $num1 /= $num2; #mesmo que $num1 = $num1 / $num2
12 say $num1;
13 $str1 = "Perl";
14 $str2 = " rocks\n";
15 $str1 .= $str2; #mesmo que $str1 = $str1 . $str2
16 $str1 x= 10; #mesmo que $str1 = $str1 x 10;
17 say $str1;

saída:

55
275
```

```
0.2
Perl rocks
Perl rocks
Perl rocks
Perl rocks
Perl rocks
Perl rocks
Perl rocks
Perl rocks
Perl rocks
Perl rocks
```

5.9 Operadores de comparação

Já usamos bastante operadores de comparação e você já sabe para que eles servem – comparam duas expressões e retornam verdadeiro ou falso. Porém, por questões de simplicidade, eu não havia comentado sobre todos os operadores. Temos alguns detalhes interessantes para ver.

Numéricos	String	Operação
`!=`	`ne`	Retorna verdadeiro se os operandos forem diferentes
`>`	`gt`	Retorna verdadeiro se o operando da esquerda for maior que o da direita
`==`	`eq`	Retorna verdadeiro se os operandos forem iguais
`>=`	`ge`	Retorna verdadeiro se o operando da esquerda for maior ou igual que o da direita
`<`	`le`	Retorna verdadeiro se o operando da esquerda for menor que o da direita
`<=`	`lt`	Retorna verdadeiro se o operando da esquerda for menor ou igual que o da direita
`<=>`	`cmp`	Retorna -1 se o operando da esquerda for menor que o da direita, 0 se forem iguais e 1 se o operando da esquerda for maior que o da direita.

Frisando que os comparadores de string usam como base a posição na tabela ASCII do primeiro caractere da string, que, se forem iguais, irá verificar o segundo elemento e então o terceiro... para determinar qual é o maior ou menor elemento. Se desejar saber o tamanho de uma string use a função **length.**

Aqui vale um exemplo com o operador <=>, já que ainda não fizemos nenhum exemplo com ele ainda.

```
 1 #exemplo5.8
 2 use feature "say";
 3 use warnings;
 4
 5 print "Digite um número: ";
 6 chomp (my $num1 = <STDIN>);
 7 print "Digite outro número: ";
 8 chomp (my $num2 = <STDIN>);
 9 $comp = $num1 <=> $num2;
10 if ($comp == -1) {say "$num2 é maior"}
11 if ($comp == 0 ) {say "São iguais"}
12 if ($comp == 1)  {say "$num1 é maior"}
13 print "Digite um caractere: ";
14 chomp ($char1 = <STDIN>);
15 print "Digite outro caractere: ";
16 chomp ($char2 = <STDIN>);
17 $comp = $char1 cmp $char2;
18 if ($comp == -1) {say "$char2 é maior"}
19 if ($comp == 0 ) {say "São iguais"}
20 if ($comp == 1)  {say "$char1 é maior"}
```

saída:

```
$ perl exemplo5.8.pl
Digite um número: 45
Digite outro número: 33
45 é maior
Digite um caractere: c
Digite outro caractere: c
São iguais
```

Esse operador pode ser utilizado juntamente com a função ***sort*** para ordenar os elementos de um array:

```
  1 #exemplo5.9
  2 use feature "say";
  3 use warnings;
  4
  5 @desordem = (15,200,18,27,52,315,1,3,0,87,52,75);
  6 @ordem = sort{$a <=> $b} @desordem;
  7 say "@ordem";
  8 @desordem = qw(w X y v A s t U s Casa cupim Warning s 2
12 7 23 b T s);
  9 @ordem = sort{$a cmp $b} @desordem;
 10 say "@ordem";

saída:

0 1 3 15 18 27 52 52 75 87 200 315
12 2 23 7 A Casa T U Warning X b cupim s s s s t v w y
```

Observe neste exemplo que a saída com o operador ***cmp*** é ordenada baseada na ordem do elemento na tabela ASCII e não pelo seu valor numérico. Por isso 12 vem antes de 2 e 23 antes de 7.

5.10. Operador ternário

O operador ternário é uma abreviação do conjunto if/else, funcionando como a seguir:

```
if <expressão1> then return <expressão2> else return <ex-
pressao3>
```

Os operadores são **"?"** e **":"**.

Veja:

```
  1 #exemplo5.10
  2 use feature "say";
  3
  4 my $num1 = 50; #declaro dois números diferentes
  5 my $num2 = 5;
  6
  7 #    Se $num1 == $num2  então "São iguais" se não "São
diferentes"
  8 say ($num1 == $num2 ? "São iguais" : "São Diferentes");
  9 $num1 = $num2; #torno eles iguais
 10
 11 #    Se $num1 == $num2  então "São iguais" se não "São
diferentes"
 12 say ($num1 == $num2 ? "São iguais" : "São diferentes");

saída:

São diferentes
São iguais
```

5.11 Loops e estruturas de decisão

Como dissemos em capítulos anteriores, **os loops permitem a execução de código de forma repetitiva enquanto uma determinada condição for verdadeira**. E **as estruturas de decisão executam uma porção de código uma única vez se a condição for verdadeira**. O Perl tem uma grande variedade de loops e estruturas de decisão pois, como dissemos antes, sempre deve haver "várias maneiras de fazer isso com Perl".

5.11.1 Decisão

Já vimos a estrutura ***if (condição){código}***. Porém, os blocos if podem ser complementados com estruturas ***eslif(condição){código} else{código}***. Criamos muitos programas que utilizavam vários blocos ***if*** para verificar diversas condições.

```
if (condição1) {
     bloco de código
}

if (condição 2) {
     bloco de código
}

if (condição 3) {
     bloco de código
}...
```

a estrutura nova ficaria:

```
 if (condição 1){
     bloco de código
}
elsif (condição 2) {
     bloco de código
}
elsif (condição 3) {
     bloco de código
}
else{
     bloco de código
}
```

As duas grandes vantagens em executar uma estrutura if-elsif-else são:

Quando uma condição for satisfeita e o código daquele bloco executado, o Perl passa para o código após a estrutura, sem a necessidade de checar as outras condições do bloco.

Se nenhuma condição for satisfeita, o Perl irá executar o código dentro do bloco else, que pode conter um código para tratar condições não previstas. Caso não haja um bloco else, o Perl irá executar o código após a estrutura.

```
1 #exemplo5.11
2 use feature "say";
3 use warnings;
4
```

```
 5 say "Vamos ver como o if funciona: ";
 6 say "Quantos anos você tem? ";
 7 chomp ($idade = <STDIN>);
 8
 9 if ($idade < 18){
10         say "Você não é maior de idade";
11 }
12 elsif ($idade >= 18 && $idade < 40){
13         say "Você já é de maior";
14 }
15 else {
16         say "Você é muito velhinho";
17 }
```

saída:

```
Vamos ver como o if funciona:
Quantos anos você tem?
10
Você não é maior de idade

Vamos ver como o if funciona:
Quantos anos você tem?
20
Você já é de maior

Vamos ver como o if funciona:
Quantos anos você tem?
45
Você é muito velhinho
```

5.11.2 O bloco *unless*

O bloco ***unless*** tem o funcionamento inverso do bloco if. Ou melhor, funciona como o bloco **if com o operador not no início da expressão condicional.** O bloco unless significa "a não ser que". Pode parecer inútil, mas aumenta a legibilidade do código em algumas situações e também torna a linguagem mais expressiva.

unless (expressão) é o mesmo que if (not expressão).

```
unless (condição){
     bloco de código
}
elsif (condição) {
     bloco de código
}
else {
     bloco de código
}
```

```
 1 #exemplo5.12
 2 use feature "say";
 3 use warnings;
 4
 5 say "Vamos ver como o unless funciona: ";
 6 say "Quantos anos você tem? ";
 7 chomp (my $idade = <STDIN>);
 8 unless ($idade >= 18){
 9         say "Você não é maior de idade";
10 }
11 else {
12         say "Você já é de maior";
13 }
```

saída:

```
Vamos ver como o unless funciona:
Quantos anos você tem?
17
Você não é maior de idade

Vamos ver como o unless funciona:
Quantos anos você tem?
20
Você já é de maior
```

5.11.3 O módulo switch

Além dos blocos condicionais ***if*** e ***unless,*** o Perl provê com o módulo switch o bloco condicional ***switch case.*** Ele funciona verificando diversas condições com um escalar dado. Seu uso é exemplificado a seguir, porém ele foi descontinuado, ou seja, seu uso não é mais recomendado, e futuramente o Perl não mais dará suporte para este módulo.

```
 1 #exemplo5.13
 2 use feature "say";
 3 use warnings;
 4 use Switch;
 5
 6 print "Digite um valor entre 0 e 5: ";
 7 chomp (my $numero = <STDIN>);
 8 switch ($numero) {
 9         case 0 {say "Você digitou Zero"}
10         case 1 {say "Você digitou Um"}
11         case 2 {say "Você digitou Dois"}
12         case 3 {say "Você digitou Três"}
13         case 4 {say "Você digitou Quatro"}
14         case 5 {say "Você digitou Cinco"}
15         else {say "Valor digitado fora de escala"}
16 };

saída:

Switch will be removed from the Perl core distribution in
the next major release. Please install it from CPAN. It is
being used at Livro_exemplo5.13.pl, line 4.
Digite um valor entre 0 e 5: 10
Valor digitado fora de escala
```

5.11.4 Given when

A estrutura **given when** é uma melhoria adicionada na versão 5.10.0 do Perl e foi otimizada na versão 5.12.0. Ela funciona com a cláusula ***given*** recebendo um escalar, e cada cláusula ***when*** testa uma possibilidade. Caso nenhuma expressão das cláusulas ***when*** seja verdadeira, a cláusula ***default*** é executada se ela existir. A documentação do Perl apresenta o seguinte exemplo:

```
given($foo) {
   when (/x/) { say '$foo contains an x'; continue }
   when (/y/) { say '$foo contains a y' }
   default    { say '$foo does not contain a y' }
}
```

Neste exemplo a variável ***$foo*** é comparada com as regex ***/x/*** e com ***/y/***. A cláusula ***continue*** informa que, caso aquela linha seja executada, todas as outras condições devem ser verificadas. Sem a cláusula ***continue***, quando uma expressão de um bloco ***when*** for verdadeira ela será executada e todo o bloco ***given*** é abandonado. Como **say** e **given when** não existem nas versões anteriores do Perl, para ativar esta funcionalidade use o pragma ***use feature ":5.12"*** ou ***use 5.12.0.*** Veja um código que, se melhorado, pode funcionar como o comando linux ***wc:***

```
  1 #exemplo5.14;
  2 use feature ":5.12"; #mesmo que use 5.12.0;
  3 use FileHandle;
  4
  5 die("Uso: $0 <nome_do_arquivo>") if not $ARGV[0];
  6 $! = undef;
  7 my $file = FileHandle->new("$ARGV[0]", "r")
  8         or die "Não foi possível abrir o arquivo
$ARGV[0] $!\n";
  9 my $linhas = 0;
 10 my $numeros = 0;
 11 my $espacos = 0;
 12 my $tabs = 0;
 13 my $chars = 0;
 14 while (!$file->eof()){
 15 #o eof significa fim de arquivo.
 16 #Assim, enquanto não for fim de arquivo o loop será
executado.
 17       my $char = $file->getc();
 18       ++$chars;
 19       given ($char){
 20              when (/\n/) {++$linhas; continue}
 21              when (/\d/) {++$numeros; continue}
```

```
22                when ($char eq ' ') {++$espacos; continue}
23                when (/\t/) {++$tabs; continue}
24        }
25 }
26 $file->close();
27 printf "O arquivo $ARGV[0] possui:\n%d caracteres; " .
28  "\n%d espaços;\n%d linhas;\n%d dígitos;\n%d tabs\n",
29  $chars, $espacos, $linhas, $numeros, $tabs;
```

Veja a saída do nosso comando comparada com o comando **wc** do linux:

```
$ perl exemplo5.14.pl /etc/passwd
O arquivo /etc/passwd possui:
1679 caracteres;
17 espaços;
35 linhas;
148 dígitos;
0 tabs

$ wc -c /etc/passwd
1679 /etc/passwd

$ wc -l /etc/passwd
35 /etc/passwd
```

Uma implementação simples do comando wc em linguagem C tem cerca de 141 linhas. Caso seja de seu interesse, o CD que acompanha o livro contém o código-fonte de implementação em C deste comando no arquivo exemplo_wc.c. É um desafio interessante criar uma versão completa deste comando em Perl e comparar com a implementação C.

5.12 Loops

A esta altura você deve estar familiarizado com o loop ***while.*** Mas o Perl fornece ainda os loops **for, foreach, until** e o modificador ***do.*** Quando vamos escrever um loop que necessite de uma variável de iteração, usamos a estrutura:

```
my $i = 0;
while($i<100){
     código;
     $i++;
}
```

Porém, podemos fazer isto de uma forma mais simples com o loop ***for.***

```
for(my $i; $i<1000; $i++){
     código;
}
```

Os opcodes gerados são essencialmente os mesmos que os gerados pelos outros loops, oferecendo apenas uma facilidade para o programador.

```
  1 #exemplo5.15
  2 use 5.12.0;
  3
  4 for (my $i = 0; $i <= 10; $i++){
  5         print"$i\t";
  6 }
  7 say "\nfim!";

saída:

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

Note que a variável $i pode ser qualquer variável do seu algoritmo, bem como uma expressão de avaliação qualquer.

5.12.1 Loops para hashes e arrays

Uma atividade comum com hashes e arrays é efetuar um processamento em todos os elementos. Para facilitar esta atividade corriqueira, o Perl fornece algumas estruturas que realizam determinada tarefa elemento por elemento, além de algumas variáveis úteis.

Como sabemos, todo array começa com índice 0. E o índice do último elemento alocado de um array é ***$#nome_do_array***. Assim, uma maneira de executar um processamento em todo um array seria como mostra este trecho de código:

```
my $var = 0;
while ($#nome_do_array >= $var++){
     say $nome_do_array[$var];
}
```

Mas o Perl vai além e fornece o loop ***foreach,*** que executa o código uma vez para cada elemento de um dado array:

```
  1 #exemplo5.16
  2 my @array = (1..10); #crio um array de 1 a 10
  3 #$var representa o elemento do array durante a iteração
do loop.
  4 foreach my $var (@array){
  5         $var++;
  6 }
  7 say "@array";

saída:

2 3 4 5 6 7 8 9 10 11
```

O loop associa cada elemento do array a uma variável informada, realizando o processamento desejado em cada elemento do array, alterando o valor de todas as suas variáveis. Caso uma variável não seja informada, a variável padrão **$_** será utilizada.

```
  1 #exemplo5.17
  2 my @array = (1..10);
  3 foreach(@array){
  4         ++$_;
  5 }
  6 say "@array";
```

Embora o loop ***foreach*** seja considerado o loop padrão para arrays, no Perl os loops ***for*** e ***foreach*** geram os mesmos opcodes. Assim:

```
for(@array){
     ++$_;
}
```

Seria o mesmo loop de forma mais abreviada.

5.12.2 A função *keys*

A função ***keys*** gera um array contendo as chaves de um hash dado como argumento. Vamos usar a variável interna do Perl %ENV neste exemplo. Esta variável armazena um hash com as variáveis de ambiente como chave e o valor das variáveis como elemento:

```
  1 #exemplo5.18
  2 use 5.12.0
  3 for (keys %ENV){
  4         say "$_ => $ENV{$_}"
  5 }

saída:

USERPROFILE => C:\Users\thiago
HOMEDRIVE => C:
COMMONPROGRAMFILES => C:\Program Files\Common Files
VBOX_INSTALL_PATH => C:\Program Files\Oracle\VirtualBox\
APPDATA => C:\Users\thiago\AppData\Roaming
PROGRAMFILES => C:\Program Files
OS => Windows_NT
LOCALAPPDATA => C:\Users\thiago\AppData\Local
PATHEXT => .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.
WSH;.MSC
GTK_HOME => C:\Program Files\Gtk+
SDL_INST_DIR => c:\Perl\bin
NUMBER_OF_PROCESSORS => 2
FP_NO_HOST_CHECK => NO
HOMEPATH => \Users\thiago
```

Esta saída pode mudar conforme o sistema utilizado.

NOTA: A partir da versão 5.12.0 a função ***keys*** foi otimizada, podendo ser utilizada com arrays. No caso dos arrays a função retorna uma lista de índices do array informado como argumento. Veja no próximo exemplo a função ***keys*** utilizada com um array.

5.12.3 A função *map*

Assim como as funções ***foreach e for***, o comando ***map*** associa o valor de cada elemento de um array, realiza o processamento e retorna um array com o resultado dos processamentos.

```
  1 #exemplo5.19
  2 my @array = (0..9);
  3 my @mapeado = map {++$_} @array;
  4 say "$mapeado[$_] => $array[$_]" for keys @array;

saída:

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

E podemos utilizar a função ***grep*** para montar um array com todos os elementos de um outro array que atendem a uma determinada expressão.

```
  1 #exemplo5.20
  2 my @array = (0..9);
  3 my @grep = grep {$_ > 5} @array;
```

```
  4 say "@grep";

saída:

6 7 8 9
```

5.12.4 A função *values*

A função ***values*** retorna um array com todos os valores de um hash.

```
  1 #exemplo5.21
  2 use 5.12.0;
  3 my %End = (cidade => "São Paulo",
  4           estado => "São Paulo",
  5           bairro => "Tucuruvi");
  6 map {say} values %End;
  7 my @sp = grep {/São|ad/} %End;
  8 say "@sp";

saída:

Tucuruvi
São Paulo
São Paulo
cidade São Paulo estado São Paulo
```

5.12.5 A função *each*

A função ***each*** retorna um array com um par de elementos. Se fornecermos um hash como argumento, retorna um array com dois valores: uma chave e seu elemento no hash. Se utilizar um array como argumento, retorna o índice do array e o escalar que está armazenado naquele índice. A função ***each*** não funciona com arrays em versões anteriores à 5.12.

```
  1 #exemplo5.21
  2 my @array = (0..10);
  3 my %hash;
  4 while (my ($index, $valor) = each @array){
```

```
$hash{$index} = $valor };
  5 say "$_ => $hash{$_}" for sort {$a <=> $b} keys %hash;

saída:

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

NOTA: A partir da versão 5.12 as funções ***values*** e ***each*** também funcionam com arrays. O comando ***perl -E*** significa executar ***perl -e*** com ***use feature ":5.12"*** ativado.

```
$ perl -E'
my @arr = ('a'..'z');
say values @arr;
'
abcdefghijklmnopqrstuvwxyz
```

Exercícios

1. Faça um programa que gere um número randômico de 0 a 10 e peça repetidamente para o usuário tentar adivinhar o número mágico. Para cada valor informado pelo usuário o programa deve dar uma dica como "o número mágico é maior que o número informado" ou "o número secreto é menor que este".

2. Faça um script que informe o sistema operacional e a pasta raiz do usuário.

Capítulo 6. Funções Internas do Perl e Funções do Programador

Bom, nós já falamos um pouco de sub-rotinas ou funções. Vimos diversas funções internas ao Perl e algumas pertencentes a módulos que complementam as funcionalidades do Perl. Mas agora vamos entrar no detalhe deste assunto e nos tornar aptos a criar nossas próprias funções.

Uma função é um bloco de código reutilizável associado a um nome, declarado pelo operador ***sub e delimitado por chaves { }***. Como vimos, as chaves são delimitadores de código para ***loops*** e blocos ***if*** e criam o que os cientistas da computação chamam de **escopo**. Aliás, neste capítulo vamos estudar conceitos como **escopo de variáveis, contexto da função e argumentos**.

Mas vamos com calma. Primeiro vamos rever como declaramos funções:

6.1 Declarando funções

Reforçando, funções são declaradas pelo operador ***sub*** e delimitadas por ***chaves {}.***

```
sub nome_da_função {
     código_da_função;
     código_da_função;
     código_da_função;
} #fim da função.
```

Vamos agora criar uma função útil. O Perl possui internamente uma função que retorna a data e a hora local do sistema. Porém, esta função retorna um array com essas informações no seguinte formato:

```
0     1     2      3      4     5      6      7      8
$sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
localtime(time);
```

Para mostrarmos a hora real para o usuário no formato utilizado no Brasil precisamos criar um código que modele essas informações da forma que desejamos.

```
sub data_brasil {
     @arr_localtime = localtime ();
     $arr_localtime[5] += 1900;
     $arr_localtime[4]++;
     return "$arr_localtime[2]:$arr_localtime[1]:$arr_local-
time[0] " .
     "$arr_localtime[3]/$arr_localtime[4]/$arr_localti-
me[5]";
}
```

Com esta função em mão podemos criar um programa que mostra a data formatada para o usuário:

```
  1 #exemplo 6.1
  2 use warnings;
  3 use feature "say";
  4
  5 say "A função localtime retorna ", localtime,
  6  " que é um formato ruim para o usuário final!";
  7 say "Por isso criamos a função data_brasil!!!";
  8 say "Agora, no Brasil: ", data_brasil();
  9
 10 sub data_brasil {
 11         @arr_localtime = localtime ();
 12         $arr_localtime[5] += 1900;
 13         $arr_localtime[4]++;
 14         return "$arr_localtime[2]:$arr_
localtime[1]:$arr_localtime[0] " .
 15         "$arr_localtime[3]/$arr_localtime[4]/$arr_lo-
caltime[5]";
 16 }

saída:

A função localtime retorna 3114133511041530 que é um forma-
to ruim para o usuário final!
Por isso criamos a função data_brasil!!!
Agora, no Brasil: 13:14:31 3/3/2011
```

É fato que no Perl as funções podem ser declaradas onde bem entendermos, no início do arquivo, no fim do arquivo ou no meio... porém, por questão de organização, costumamos deixar as funções de usuário (funções que nós, usuários do Perl criamos) no fim do arquivo.

6.2 Argumentos

Argumentos são valores enviados à função pelo código que evoca a função. Por exemplo, a função interna ***sin*** recebe como argumento um valor em radianos. ***$seno_de_45 = sin 3.1415/4***.

A função ***data_brasil*** não possui argumentos. Ela apenas processa os dados da função ***localtime***. Mas suponha que eu não queira uma função que funcione apenas com o horário do momento. Eu quero uma função que formate qualquer array de data enviado. Assim, podemos formatar a data de qualquer tempo. Para isso vamos entender como o Perl trabalha com argumentos.

Os argumentos de uma função são armazenados na variável interna do Perl @_. Então quando enviarmos uma lista de argumentos para o Perl, eles serão armazenados em @_, independentemente do tipo de dado que sejam.

```
  1 #exemplo6.2
  2 use 5.12.0;
  3
  4 sub data_brasil2{
  5         $_[5] += 1900;
  6         return "$_[2]:$_[1]:$_[0] $_[3]/$_[4]/$_[5]";
  7 }
  8
  9 my @data = localtime;
 10 say data_brasil2(@data);

saída:

20:25:4 14/8/2011
```

Como discutimos anteriormente, para acessarmos os valores de um determinado índice do array usamos ***$nome_do_array[índice]***. Se o nosso array é @_, então

seus valores estão acessíveis em ***$_[0], $_[1], $_[2]*** e assim por diante. Para chamarmos uma função com argumentos, usamos:

```
função($argumento1, $argumento2, $argumento3);
```

Veja o exemplo:

```
  1 #exemplo6.3
  2 use warnings;
  3 use feature "say";
  4 use Math::Round "nearest";#Carrega o módulo Math::Round
  5
  6 printf "a média entre 5, 3 e 2 é: %.3f\n", me-
dia(5,3,2);
  7 printf "a média entre 1, 7, 9, 11 e 5 é: %.3f\n",
  8  nearest 0.001, media(1,7,9,11,5);
  9 printf "a média entre 6, 8, 11 e 120 é: %.3f\n", nea-
rest 0.001,
 10  media(6,8,11,120);
 11 printf "a média entre 1500, 1900, 1850 e 2001 é:
%.3f\n", nearest 0.001,
 12  media(1500,1900,1850,2001);
 10 say "também podemos enviar todo um array como argumen-
to:";
 12
 13 @minhas_notas = (5, 7.8, 10, 10, 3.2, 9);
 14 printf "A média é: %.2f\n", media(@minhas_notas);
 15
 16 sub media { #cria uma função chamada média
 17         $i = 0; #cria uma variável, $i para controlar o
loop
 18         $soma = 0; #$soma armazena o último valor da
doma dos elementos
 19         while ($i <= $#_){
 20                 $soma += $_[$i]; #realiza a soma
 21                 $i++
 22         }
```

```
 23         return $soma / ($#_ + 1) #retorna a média
 24 }

saída:

a média entre 5, 3 e 2 é: 3.333
a média entre 1, 7, 9, 11 e 5 é: 6.600
a média entre 6, 8, 11 e 120 é: 36.250
a média entre 1500, 1900, 1850 e 2001 é: 1812.750
também podemos enviar todo um array como argumento:
A média é: 7.50
```

Perceba que nada limita a quantidade de elementos do array @_ a não ser a própria capacidade do seu sistema. Os elementos de @_ não utilizados pela função são desprezados. Então se a chamada da função tiver mais elementos que os necessários, o Perl irá ignorá-los. Isso faz parte da filosofia do Perl: "sem limites desnecessários!". A nossa função utiliza tantos elementos quantos forem fornecidos nos argumentos. Isso graças à variável **$#_.** Sempre que criamos uma array, o Perl cria uma variável ***$#nome_do_array*** que armazena o **valor numérico do último índice do array (posição do último elemento válido)**. Como o valor do primeiro índice é sempre 0 (posição do primeiro elemento do array), então podemos manipular todos os valores de um array de tamanho desconhecido, entre 0 e o último índice em um loop ***while***. Perceba, no nosso exemplo, que utilizamos diversas vezes a função **media** sem termos que repetir o código dela. Esse é o princípio da reusabilidade de código. Quanto menos código repetido no nosso programa, mais fácil sua manutenção.

6.3 Valor de retorno

Todas as funções no Perl retornam um valor. Não que este valor possa ou deva ser utilizado por você sempre. Mas é assim que as funções funcionam no Perl. Elas sempre retornam um valor. Veja este exemplo:

```
  1 exemplo6.4
  2 use warnings;
  3 use feature "say";
  4 $v1 = say "Toda função retorna um valor";
  5 say "Veja: $v1";
```

```
saída:
Toda função retorna um valor
Veja: 1
```

A função interna do Perl ***say*** retorna o valor "1" quando sua execução ocorreu sem problemas. Mesmo sendo um valor que dificilmente seria aproveitado, toda função vai retornar um valor. Este valor é definido pelo operador ***return*** nas funções criadas pelo programador. Caso ***return*** seja omitido no código da função, esta retornará o valor da última expressão executada pelo bloco de código da função. Quando a execução da função chegar a um operador ***return*** a execução do código retorna para a função que evocou o bloco de código. No nosso exemplo, retorna para ***main***.

```
 1 #exemplo6.5
 2 use warnings;
 3 use feature "say";
 4
 5 do{
 6   print "Digite um número de 1 a 3 ou 0 para saír: ";
 7   chomp ($valor = <STDIN>);
 8   say "Você digitou ", opcoes($valor);
 9 }while($valor);
10
11   sub opcoes {
12           $opcao = $_[0];
13           if ($opcao == 0){
14                   exit #encerra o programa
15           }
16           if ($opcao == 1){
17                   return "Um" #retorna Um
18           }
19           if ($opcao == 2) {
20                   return "Dois" #retorna Dois
21           }
22           if ($opcao == 3) {
23                   return "Três" #retorna Três
24           }
```

```
 25
 26            say "Opção inválida";
 27            $opcao; #retorna o valor armazenado em $opcao
 28          }

saída:

Digite um número de 1 a 3 ou 0 para saír: 5
Opção inválida
Você digitou 5
Digite um número de 1 a 3 ou 0 para saír: 1
Você digitou Um
Digite um número de 1 a 3 ou 0 para saír: 2
Você digitou Dois
Digite um número de 1 a 3 ou 0 para saír: 3
Você digitou Três
Digite um número de 1 a 3 ou 0 para saír: 4
Opção inválida
Você digitou 4
Digite um número de 1 a 3 ou 0 para saír: 0
```

Este exemplo tem um operador novo, o ***do***. O ***do*** serve para executarmos um código dentro de um loop ***while*** antes de verificarmos se a condição de verificação é verdadeira ou falsa. Útil quando queremos que o código dentro do loop execute ao menos uma vez independentemente da condição de controle do loop.

O código é bem simples. O usuário deve digitar 1, 2 ou 3. Então o programa vai entrar no **loop** e chamar a função ***opcoes( ).*** A função irá verificar se o valor digitado pelo usuário está dentro da range definida e imprimir o número digitado por extenso. Se o valor digitado estiver fora da range o programa irá reclamar e retornar o número digitado. O importante aqui é você perceber que a função retorna o valor da primeira cláusula ***return*** válida que encontrar ou então irá retornar o valor da última expressão encontrada na função. No nosso exemplo, quando o usuário entrar uma opção inválida, a variável ***$opcao*** será o valor retornado, mesmo não havendo uma cláusula ***return.***

6.4 A função *main*

Main é a função principal de um programa escrito em linguagem C. No Perl você não precisa declarar uma função ***main***, mas ela simplesmente está aí. Esta função serve para organizar as variáveis léxicas do seu código e evocar as funções na ordem especificada pelo programador. Esta função principal utiliza o array ***@ARGV*** como lista de argumentos, diferentemente das outras funções que utilizam **@_.** Ops... variáveis léxicas... ainda não falamos disso.

6.5 O escopo das variáveis

Eu evitei no decorrer do livro atá agora utilizar variáveis léxicas. Isso quer dizer que a maioria das variáveis que utilizamos até o momento não têm escopo e podem ser acessadas em qualquer parte do programa.

```
  1  #exemplo6.6
  2  use warnings; #ativa warnings
  3  use feature "say";
  4
  5  say "Estou em main!";
  6  $str1 = "Esta str pode ser acessada por todas as
subs";
  7  my $str2 = "Esta str é Léxica"; #o operador my cria
variáveis locais
  8
  9  sub testa_escopo{
 10        say "Estou em testa_escopo!!!";
 11        say $str1;
 12        say "$str2";
 13        my $str3 = "main não me enxerga!!!";
 14        say $str3;
 15  }
 16
 17  say "Ainda estou em main!";
 18  testa_escopo();
 19  say "De volta em main!!!";
```

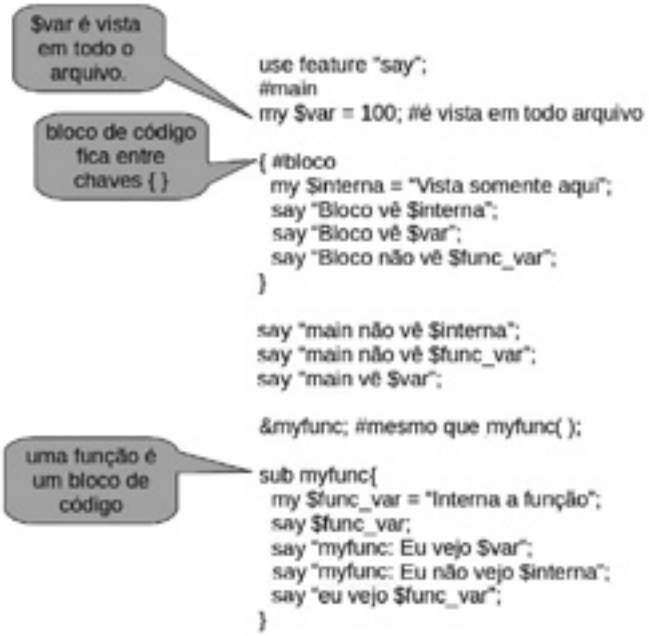
saída:

Name "main::str3" used only once: possible typo at -e line 20.

```
Estou em main!
Ainda estou em main!
Estou em testa_escopo!!!
Esta str pode ser acessada por todas as subs
Esta str é Léxica
main não me enxerga!!!
De volta em main!!!
Use of uninitialized value $str3 in concatenation (.) or
string at -e line 20.
$str3 não é acessível:
```

Perceba que a ***$str3*** de ***testa_escopo*** não está acessível em main. Uma variável criada pelo operador ***my*** está limitada pelos colchetes mais internos de onde ela for criada, seja em uma função, loop ou estrutura de decisão. Penso que a definição mais simples a que posso chegar para o escopo de uma variável seria: "é a parte do programa em que podemos usar o nome da variável para nos referirmos a ela".

Apesar de ser declarada com **my,** a variável ***$str2*** pode ser vista pela função testa_escopo. Isso ocorre porque as variáveis declaradas em ***main*** podem ser acessadas em qualquer ponto no arquivo, desde que não sejam declaradas dentro de um bloco de código. Veja este esquema:



Quanto ao escopo, uma variável pode ser definida como ***global, local ou dinâmica.*** Nossos programas até agora usaram variáveis em escopo global, isto é, qualquer parte do programa pode acessar a variável, inclusive outros arquivos de módulos. Isso não é um problema em programas pequenos com poucas dezenas de linhas, mas torna-se um problema em programas grandes, que envolvem diversos arquivos e, principalmente, diversos programadores, podendo causar conflitos de nomes ou alterar o funcionamento previsto do programa. Vamos voltar ao exemplo da média (exemplo 6.3) e mostrar como a não utilização de escopos pode causar efeitos indesejáveis:

```
  1 #exemplo6.7
  2 use warnings;
  3 use feature "say";
  4 use Math::Round "nearest";
  5
  6 @minhas_notas = (10, 9.7, 6.5, 8.3); #variável é global
  7 say "Minhas notas são: @minhas_notas";
  8 say "Logo minha média é: ", nearest (0.01, media(@mi-
nhas_notas));
  9 say "Porém a sub media mudou as minhas notas: @minhas_
notas";
 10 say "Veja que tenho acesso à variável \$i: $i";
 11 say "O que pode causar muito problema em programas lon-
gos";
 12
 13 sub media {
 14         $i = 0;
 15         $media = 0;
 16    while ($i <= $#minhas_notas){ #A variável é global,
podendo ser acessada
 17                                  #de qualquer lugar
dentro do programa.
 18                                  #inclusive outros ar-
quivos
 19        $minhas_notas[$i] /= ($#minhas_notas + 1);
 20         $media += $minhas_notas[$i];
 21         $i++;
 22   }
```

```
 23   return $media;
 24 }

saída:

Minhas notas são: 10 9.7 6.5 8.3
Logo minha média é: 8.63
Porém a sub media mudou as minhas notas: 2.5 2.425 1.625
2.075
Veja que tenho acesso à variável $i: 4
O que pode causar muito problema em programas longos
```

Perceba: uma mudança na lógica do programa pode afetar profundamente todo o funcionamento se as variáveis não estiverem separadas, devidamente declaradas em seu próprio espaço ou se utilizarmos variáveis de ***main*** diretamente nas funções, ao invés de passarmos argumentos e recebermos valores de retorno.

```
  1 #exemplo6.8
  2 use warnings;
  3 use feature "say";
  4
  5 say "Veja uma sub sem variáveis globais:";
  6 sub1();
  7 our $var1 = 10;
  8 our $var2 = 25;
  9 say "\$var1 e \$var2 são variaveis globais: ";
 10 say "com o mesmo nome que as variáveis de sub1";
 11 say "$var1\t$var2\n\n";
 12 sub1();
 13 say "\n\$var1 global: $var1";
 14 say "\n\$var2 global: $var2";
 15
 16 sub sub1{
 17   my $var1 = 100;
 18   local $var2 = 50;
 19   say "Estou em sub1!!!";
 20   say "Aqui \$var1 é privada à sub1: $var1\té léxi-
```

```
ca!!";
 21   say "Aqui \$var2 é declarada como local à sub1:
$var2\té dinâmica!!!";
 22  say "Então, se eu evocar sub2 daqui, \$var2 estará
acessível";
 23   say "porem esta \$var2 não está visível em main\n\n";
 24   sub2();
 25 }
 26
 27 sub sub2{
 28   say "Estou em sub2!!!";
 29   say "Eu vejo a \$var1 de main: $var1\t(global)";
 30   say "Porém eu vejo a \$var2 de sub1:$var2\
t(dinâmica)";
 31 }
```

saída:

```
Veja uma sub sem variaveis globais:
Estou em sub1!!!
Aqui $var1 é privada à sub1: 100   é léxica!!
Aqui $var2 é declarada como local à sub1: 50   é dinâmi-
ca!!!
Então, se eu evocar sub2 daqui, $var2 estará acessível
porem esta $var2 não está visível em main


Estou em sub2!!!
Eu vejo a $var1 de main:     (global)
Porém eu vejo a $var2 de sub1:50   (dinâmica)
$var1 e $var2 são variaveis globais:
com o mesmo nome que as variáveis de sub1
10   25


Estou em sub1!!!
Aqui $var1 é privada à sub1: 100   é léxica!!
Aqui $var2 é declarada como local à sub1: 50   é dinâmi-
```

```
ca!!!
Então, se eu evocar sub2 daqui, $var2 estará acessível
porem esta $var2 não está visível em main


Estou em sub2!!!
Eu vejo a $var1 de main: 10  (global)
Porém eu vejo a $var2 de sub1:50   (dinâmica)

$var1 global: 10

$var2 global: 25
```

Então, uma variável léxica – criada com ***my*** – é válida apenas dentro do bloco onde foi criada. Uma variável dinâmica – criada com o operador ***local*** – é válida durante a execução do bloco onde foi criada, incluindo chamadas a outras funções e blocos mais internos. E as variáveis globais – criadas com ***our*** – são válidas em todo o programa.

```
  1 #exemplo6.9
  2 use warnings;
  3 use feature "say";
  4
  5 my $var1 = 100;
  6 my $var2 = 500;
  7 if ($var1<1000){
  8         my $var2 = 1000;
  9         say "\$var2 dentro deste bloco é: $var2";
 10         say "Porém eu posso ver \$var1: $var1"
 11 }
 12 say $var2;

saída:

$var2 dentro deste bloco é: 1000
Porém eu posso ver $var1: 100
500
```

Veja neste exemplo que o escopo de variáveis se aplica aos blocos de controle, como ***if, else, for, while*** e outros.

6.6 Recursividade

Existem coisas que podem ser estranhas à primeira vista em programação. Uma delas é a recursividade. Recursividade nada mais é que uma função evocar a si própria no código. Embora estranho, isso é uma forma de implementar um loop. A função executa uma tarefa e chama a si própria, dependendo da função que deve executar.

```
 1 #exemplo6.10
 2 use warnings;
 3 use strict;
 4 use feature "say";
 5
 6 my $i = 0;
 7 while($i<=10){
 8         say "$i! = " . fatorial($i);
 9         $i++;
10 }
11
12 sub fatorial{
13         my $valor = $_[0];
14         if($valor <=1){
15                 return 1
16         }
17         return $valor * fatorial ($valor - 1)
18 }
```

saída:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
```

```
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

A função ***fatorial*** chama a si própria e o argumento quando evoca é ***$i*** quando chamada por ***main*** e ***$valor – 1*** quando chamada pela própria função. Todo algoritmo que pode ser feito de forma recursiva pode ser feito de forma iterativa e vice-versa.

```
 1 #exemplo6.11
 2 use warnings;
 3 use feature "say";
 4 use strict;
 5
 6 say "Versão para cálculo de fatorial por iteração";
 7 my $i = 0;
 8 while ($i <= 10){
 9         if ($i <= 1){
10                 say "$i! = 1";
11         }
12         if ($i == 2){
13                 say "$i! = 2";
14         }
15         my $resultado = $i;
16         my $j = $i;
17         my $k = 2;
18         while ($j > 2){
19                 $k++;
20                 $resultado = $resultado *  ($k - 1);
21                 $j--;
22         }
23         if ($i>2){
24                 say "$i! = $resultado"
25         }
26         $i++
27 }
```

O que irá definir se você deverá utilizar iteratividade ou recursividade será o projeto. Se o projeto exigir desempenho, prefira iteratividade; se você preferir a legibilidade do código, a recursividade pode ser uma escolha melhor.

NOTA: O pragma ***use warnings*** provê informações úteis de debug e alertas ao programador. Você pode verificar todas as informações sobre este pragma online na documentação do seu sistema com o comando ***perldoc warning.*** Já o pragma ***use strict*** irá garantir que você utilize algumas boas práticas de programação, como declarar variáveis e não utilizar "barewords" como chamada a funções. Mais sobre ***strict*** em ***perldoc strict***. O Perl 5.12 possui um pragma otimizado: ***use 5.12.0.*** Este pragma substitui ***use strict, use warnings e use feature.*** Usaremos este pragma daqui em diante.

6.7 Funções internas do Perl

Perl é uma linguagem com muitos verbos. Tal qual o português. E mesmo nós, nativos da língua lusa, ainda precisamos de um dicionário para certas palavras, imagine quando aprendemos um idioma estrangeiro. No nosso caso o idioma estrangeiro é Perl, e os dicionários são o **perldoc** e o **perlfunc**. O **perlfunc** é um documento com cerca de cem páginas que explica com exemplos todas as funções do Perl. O documento mais atual, em inglês, está no link http://perldoc.perl.org/perlfunc.html. Caso você tenha dificuldade com o inglês, existe uma versão menos atual, mas ainda assim muito útil em http://www.perl.org.br/Perldoc/V500807/Perlfunc. Você também pode acessar o ***perlfunc*** da sua versão Perl com o comando ***perldoc perlfunc.***

6.7.1 Funções Perl por categoria

Escalares

```
chomp, chop, chr, crypt, hex, index, lc, lcfirst, length,
oct, ord, pack, q//, qq//, reverse, rindex, sprintf, subs-
tr, tr///, uc, ucfirst, y///
```

Expressões regulares

```
m//, pos, qr//, quotemeta, s///, split, study
```

Números

abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand

Arrays e hashes

pop, push, shift, splice, unshift
grep, join, map, qw//, reverse, sort, unpack
delete, each, exists, keys, values

Entrada, saída e filehandles

binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, readpipe, rewinddir, seek, seekdir, select (ready file descriptors), syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write
chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, select (ready file descriptors), select (output filehandle), stat, symlink, sysopen, umask, unlink, utime

Dados e estruturas de tamanho predefinido

pack, read, syscall, sysread, sysseek, syswrite, unpack, vec

Controle do fluxo do programa

caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray

Escopo

caller, import, local, my, no, our, package, use

Diversas

defined, dump, eval, formline, lock, prototype, reset, scalar, undef, wantarray

Administração e criação de processos

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid

Funções para módulos

do, import, no, package, require, use

OOP

bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use

Programação de rede de baixo nível

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

Comunicação entre processos

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

Informações de usuários e grupos

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

Informações de rede em baixo nível

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

Hora e data

gmtime, localtime, time, times

6.7.2 Como trabalhar com as funções do Perl

Primeiro, dialeticamente, as **funções** do Perl são chamadas de operadores em muitas literaturas. Eu prefiro usar o termo funções pois muitos programadores Perl começaram em outras linguagens, e mantemos um termo comum com o C e outras linguagens amplamente utilizadas. Em outras linguagens existe a forte distinção entre **operadores** e **funções**. Em seu livro "Programming Perl", Larry Wall descreve funções internas da linguagem Perl como "**prefix-operators**", uma vez que elas realizam processamento com os argumentos que se encontram à direita do operador.

Outra regra importante sobre as funções internas do Perl: elas não precisam de parênteses para os argumentos. Veja este exemplo (que você também vai encontrar no perlfunc):

```
1 #exemplo6.12
2 no warning;
3 print 1+2*4;        # 9.
4 print(1+2) * 4;     # 3
5 print (1+2)*4;      # 3
6 print +(1+2)*4;     # 12
7 print ((1+2)*4);    # 12
```

Os programadores Perl não têm o hábito de colocar os argumentos das funções internas entre parênteses, a não ser que facilite o entendimento do que deve ser feito para o programador que fará manutenção futura do código. O exemplo 6.12 não funciona com o pragma **use warning** ativado.

Em contrapartida, os argumentos das funções definidas pelo programador devem estar entre parênteses, a não ser que o programador declare a função ou o protótipo da função antes de utilizá-la.

6.8 Técnicas para trabalhar com argumentos em funções

Muitos programadores não gostam da ideia de não poder especificar quais variáveis serão passadas para sua função, seu tipo etc. Mas, na verdade, podem. **O Perl apenas não obriga a isto**. Faz parte da sua filosofia: "não forçar limites desnecessários". Para criar este tipo de controle em Perl podemos usar a técnica dos **argumentos nomeados**. Esta técnica consiste em passar um hash como argumento.

Assim, as chaves do hash funcionam como nomes dos argumentos e o valor da chave como a variável enviada ao processamento:

```
 1 #exemplo6.13
 2 use 5.12.0;
 3 use Math::Trig qw/pi/;
 4
 5 area(figura => 'circulo', raio => 5);
 6 area(figura => 'retangulo', base => 5, altura => 10);
 7 area(figura => 'quadrado', lado => 5);
 8 area(figura => 'triangulo', base => 5, altura => 10);
 9
10 sub area{
11         my %args = @_;
12         die "Lista de argumentos inválida"
13                 if not (defined $args{figura});
14         given($args{figura}){
15                 when($_ eq 'circulo'){ say pi *
$args{raio}**2}
16                 when($_ eq 'retangulo'){ say
$args{base} * $args{altura}}
17                 when($_ eq 'quadrado') {say
$args{lado}**2}
18                 default {say "figura inválida"}
19         }
20 }
```

saída:

```
78.5398163397448
50
25
figura inválida
```

6.9 Passando referências como argumentos

Como falamos pouco sobre referências, cabem aqui mais algumas palavrinhas. Vejamos... para criar uma referência nós precisamos criar um escalar e atribuirmos um array, hash ou outro escalar ao primeiro com o operador ****. Para acessar seus valores, usamos a variável escalar que recebeu a referência e o operador **->** ou **@$** para um array_ref, **%$** para um hash_ref e **$$** para uma referência a um escalar, como no exemplo 6.14:

```
  1 #exemplo6.14
  2 use 5.12.0;
  3
  4 my @array = ('a'..'z');
  5 my $array_ref = \@array;
  6 say @$array_ref; #acessa todos os elementos de @array;
  7 say $array_ref->[5]; #acessa apenas o índice 5;
  8 say $array_ref->[17]; #acessa apenas o índice 17
  9 my %hash = (host1 => '192.168.5.4',
 10         host2 => '192.168.5.5',
 11         gateway => '192.168.5.1');
 12 my $hash_ref = \%hash;
 13 say %$hash_ref; #imprime todo o hash
 14 say $hash_ref->{gateway}; #imprime apenas o valor de
$hash{gateway}

saída:

abcdefghijklmnopqrstuvwxyz
f
r
gateway192.168.5.1host2192.168.5.5host1192.168.5.4
192.168.5.1
```

Então, se quisermos passar diversos arrays e hashes podemos "encapsular" nossos arrays e hashes dentro de escalares e enviar para nossas funções:

```
  1 #exemplo6.15
  2 use 5.12.0;
  3
```

```
  4 my @arr1 = (0 .. 100);
  5 my @arr2 = (1 .. 10);
  6 pass_by_ref(\@arr1, \@arr2);
  7
  8 sub pass_by_ref {
  9   #$somar e $multiplicar são referências aos arrays @
arr1 e @arr2.
 10   my ($somar, $multiplicar) = @_;
 11   my $soma = 0;
 12   $soma += $_ for @$somar; #passa os valores do array @
arr1 para o loop
 13   say "A soma dos elementos de $somar é: $soma";
 14
 15   my $multiplicacao = 1;
 16   $multiplicacao *= $_ for @$multiplicar;#passa os va-
lores do array @arr2
 17   say "A multiplicação dos elementos de $multiplicar é:
$multiplicacao    "
 18 }
```

saída:

A soma de ARRAY(0x12f0dcc) é: 5050
A multiplicação dos elementos de ARRAY(0x12fdd84) é: 3628800

```
  1 #exemplo6.16
  2 use 5.12.0;
  3 use Net::Ping;
  4 #realiza teste de keep-alive na porta 80.
  5
  6 my @hosts = qw/uol.com.br ticursos.net sao-paulo.
pm.org/;
  7
  8 is_alive(hosts => \@hosts, port => 80); #função que de-
finimos abaixo.
  9
 10 #Utiliza como argumentos: hosts que recebem uma refe-
rência de um array
```

```
 11 #e port que recebe um escalar. Assim não importa a or-
dem dos elementos.
 12
 13 sub is_alive{
 14   my %args = @_;
 15
 16   foreach (@{$args{hosts}}){
 17         my $host = $_;
 18         my $p = Net::Ping->new(); #Cria um objeto
Net::Ping
 19         $p->port_number($args{port}); #define a porta a
ser pingada
 20         $p->hires();    #ativa contador de tempo em ms.
 21         my ($ret, $duration, $ip) = $p->ping($host,
5.5); #pinga o elemento
 22
 23         printf("$host [ip: $ip] is alive (packet return
time: %.2f ms)\n",
 24                 1000 * $duration) if $ret;
 25         $p->close();
 26    }
 27 }
```

saída:

```
uol.com.br [ip: 200.221.2.45] is alive (packet return time:
12.69 ms)
ticursos.net [ip: 174.120.239.219] is alive (packet return
time: 170.93 ms)
sao-paulo.pm.org [ip: 64.150.190.83] is alive (packet re-
turn time: 208.90 ms)
```

Com a utilização da técnica **"named arguments"** e utilização de referências, temos total controle dos valores passados para minha função, sem me preocupar com algoritmos doidos para contar os elementos do array ou utilizar slices do array @_.

Se tiver necessidade de um controle maior dos argumentos da sua função, utilize o módulo CPAN **Params::Util**, que fornece verificação de tipo mais profunda, como _ARRAY, _ARRAYLIKE, _STRING, _POSINT etc.

```
  1 #exemplo6.17
  2 use 5.12.0;
  3 use Params::Util qw/:ALL/;
  4 use Net::Ping;
  5
  6 my @hosts = qw(uol.com.br terra.com.br yahoo.com.br
google.com.br
  7                ticursos.net cpan.org freebsd.org ftp.
freebsd.org);
  8 my %ports = ( http => 80,
  9                           ftp => 21,
 10                           );
 11
 12 check_port(hosts => \@hosts, ports => \%ports);
 13
 14 sub check_port{
 15         my %args = @_;
 16
 17     die "Uso dos Argumentos hosts => ARRAY_REF, ports
=> HASH_REF"
 18         unless (_ARRAYLIKE($args{hosts}) and _
HASHLIKE($args{ports}));
 19
 20 #Verificar se $args{host} é uma referência para um ar-
ray
 21 #e se $args{ports} é uma referência para um hash.
 22     for my $host (@{$args{hosts}}){
 23         my $ping = Net::Ping->new();
 24
 25         for my $port (keys %{$args{ports}}){
 26             $ping->port_number($args{ports}->{$port});
 27             $ping->hires();
 28             my ($ret, $duration, $ip) = $ping-
>ping($host, 5.5);
 29
 30            printf("$host [ip: $ip] esta up e porta do
serviço $port está" .
 31                   "escutando (packet return time: %.2f
```

```
ms)\n",
 32                    1000 * $duration) if $ret;
 33         }
 34     $ping->close();
 35    }
 36 }

saída:

uol.com.br [ip: 200.221.2.45] esta up e porta do serviço
http está escutando (packet return time: 9.89 ms)
terra.com.br [ip: 200.154.56.80] esta up e porta do serviço
http está escutando (packet return time: 32.06 ms)
yahoo.com.br [ip: 69.147.76.254] esta up e porta do serviço
http está escutando (packet return time: 128.77 ms)
google.com.br [ip: 64.233.163.104] esta up e porta do ser-
viço http está escutando (packet return time: 11.30 ms)
ticursos.net [ip: 174.120.239.219] esta up e porta do ser-
viço ftp está escutando (packet return time: 233.63 ms)
ticursos.net [ip: 174.120.239.219] esta up e porta do ser-
viço http está escutando (packet return time: 235.70 ms)
cpan.org [ip: 64.235.248.110] esta up e porta do serviço
http está escutando (packet return time: 228.70 ms)
freebsd.org [ip: 69.147.83.40] esta up e porta do serviço
http está escutando (packet return time: 200.46 ms)
ftp.freebsd.org [ip: 204.152.184.73] esta up e porta do
serviço ftp está escutando (packet return time: 211.09 ms)
ftp.freebsd.org [ip: 204.152.184.73] esta up e porta do
serviço http está escutando (packet return time: 220.98 ms)
```

Nosso programa utiliza o módulo CPAN ***Net::Ping***. A nossa função ***check_port*** recebe uma referência para um array com uma lista de IPs ou nomes válidos na rede no argumento ***hosts*** e uma referência a um hash com nome e número da porta TCP no argumento ***ports***. Caso algum argumento não atenda à verificação, o programa encerra com a mensagem de erro "Uso dos Argumentos hosts => ARRAY_REF, ports => HASH_REF".

Se você pretende criar módulos que serão utilizados por terceiros, as funções de verificações dos módulo ***Param::Util*** podem diminuir bastante o trabalho de checagem de dados.

6.10. Protótipos

Alguns programadores podem preferir protótipos que permitem declarar e verificar o tipo de dados passados a uma função em tempo de compilação. A prototipagem padrão Perl utiliza o seguinte formato:

```
sub minha_sub($$$@){
     código.
}
```

Isso significa que nossa função recebe três escalares e uma lista como argumentos. Nesse caso, a lista (array ou hash) deve ser o último elemento do protótipo, pelos mesmos motivos que já explicamos no início do capítulo. Se quiser criar um protótipo de uma função que receba também uma refrência de uma lista, no caso um hash:

```
sub minha_sub($$$\%){
     código.
}
```

Se desejar ampliar a funcionalidade dos protótipos, pode utilizar o módulo ***Arguments*** do CPAN.

6.11 Retornando valores com wantarray

As funções dos nossos exemplos até agora executaram tarefas, mas não retornaram valores úteis. Pelo menos não formalmente. **Toda função em Perl retorna um valor. Depende do programador que utiliza a função aproveitar ou não este valor, e do desenvolvedor da função criar ou não um valor de retorno útil.**

Se nenhum valor de retorno for especificado pelo programador, o Perl retornará o valor da última expressão executada dentro da função. Para especificar um valor de retorno podemos utilizar a função ***return***. A partir do momento em que a função ***return*** é executada, o Perl retorna à função que chamou a sua função, também conhecida como ***caller***. Assim, ***return*** também é uma função de controle de fluxo.

```
 1 #exemplo6.18
 2 use 5.12.0;
 3
 4 my %info = get_user_info();
 5 map {say "$_ => $info{$_}"} keys %info;
 6
 7 sub get_user_info{
 8         my %user_info;
 9         for (keys %ENV){
10                 if ($_ eq "USER" or $_ eq "USERNAME"){
11                         $user_info{User} = $ENV{$_};
12                 }
13                 elsif ($_ eq 'PATH'){
14                         $user_info{Path} = $ENV{$_};
15                 }
16                 elsif ( /\bHOME\b|HOMEPATH/){
17                         $user_info{Home} = $ENV{$_};
18                 }
19                 else {next}
20         }
21         return %user_info
22 }
```

saída:

```
Home => \Users\thiago
Path => H:/Java/bin/client;H:/Java/bin;C:\Windows\
system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\
System32\WindowsPowerShell\v1.0\;C:\Program Files\MySQL\
MySQL Server 5.1\bin;C:\strawberry5.12\c\bin;C:\straw-
berry5.12\perl\site\bin;C:\strawberry5.12\perl\bin;h:\Par-
rot-2.5.0\bin;H:\Java\bin;C:\Program Files\Gtk+\bin
User => thiago
```

Podemos deixar a nossa sub sensível ao contexto com a função Perl ***wantarray.*** Esta função verifica se o ***caller*** espera receber como retorno uma lista ***(array ou hash)*** ou um escalar.

```
 1 #exemplo6.19
 2 use 5.12.0;
```

```
  3
  4 my %info = get_user_info(); #em contexto de lista re-
torna um hash
  5 map {say "$_ => $info{$_}"} keys %info;
  6 my $user = get_user_info(); #em contexto escalar retor-
na o nome do usuário
  7 say $user;
  8
  9 sub get_user_info{
 10         my %user_info;
 11         for (keys %ENV){
 12                 if ($_ eq "USER" or $_ eq "USERNAME"){
 13                         $user_info{User} = $ENV{$_};
 14                 }
 15                 elsif ($_ eq 'PATH'){
 16                         $user_info{Path} = $ENV{$_};
 17                 }
 18                 elsif ( /\bHOME\b|HOMEPATH/){
 19                         $user_info{Home} = $ENV{$_};
 20                 }
 21                 else {next}
 22         }
 23         #verifica se o caller é uma lista ou escalar.
 24         return wantarray ? %user_info : $user_
info{User};
 25 }
```

saída:

```
Home => \Users\thiago
Path => H:/Java/bin/client;H:/Java/bin;C:\Windows\
system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\
System32\WindowsPowerShell\v1.0\;C:\Program Files\MySQL\
MySQL Server 5.1\bin;C:\strawberry5.12\c\bin;C:\straw-
berry5.12\perl\site\bin;C:\strawberry5.12\perl\bin;h:\Par-
rot-2.5.0\bin;H:\Java\bin;C:\Program Files\Gtk+\bin
User => thiago
thiago
```

Outro contexto importante do Perl é o **void context,** que ocorre se nossa função não é associada a nenhum contexto. Por exemplo, sempre que chamamos a fun-

ção ***say*** em nossos exemplos, ela está no contexto **void,** uma vez que seu valor de retorno não é associado a nenhuma variável. Ou seja, não está associada a um escalar, nem a um array, nem a um contexto booleano. Então mudamos o final do nosso exemplo:

```
22   }
23   #verifica se o caller é uma lista, escalar ou void.
24   die "Função não pode ser usada em void context" unless
defined wantarray;
25   return wantarray ? %user_info : $user_info{User};
26 }
```

NOTA: Quando em void context a função wantarray retorna undef.

Exercícios

1. Elabore uma função que calcule o montante final de um investimento. A função deve receber três argumentos: o montante inicial, o rendimento mensal e o tempo de investimento, em meses.

2. Crie uma função repetir que recebe como argumentos uma string **s** e um inteiro **n**. Quando estiver em contexto, escalar a função deve imprimir a string **s** por **n** vezes. Quando estiver em um contexto de lista, deve retornar um array contendo **n** elementos. Cada elemento dever se a própria string **s.**

3. Refaça os exercícios anteriores utilizando a técnica de argumentos nomeados.

4. Faça um programa simples de folha de pagamento. O usuário informa o nome do funcionário, as horas trabalhadas no mês e o valor da hora. Se o número de horas trabalhadas for maior que 144, o programa deve calcular os excedentes como horas extras, adicionando 30% ao valor da hora informado.

5. Monte uma função que retorne verdadeiro se um ano informado como argumento é um ano bissexto e retorne falso se não. Anos bissextos são divisíveis por quatro, exceto os que forem divisíveis por cem. Anos divisíveis por cem e por quatrocentos também são bissextos. Exemplo: o ano 1800 é divisível por 4 e por 100, mas não por 400, portanto não é bissexto. O ano 2004 é divisível por 4, mas não por 100, portanto é bissexto. O ano 2000 é divisível por 4, por 100 e por 400, portanto é bissexto. Já 2001 não é divisível por 4, portanto não é bissexto.

Capítulo 7. Variáveis de Lista: Arrays e Hashes

7.1 Introdução a escolha de dados

Existem em programação diversos tipos de arranjos de dados. Para entender essa variedade de arranjos (arrays, hashes, referências) vamos fazer uma comparação com os diversos tipos de calendários.

Todos os calendários fazem, basicamente, a mesma coisa: armazenam e mostram datas. Alguns calendários mostram datas com uma semana por folha. Agendas mostram as datas com um ou dois dias em cada folha. Calendários em talões de cheque mostram datas do ano todo em uma folha. Por que maneiras diferentes de mostrar datas? Cada tipo de calendário é interessante para uma atividade específica. Você não pretende colocar, por exemplo, todas as suas reuniões no calendário do seu talão (a não ser que sejam muito poucas). Nem colocar as datas de depósito de cheques no calendário que enfeita a parede da sua sala. Assim, os diversos tipos de dados e arranjos do Perl podem e devem ser estudados para cada aplicação, a fim de encontrar o melhor modelo que atenda à sua necessidade.

7.2 A função qw

A função ***qw*** é mais uma de nossas ferramentas para preguiçosos. Quando declaramos um array de strings utilizamos a sintaxe: ***@array = ('string1', 'string2', 'string3'...)***. Até o momento em que um programador preguiçoso e cansado de tanto digitar aspas e vírgulas criou a função qw – quote words. Essa função retorna um array como se todas as palavras digitadas estivessem separadas por vírgula e entre apóstrofos. A função ***qw*** pode utilizar qualquer caractere para definir o início e o fim do array.

```
1 #exemplo7.1
2 use 5.12.0;
3
4 my @array1 = qw#casa carro ap sitio#; #o caractere # de-
fine o limite do array
5 my @array2 = qw{avião helicóptero jato};
6 my %hash = qw(host1 10.0.0.100 host2 10.0.0.101 host3
10.0.0.102);
7 map{say "$_ => $hash{$_}"} keys %hash;
8 say "@array1\n@array2";

saída:

host3 => 10.0.0.102
host2 => 10.0.0.101
host1 => 10.0.0.100
casa carro ap sitio
avião helicóptero jato
```

7.3 Manuseando um array

Os arrays em Perl têm seus recursos alocados dinamicamente. Assim, ao criar um array não precisamos saber quantos e quais índices serão utilizados. Isso permite grande flexibilidade. A variável **$#nome_do_array** armazena o índice do último elemento do array. Assim, você pode rapidamente aumentar ou diminuir o tamanho de um array modificando o valor da variável. Se você diminuir o valor de ***$#nome_do_array*** o seu array será truncado e os valores dos índices serão perdidos. Aumentando o seu valor será atribuído o valor ***undef*** aos índices adicionados.

```
1 #exemplo7.2
2 use 5.12.0;
3
4 my @array1 = qw#casa carro ap sitio#;
5 $#array1 = 1; #ap e sítio são eliminados
6 say "@array1";
7 $#array1 = 3; #2 novos índices com valor undef.
```

```
  8 for (@array1){
  9         say "Valor undef no índice" and next if not de-
fined $_;
 10         say $_
 11 }

saída:

casa carro
casa
carro
Valor undef no índice
Valor undef no índice
```

Além desta facilidade, o Perl oferece as funções ***push***, ***pop, shift*** e ***unshift***. As funções ***push*** e ***pop*** adicionam ou removem elementos ao fim do array respectivamente e alteram o valor de ***$#nome_do_array.*** Já as funções ***unshift*** e ***shift*** inserem e removem elementos no início do array, também modificando a variável ***$#nome_do_array***.

```
 1 #exemplo7.3
 2 use 5.12.0;
 3
 4 my @array = (0..10);
 5 push @array, 11; #podemos adicionar um escalar
 6 push @array, (12..15); #ou toda uma lista ao final do
array
 7 say "@array";
 8 say pop @array; #remove o último elemento de um array e
passa para o caller.
 9 say "@array"; #array não tem mais o elemento 15
10
11 #podemos adicionar um escalar ou toda uma lista ao iní-
cio do array
12 unshift @array, (-5..-1);
13 say "@array";
14
```

```
15 #remove o primeiro elemento de um array e o entrega para
o caller.
16 say shift @array for 1..3;
17 say "@array";

saída:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
-5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
-5
-4
-3
-2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

A outra função que permite que você manuseie os arrays de forma muito prática é a ***splice***. Esta função tem o seguinte formato: ***splice @array, [offset], [length], [@list]***. Ela trabalha removendo os elementos a partir de offset até offset + length e substitui pelos elementos de @list. Caso você não informe @list, os elementos serão apenas removidos.

```
  1 #exemplo7.4
  2 use 5.12.0;
  3
  4 my @array = ('a'..'z');
  5 say scalar @array;
  6 say @array;
  7 say "Meu Array é: @array";
  8
  9 splice @array, $#array; #mesmo que pop @array, remove o
último elemento
 10 say "pop: @array";
 11
 12 splice @array, 0, 0, 'z'; #mesmo que unshift @array,
'z'. Remove o primeiro
 13                          #elemento e substituí por 'z'
```

```
 14 say "unshift: @array";
 15
 16 splice @array, 0, 1; #mesmo que shift @array
 17 say  "shift: @array";
 18
 19 splice @array, $#array,$#array , 'z'; #mesmo que push @
array, 'z'
 20 say "push: @array";
 21
 22 my @removed = splice @array, 10, 5; #remove 5 elemen-
tos, iniciando pelo
 23                                      #índice 10
 24 say "remove uma parte: @array";
 25 say "parte removida: @removed";
 26
 27 splice @array, 10,0, @removed; #Adiciona o array @remo-
ved na posição 10
 28 say "adiciona uma parte: @array";
 29
 30 #substitui os elementos da posição 10 a 14 por K L M N O
 31 my @maiusculo = ('K'..'O');
 32 my @substituido = splice @array, 10, scalar @maiusculo,
@maiusculo;
 33 say "Substitui uma parte: @array";
 34 say "Parte substituída: @substituido";
```

saída:

```
26
abcdefghijklmnopqrstuvwxyz
Meu Array é: a b c d e f g h i j k l m n o p q r s t u v w
x y z
pop: a b c d e f g h i j k l m n o p q r s t u v w x y
unshift: z a b c d e f g h i j k l m n o p q r s t u v w x
y
shift: a b c d e f g h i j k l m n o p q r s t u v w x y
push: a b c d e f g h i j k l m n o p q r s t u v w x z
remove uma parte: a b c d e f g h i j p q r s t u v w x z
```

```
parte removida: k l m n o
adiciona uma parte: a b c d e f g h i j k l m n o p q r s t
u v w x z
Substitui uma parte: a b c d e f g h i j K L M N O p q r s
t u v w x z
Parte substituída: k l m n o
```

Além destas funções temos os ***slices*** (pedaços) de ***arrays*** como mais uma ferramenta de manuseio. Posso utilizar um grupo de índices ao mesmo tempo e processar estes elementos em grupo, independentemente do resto do ***array***:

```
 1 #exemplo7.5
 2 use 5.12.0;
 3
 4 my @array = ('A'..'Z');
 5 say "Meu array é: @array";
 6 #Este slice tem os índices do 15 ao 17 o 19 e o 22
 7 my @slice1 = @array[15..17,19,22];
 8 say "Meu pedacinho: @slice1";
 9 #Aqui eu substituí os valores dos índices de 2 a 7
10 @array[2..7] = (5..10);
11 say "Pedaço modificado: @array";
```

saída:

```
Meu array é: A B C D E F G H I J K L M N O P Q R S T U V W
X Y Z
Meu pedacinho: P Q R T W
Pedaço modificado: A B 5 6 7 8 9 10 I J K L M N O P Q R S T
U V W X Y Z
```

7.3.1 Ordenando o array

Os dados de um array nem sempre são fornecidos na ordem desejada ou necessária. É interessante para execução de algoritmos de pesquisa, ou para apresentar saída de dados para o usuário, que os elementos do array estejam em alguma ordenação, como do maior para o menor valor ou vice-versa. As funções internas do Perl ***sort*** e ***reverse*** podem resolver muitos destes problemas.

```
  1 #exemplo7.6
  2 use 5.12.0;
  3
  4 #A função sort recebe um bloco de código ou uma função
e um array como
  5 #argumentos. Os elementos do array são extraídos e en-
viados para as
  6 # variáveis  internas $a e $b, que são passadas como
argumentos para o
  7 # bloco de código. O bloco deve informar -1, 0 ou 1
para cada par enviado.
  8
  9 my @baguncado = qw/10 0 5 27 32 829 567 58 3 56 9 17/;
#crio o array
 10 my @ordem_numerica = sort { $a <=> $b } @baguncado;
#ordem numérica
 11 my @ordem_ascii = sort { $a cmp $b } @baguncado; #orde-
no alfabética com cmp
 12 my @inverso = reverse @ordem_numerica; #ordem reversa
 13
 14 say "Em ordem numérica: @ordem_numerica";
 15 say "Invertendo: @inverso";
 16 say "Na ordem da tabela ASCII: @ordem_ascii";
 17
 18 my @utilidades = qw/casa Carro ap Piscina escola Sítio
quadra/;
 19 my @utilidades_ordem_ascii = sort { $a cmp $b} @utili-
dades;
 20 my @utilidades_ordem_alfabetica = sort { uc($a) cmp
uc($b) } @utilidades;
 21 my @utilidades_ordem_decres = sort{ uc($b) cmp uc($a) }
@utilidades;
 22
 23 say "Meus utensílios em ordem ascii: @utilidades_or-
dem_ascii";
 24 say "Meus utensílios em ordem alfabetica: @utilidades_
ordem_alfabetica";
 25 say "Meus utensílios em ordem decrescente: @utilidades_
ordem_decres";
```

```
saída:

Em ordem numérica: 0 3 5 9 10 17 27 32 56 58 567 829
Invertendo: 829 567 58 56 32 27 17 10 9 5 3 0
Na ordem da tabela ASCII: 0 10 17 27 3 32 5 56 567 58 829 9
Meus utensílios em ordem ascii: Carro Piscina Sitio ap casa
escola quadra
Meus utensílios em ordem alfabetica: ap Carro casa escola
Piscina quadra Sítio
Meus utensílios em ordem decrescente: Sitio quadra Piscina
escola casa Carro ap
```

7.3.2 Transformando strings em arrays e arrays em strings

A função interna ***split*** converte uma string em um array cujos elementos são separados pelo elemento separador informado – caractere, string ou regex. De forma contrária, a função ***join*** une os elementos de array em uma única string com o elemento de união informado.

```
  1 #exemplo7.7
  2 use 5.12.0;
  3
  4 my $csv =
  5  'vamos supor;um arquivo;do tipo csv;que seria; uma
abstração; de tabela;';
  6 my @elementos = split /;/, $csv;
  7 say "Após split: @elementos";
  8 my $string_final = join " ", @elementos;
  9 say "após join: $string_final";

saída:

Após split: vamos supor um arquivo do tipo csv que seria
uma abstração  de tabela
após join: vamos supor um arquivo do tipo csv que seria
uma abstração  de tabela
```

```
  1 #exemplo7.8
  2 use 5.12.0;
  3
  4 my @csv = ('vamos supor;um arquivo;do tipo csv',
  5         'que seria;uma abstração;de tabela;',
  6          'onde cada; linha é;separada por \\n',
  7         'e cada elemento;pelo caractere;ponto-e-virgu-
la');
  8
  9 my @elementos = map {join " ", split /;/} @csv;
 10 say "Meu CSV: @csv";
 11 say "Linha: ", $_ for @elementos;

saída:

Meu CSV: vamos supor;um arquivo;do tipo csv que seria;uma
abstração;de tabela; onde cada; linha é;separada por \n e
cada elemento;pelo caractere;ponto-e-virgula
Linha: vamos supor um arquivo do tipo csv
Linha: que seria uma abstração de tabela
Linha: onde cada  linha é separada por \n
Linha: e cada elemento pelo caractere ponto-e-virgula
```

7.4 Manipulando os hashes

Já utilizamos muitos hashes nos exemplos ao longo do livro. Logo, você sabe que:

- Os hashes são coleções de dados ou tabelas formadas por pares "chaves" e "valores" ou keys e values. As posições destes pares não são ordenadas como as dos arrays;
- Têm alocamento de recursos dinâmico;
- Sempre devem ter um número par de elementos;
- Variáveis hash tem o prefixo % e o índice fica entre chaves { }.

7.4.1 Funções para manusear hashes

Para manusear os hashes o Perl provê funções muito interessantes. Vamos começar com as funções ***keys*** *e* ***values***. Estas funções retornam um array com as chaves ou valores das chaves de um hash. Mas lembre-se de que os hashes não armaze-

nam os valores de forma ordenada como os arrays. Na versão 5.12.0 do Perl foi adicionada a estas funções a capacidade de trabalhar com arrays.

Detalhe importante: estas funções resetam o iterador interno do array ou do hash. Este iterador é utilizado em loops ***for*** e ***foreach***.

```
  1 #exempl7.9
  2 use 5.12.0;
  3
  4 my %propriedades = (
  5         casa => 150000.00,
  6         carro => 30000.00,
  7         ap => 80000.00,
  8         );
  9
 10 say for keys %propriedades;
 11 say for values %propriedades;
 12
 13 #No perl 5.12 as funções keys e values também funcionam
com arrays
 14 my @array = (('A'..'Z'), ('a'..'z'));
 15 say values @array;
 16 while (my($chave, $valor) = each %propriedades){
 17         say "$chave custa $valor";
 18 #Se você ativar a linha abaixo o loop executa para sem-
pre...
 19 #       keys %propriedades;
 20 }
```

saída:

```
casa
carro
ap
150000
30000
80000
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
casa custa 150000
carro custa 30000
ap custa 80000
```

A função ***each*** retorna um par chave valor a cada iteração do loop. Se a função ***keys*** ou a função ***values*** for executada internamente ao loop, o índice interno será reiniciado, então o loop do nosso exemplo executará para sempre. Se o argumento da função ***each*** for um array iremos receber um par índice valor.

```
 1 #exemplo7.10
 2 use 5.12.0;
 3 my @letras = ('a'..'e');
 4
 5 while (my ($indice, $letra) = each @letras){
 6         say "$indice => $letra";
 7 }

saída:

0 => a
1 => b
2 => c
3 => d
4 => e
```

Além destas, o Perl oferece a função ***exists*** para verificar se uma determinada chave existe ou foi inicializada no hash, mesmo que seu valor seja ***undef***:

```
 1 #exemplo8.11
 2 use 5.12.0;
 3
 4 my %propriedades = (
 5         casa => 150000.00,
 6         carro => 30000.00,
 7         ap => 80000.00,
 8         moto => undef
 9         );
10
11 say "Você tem um carro" if exists $propriedades{carro};
12 say "Você tem uma moto" if exists $propriedades{moto};
13 say "Voce tem um sítio" if exists $propriedades{sitio};
```

```
 14 #a chave sitio não existe
 15
 16 map{say "$_ vale $propriedades{$_}" if defined $pro-
priedades{$_}}
 17  keys %propriedades;
 18 #Repare que o valor de moto é undef, logo não é impres-
so por say loop.

saída:

Você tem um carro
Você tem uma moto
casa vale 150000
carro vale 30000
ap vale 80000
```

A última função nativa importante para trabalhar com ***hashes*** que discutiremos aqui é a função ***delete***. Ela apaga uma chave de um hash.

```
  1 #exemplo7.12
  2 use 5.12.0;
  3
  4 my %propriedades = (
  5         casa => 150000.00,
  6         carro => 30000.00,
  7         ap => 80000.00,
  8         moto => undef
  9         );
 10 delete $propriedades{carro}; #deleta apenas um elemento
 11 delete @propriedades{qw/casa ap/}; #deleta um slice de
elementos
 12 say "Sobrou apenas $_" for keys %propriedades;

saída:

Sobrou apenas moto
```

Note que podemos fornecer uma chave ou um array de chaves para deletar.

7.5 Arrays de arrays...

Como dissemos, os arrays armazenam coleções de escalares. Também dissemos que um escalar pode armazenar uma referência a um array. Então, por lógica simples, podemos concluir que um array pode armazenar escalares que são referências para outros arrays. Confundiu? Vejamos...

```
  1 #exemplo7.13
  2 use 5.12.0;
  3
  4 my @array_numeros = 0 .. 9 ; #criamos alguns arrays
  5 my @array_letras = 'a' .. 'z' ;
  6 #criamos referências para os arrays
  7 my $ref1 = \@array_numeros;
  8 my $ref2 = \@array_letras;
  9 #e criamos um array de referências a arrays, ou arrays
de arrays
 10 my @array_of_arrays = ($ref1, $ref2);
 11 say "@array_of_arrays";
 12 #para acessarmos, usamos:
 13 say $array_of_arrays[0][1]; #acessa o elemento 1 do ar-
ray no índice 0
 14 say $array_of_arrays[1][20]; #acessa o elemento 20 do
array no índice 1

saída:

ARRAY(0x1370f8c) ARRAY(0x137124c)
1
u
```

Veja que o ***@array_of_arrays*** armazena as referências em seus índices. Assim, e para acessarmos os elementos finais, usamos o índice do primeiro array e o índice do segundo array. ***O Perl não impõe limites, de forma que você pode inserir arrays em arrays, desta maneira sendo limitado apenas pela capacidade do***

seu sistema. Mas nosso exemplo ainda tem que melhorar. É chato ter que declarar dezenas de arrays para somente depois colocá-los num outro array. Para isto existem os *arrays anônimos.* Os arrays anônimos não têm uma variável do tipo array associadas a eles... eles já nascem referências. Veja no exemplo como criar um array anônimo:

```
  1 #exemplo7.14
  2 use 5.12.0;
  3
  4 my $cidade = ['São Paulo', 'SP', 'Brasil']; #Este array
é anônimo
  5 my $endereco = ['Rua dos bobos', 'número 0', 'Bairro
das Toupeiras'];
  6 my @dados = ($endereco, $cidade);
  7 for my $array (@dados){
  8         say join ", ", @$array;
  9 }
 10 #ou podemos ainda criar as referências como elementos
do array
 11 # e a referência é passada direto para o índice.
 12 my @dados2 = ( ['São Paulo', 'SP', 'Brasil'],
 13               ['Rua dos bobos', 'número 0', 'Bairro das
Toupeiras']);
 14 say "Estado: ", $dados2[0][1];

saída:

Rua dos bobos, número 0, Bairro das Toupeiras
São Paulo, SP, Brasil
Estado: SP
```

7.6 Arrays de hashes

Seguindo a mesma linha de abstração, os hashes são coleções de escalares indexadas por uma chave. Estes hashes podem ser referenciados por uma variável escalar e armazenados dentro de um array:

```
  1 #exemplo7.15
  2 use 5.12.0;
  3
  4 my %livro = ( titulo => 'Viagem ao Centro da Terra',
  5                 autor => 'Jules Gabriel Verne',
  6                 editora => 'diversas'); #crio um hash
  7
  8 my %linux = ( distro => 'Ubuntu',
  9                 versao => '10' ); #crio outro hash
 10
 11 my @array_of_hashes = (\%livro, \%linux); #armazeno as
referências no array
 12 say "Meu array contém: @array_of_hashes";
 13 printf "Meu livro preferido é %s do autor %s\n",
 14        $array_of_hashes[0]{titulo},
 15         $array_of_hashes[0]->{autor};
 16
 17 say "Meu linux:";
 18 for (sort keys %{$array_of_hashes[1]}){
 19         say "\t$_ => $array_of_hashes[1]{$_}"
 20 }
```

saída:

```
Meu array contém: HASH(0x99dad58) HASH(0x99dafb8)
Meu livro preferido é Viagem ao Centro da Terra do autor
Jules Gabriel Verne
Meu linux:
     distro => Ubuntu
     versao => 10
```

7.7 Hashes de arrays

Seguindo a mesma filosofia, podemos criar hashes povoados com referências a arrays ou mesmo outros hashes indefinidamente. E como sempre: limitado apenas pela capacidade do seu sistema e pela nossa capacidade humana de abstrair...

```
  1 #exemplo7.16
  2 use 5.12.0;
  3 use utf8;
  4 #utf8 permite utilizar caracteres latinos, como ç em
nomes de variáveis.
  5
  6 my %programacao = (
  7 segunda => [{nome => 'two and a half', horário =>
"19:00", canal => 'WB'},
  8                 {nome => 'Simpsons', horário =>
"21:00", canal => 'FX'}],
  9 quinta => [{nome => 'house', horário => "20:00", canal
=> 'UC'},
 10            {nome => 'Law and order', horário =>
"21:00", canal => 'UC' }],
 11 );
 12
 13 for my $dias (reverse sort keys %programacao){
 14     say "Na $dias eu assisto:";
 15     for my $programa (@{$programacao{$dias}}){
 16        for (reverse sort keys %{$programa}){
 17              say "${_}: $$programa{$_}";
 18         }
 19     }
 20 }
```

saída:

```
Na segunda eu assisto:
nome: two and a half
horário: 19:00
canal: WB
nome: Simpsons
horário: 21:00
canal: fx
```

```
Na quinta eu assisto:
nome: house
horário: 20:00
canal: UC
nome: Law and order
horário: 21:00
canal: UC
```

Para finalizar a nossa conversa sobre estruturas de dados mais complexas, vamos falar de hashes de funções.

7.8 Hashes de funções

Agora, caso tenha se confundido com isso tudo, eu esclareço! Além das referências a arrays, hashes e outros "bichos", podemos fazer uma referência a uma função e até declarar esta função de forma anônima:

```
  1 #exemplo7.17
  2 use 5.12.0;
  3
  4 sub power_of{ #crio a função powerof
  5         my %args = @_;
  6         return $args{base}**$args{expoente}; #retorna o
valor calculado
  7 }
  8
  9 #Referência a uma função nomeada.
 10 my $sub_ref = \&power_of;
 11 say "A referência para uma sub é do tipo CODE: ", $sub_
ref;
 12 say "3 elevado ao cubo é ", $sub_ref->(base => 3, expo-
ente => 3);
 13
 14 #Mas como os outros tipos, podemos ter uma referência
anônima:
 15 my $radix = sub{ my %args = @_;
 16                     return $args{base} **
(1/$args{raiz}) };
 17
 18 say "A referência de \$radix é: ", $radix;
```

```
 19 say "A raíz cúbica de 46 é: ", $radix->(base => 46,
raiz => 3);

saída:

A referência para uma sub é do tipo CODE: CODE(0x817bd30)
3 elevado ao cubo é 27
A referência de $radix é: CODE(0x817c020)
A raíz cúbica de 46 é: 3.58304787101595
```

Agora que você sabe como associar as funções a um escalar, pode colocar este escalar em um array ou hash, da mesma forma que colocamos as outras referências ou mesmo utilizar este escalar como retorno de uma outra função. Falaremos disso mais adiante.

```
  1 #exemplo7.18
  2 use 5.12.0;
  3 use Net::Ping;
  4
  5 #um hash de funções...
  6 my %testes = (
  7     http => sub{
  8         my $host = shift; #variável @_ é implícita den-
tro de uma função
  9         my @to_return = ping_port(host => $host, porta
=> 80);
 10         return wantarray ? @to_return : $to_return[0];
 11    },
 12    ftp =>  sub{
 13         my $host = shift; #a variável @_ é implícita.
Mesmo que shift @_.
 14         my @to_return = ping_port(host => $host, porta
=> 21);
 15         return wantarray ? @to_return : $to_return[0];
 16   },
 17 );
 18
 19 #usando as funções do hash...
```

```
 20 my @ping_teste = $testes{http}->('microsoft.com');
 21 printf "IP: %s\t Isup: %s \t tempo: %.5f seg\n", $ping_
teste[2],
 22         $ping_teste[0] ? 'Sim' : 'Não', $ping_teste[1];
 23
 24 my $is_live_ftp = $testes{ftp}->('ftp.freebsd.org');
 25 say "ftp da freebsd está up!" if $is_live_ftp;
 26
 27 sub ping_port {
 28 #uso: fornecer argumentos porta => 'numero_da_porta'
 29 #                          host => 'ip_ou_nome'
 30
 31    my %args = @_;
 32    my $ping = Net::Ping->new( );
 33    $ping->port_number($args{porta});
 34    $ping->hires();
 35    my @to_return = $ping->ping($args{host}, 5.5);
 36    return @to_return; #retorna resultados do ping
 37 }

saída:

IP: 207.46.232.182      Isup: Sim   tempo: 0.22641 seg
ftp da freebsd está up!
```

E aí está. Um hash de funções... se você está pensando: "isto é muito trabalhoso, eu teria menos trabalho usando subs normais", pense nesta possibilidade: Logo você vai aprender a criar módulos. Imagine um módulo que te permita fazer isso:

```
my @resultado = $teste{ftp}->($host)
```

para todos os seus hosts e portas disponíveis? Bem menos trabalhoso, certo?

Exercícios

1. Monte uma função em Perl que verifique se um array é um subgrupo de outro array, ou seja, se os elementos de um array estão ou não em outro array.

2. Monte funções para as seguintes operações de grupos de elementos, onde os grupos são representados por arrays: União, Interseção (quais elementos os arrays têm em comum) e Diferença (quais elementos os arrays não têm em comum).

3. Crie uma função que faça uma pesquisa binária em um array de strings.

4. Com as funções shift, unshift, pop e push é muito fácil criar uma função que receba um array e o processe como uma lista circular. Crie um array de subs que serão processadas como uma lista circular por uma função. Se não souber o que é uma lista circular, faça uma pesquisa na internet ou em um livro sobre algoritmos e tipos de dados.

5. Crie uma função que imprima as chaves e os valores de um hash organizados por chave.

6. Crie uma função que imprima as chaves e os valores de um hash organizados por valor.

7. Monte um programa em Perl que abra um arquivo qualquer com o módulo Filehandle e conte o número de palavras, linhas e caracteres do arquivo. Se tiver dúvidas sobre o módulo Filehandle leia o ***perldoc*** deste módulo.

8. Monte um programa que peça para o usuário escolher um número. Então o programa executa uma função qualquer em um array de funções, cujo número escolhido seja o índice do array. Observe que isso funciona como um given/when, só que é executado mais rapidamente.

9. Refaça os exercícios dos capítulos anteriores e veja como você pode incrementar código e reduzir esforços com as funções que aprendemos.

10. Crie uma função que verifique chaves idênticas em dois hashes.

11. Crie uma função que verifique valores idênticos em dois hashes.

12. Crie uma função que verifique pares valor e chave idênticos em dois hashes.

13. Crie uma função que verifique se um hash é um subconjunto de outro hash (ou seja, se todas as suas chaves e valores estão inseridos no outro hash).

Capítulo 8. Expressões Regulares

Uma das características mais famosas do Perl é a sua capacidade de processar grandes volumes de dados e gerar relatórios por meio do seu engenho de expressões regulares. Uma vez definido um padrão, o Perl pode informar se ocorreu, onde ocorreu e como ocorreu um match deste padrão nos dados em processamento.

Um padrão é processado no Perl por meio do operador **m**, que pode ser meramente omitido. Dada a frequente utilização das regex na linguagem, o operador **m** se tornou um elemento implícito. Uma expressão como **m/ao/** pode aparecer na forma simplificada **/ao/** e é altamente integrada com as variáveis implícitas do Perl. Assim:

```
$_ =~ m/ao/;
```

torna-se

```
/ao/;
```

Indo muito além, o **engenho de regex** possui diversos **metacaracteres** e otimizações que aumentam o poder das regex no Perl, transformando uma centena de linhas de código de verificação em apenas uma linha. Em contrapartida, pode-se considerar sua interpretação onerosa para olhares menos atentos ou profissionais que não conheçam os detalhes das regex em Perl.

Além da verificação de match, existe o operador de substituição **s/regex/substituição/**, que realiza modificação automaticamente. O engenho também permite a integração com funções como **split** e **grep**.

8.1. As expressões regulares

As expressões regulares são um conjunto de caracteres ou metacaracteres com um significado específico. Uma expressão regular, por exemplo, é avaliada caractere por caractere de acordo com a sequência informada na regex. Assim, uma regex como **/bat-movel/** é validada somente se o texto analisado tiver todos os caracteres informados:

```
  1 #exemplo8.1
  2 use 5.12.0;
  3
  4 my $string1 = "O peito do pé de pedro é preto";
  5 my $string2 = "O rato roeu a roupa do rei de roma";
  6 print "Match pe in \"$string2\"\n" if $string2 =~ /pe/;
  7 print "matched pe in \"$string1\"\n" if $string1 =~ /
pe/;

saída:

matched pe in "O peito do pé de pedro é preto"
```

No exemplo8.1 a regex **/pe/** só realizou match (retornou true com o operador =~) após encontrar uma sequência **pe** qualquer. Isso é muito semelhante ao mecanismo de buscas de muitos editores e processadores de texto, e posso garantir que muitos usam o engenho de **regexp** do Perl. E como ter certeza de qual **pe** o Perl encontrou na minha string se temos as palavras pedro e preto com a sequência pe? Inicialmente com as variáveis **Pre-match ($`), Match ($&) e Pós-Match ($')**:

```
  1 #exemplo8.2
  2 use 5.12.0;
  3
  4 my $string1 = "O peito do pé de pedro é preto";
  5 my $string2 = "O rato roeu a roupa do rei de roma";
  6 say "Match pe in \"$string2\"" if $string2 =~ /pe/;
  7 say "matched pe in \"$string1\"" if $string1 =~ /pe/;
  8
  9 say $`; #Pré Match
```

```
10 say $&; #Match
11 say $'; #Pós Match

saída:

matched pe in "O peito do pé de pedro é preto"
O
pe
ito do pé de pedro é preto
```

Na forma que está escrita, a regex **/pe/** informa a primeira sequência de caracteres idênticos encontrados na string e não a **"sequência exata"**. De sorte que, mesmo peito e pedro tenham a mesma sequência, peito aparece primeiro. Agora vamos ampliar o poder das **regex** com os **metacaracteres.**

8.2 Os metacaracteres

O prefixo meta, de origem grega, significa mudança. Os **metacaracteres** têm como função ampliar as capacidades de uma regex, modificando o significado dos caracteres de uma regex. Eles não buscam pelo seu caractere específico, mas implicitamente buscam outra informação mais ampla.

O metacaractere **?** busca pela existência ou não do caractere anterior. Por exemplo: **$string =~ m/pei?/** significa que estamos procurando uma sequência que pode conter ou não o caractere **i**. Assim, no exemplo8.3, o match ocorreria em peito e pedra.

```
1 #exemplo8.3
2 use 5.12.0;
3
4 my @strings = qw/pedra peito prata/;
5 for(@strings){
6         print "$& => $_\n" if /pei?/
7 }

saída:

pe => pedra
pei => peito
```

Veja neste exemplo que a **regexp** efetuou o match tanto com **pe** como **pei**. Assim, vamos pegar um exemplo mais mateiral: o nome Tiago tem duas vertentes, uma com h e outra sem h. Assim **Thiago** e **Tiago** são o mesmo nome. Se quisermos pesquisar as duas possibilidades a **regex**, seria: **/Th?iago/**. Veja:

```
  1 #exemplo8.4.pl
  2 use 5.12.0;
  3
  4 my @strings = qw/Toledo Thiago Felipao Tiago Triago/;
  5 for(@strings){
  6         print "$& => $_\n" if /Th?i/
  7 }

saída:

Thi => Thiago
Ti => Tiago
```

O metacaractere **** elimina o significado do metacaractere que o prossegue tornando-o um caractere comum. Assim, **\?** significaria que estamos procurando pelo próprio caractere ?:

```
  1 #exemplo8.5.pl
  2 use 5.12.0;
  3
  4 my @strings = qw/Toledo Thiago Felipao Tiago Triago
Th?iestrano/;
  5 for(@strings){
  6         print "$& => $_\n" if /Th\?i/
  7 }

saída:

Th?i => Th?iestrano
```

O metacaractere **.** (ponto) significa **qualquer caractere**. Veja:

```
  1 #exemplo8.6
  2 use 5.12.0;
  3
  4 my @strings = qw/Toledo Thiago Felipao Tiago Triago
Th?iestrano/;
  5 for(@strings){
  6         print "$& => $_\n" if /T.i/
  7 }

saída:

Thi => Thiago
Tri => Triago
```

A regex **/T.i/** realizou o match com Thiago e Triago. Os metacaracteres também podem ser combinados, assim **/T.?i/** significaria ter ou não um caractere entre T e i:

```
  1 #exemplo8.7.pl
  2 use 5.12.0;
  3
  4 my @strings = qw/Toledo Thiago Felipao Tiago Triago
Th?iestrano/;
  5 for(@strings){
  6         print "$& => $_\n" if /T.?i/
  7 }

saída:

Thi => Thiago
Ti => Tiago
Tri => Triago
```

O metacaractere **+** indica que deve haver pelo menos 1 caractere igual ao caractere que o antecede, mas é possível haver vários outros. De forma que **/a+/** realizará match com a, aa, aaaa...

```
1 #exemplo8.8
2 use 5.12.0;
3
4 my @strings = qw/casa caaabanaaaa caastelooo/;
5 for(@strings){
6 $_ =~ s/a+/a/g;
7         print "$_\n";
8 }
```

saída:

```
casa
cabana
castelooo
```

O operador **s** executou a substituição dos agrupamentos de **aaa** por **a**. O operador ***g*** colocado ao final da expressão informa ao compilador perl que ele não deve se limitar ao primeiro match, mas realizar a substituição em todos os agrupamentos do caractere **a** que encontrar na string.

Com o * dizemos que pode haver uma, muitas ou nenhuma aparição do caractere que o antecede. Assim, **/.*/** efetuaria match com qualquer string existente. Vejamos:

```
1 #exemplo8.9
2 use 5.12.0;
3
4 my @strings = qw/thiago tiago thhhhiago triago/;
5 for (@strings){
6         print "$& => $_\n" if /th*i/;
7 }
```

saída:

```
thi => thiago
ti => tiago
thhhhi => thhhhiago
```

A regex **/th*i/** informa que pode haver um **h**, uma sequência do caractere **h** ou nenhum **h** entre os caracteres **t** e **i**. Qualquer outro caractere exclui a sequência. Logo, a expressão efetuou match com **thi, ti** e **thhhhi,** mas não **tri**.

Porém os **metas *, ?, +** e **.** são muito genéricos para algumas aplicações. Assim podemos nos valer de quantificadores mais específicos. Com { } eu posso quantificar um máximo e mínimo de aparições de um determinado caractere na minha regex. Assim, **/th{4}i /** significa um **t** seguido de **4 h** e um **i**.

```
1 #exemplo8.10
2 use 5.12.0;
3
4 my @strings = qw/thiago tiago thhhhiago triago/;
5 for (@strings){
6         say "$& => $_" if /th{4}i/;
7 }

saída:

thhhhi => thhhhiago
```

O quantificador { } também permite que eu estabeleça um valor mínimo e um valor máximo no qual o caractere deve aparecer: {2,4} significa no mínimo duas vezes e no máximo seis.

```
1 #exemplo8.11
2 use 5.12.0;
3
4 my @strings = qw/tiago thiago thhiago thhhiago thhh-
hiago triago/;
5 for (@strings){
6         say "$& => $_" if /th{2,4}i/;
7 }

saída:

thhi => thhiago
thhhi => thhhiago
thhhhi => thhhhiago
```

Ainda podemos fazer agrupamentos com os parênteses ( ), de forma que os caracteres dentro de um agrupamento sejam interpretados como um único elemento pelos outros **metacaracteres**. Por exemplo, a regex **/(00:){2}00/** significa **00:00:00**. Nesta regex o conjunto de caracteres **00:** deve ocorrer pelo menos duas vezes. Podemos criar uma máscara interessante para fazer o match em da-

dos com formato de hora com o metacaractere **\d** (que significa dígito) com a regex **/(\d{2}:){2}\d{2}/**. Essa regex efetuaria match com **12:30:40** ou com 03:18:01, mas não com **03:125:26.** Por exemplo:

```
  1 #exemplo8.12
  2 use 5.12.0;
  3
  4 my @strings = qw/18:30:23 52c:00:18 23:41:123
124:04:13/;
  5 for(@strings){
  6         say "$&" if /(\d{2}:){2}\d{2}/;
  7 }

saída:

18:30:23
23:41:12
24:04:13
```

Nossa máscara para horas ainda tem uma falha. Ela aceita strings com mais de dois dígitos no final ou no início da sequência. Apesar do match ocorrer somente com os dois dígitos antes do caractere, a nossa regex permite dígitos antes ou depois dos limites dentro da string. Para sanar estes problemas temos os metas: **^** e **$**. Eles significam, respectivamente, no começo da string e no fim da string. Então nossa máscara ficará perfeita assim: **/^(\d{2}:){2}\d{2}$/**. Esta regex também poderia ter sido escrita como: **/^\d\d:\d\d:\d\d$/,** mas da primeira forma eu pude demonstrar como usar diversos metacaracteres do Perl. Vejamos a nossa nova máscara em ação:

```
  1 #exemplo8.13
  2 use 5.12.0;
  3
  4 my @strings = qw/18:30:23 52c:00:18 23:41:123
124:04:13/;
  5 for(@strings){
  6         print "$&\n" if /^(\d{2}:){2}\d{2}$/;
  7 }

saída:

18:30:23
```

O metacaractere **|** tem a função de operador lógico **or** dentro de uma regex. /(Thiago|Felipe)/, por exemplo, significa match com Thiago ou com Felipe.

```
  1 #exemplo8.14
  2 use 5.12.2;
  3
  4 my @sentenca;
  5 push @sentenca, "Então Thiago disse: Esse tal de Perl é
da hora!!!",
  6             "Eu prefiro VB, respondeu o Script Kid",
  7             "Também gostei do Perl, retrucou Unix-Guy";
  8
  9 say for grep(/(Thiago)|(Unix-Guy)/, @sentenca);

saída:

Então Thiago disse: Esse tal de Perl é da hora!!!
Também gostei do Perl, retrucou Unix-Guy
```

Neste caso a função **grep** procura por Thiago ou Unix-Guy. Além destes, temos o metacaractere para criação de classes de caracteres. Por meio dele podemos especificar grupos de caracteres, como **[abcdef]**, que irão efetuar o match caso ocorra a incidência de qualquer um destes caracteres na string. Também podemos especificar este mesmo agrupamento de caracteres com **[a-f]**. Assim podemos criar padrões mais complexos, como **[0-9]** para dígitos ou **[a-z]** para letras minúsculas. Podemos ainda juntar diversos grupos e formar uma classe complexa, como alfanumérica **[0-9a-zA-Z]**, ou especificar um par maiúsculo/minúsculo **[Aa]**. Um exemplo da utilidade deste meta é verificar se não existem caracteres indevidos na entrada do usuário.

```
  1 #exemplo8.15
  2 use 5.12.0;
  3
  4 sub is_integer{
  5 my $number = shift;
  6         return $number =~ m(^[0-9]+(\.0+)?$) ? 1 : 0;
  7 }
```

```
 8 say "Digite um número inteiro: ";
 9 chomp(my $int = <STDIN>);
10 if(is_integer $int){
11         say "Você digitou um inteiro"
12 }
13 else{
14         say "Este número não é um inteiro"
15 }
```

saídas:

```
Digite um número inteiro:
23
Você digitou um inteiro

Digite um número inteiro:
45.10
Este número não é um inteiro

Digite um número inteiro:
112.00
Você digitou um inteiro
```

Neste exemplo a regex funciona da seguinte forma: **^[0..9]+** diz que a string deve conter uma sequência de dígitos de 0 a 9, como qualquer número inteiro. Além disso, os metacaracteres definem que a string deve começar com dígitos. Definimos que a sequência pode terminar com um ponto, desde que seguido de zeros **(\.0+)? $**. Essa regex efetua match com sequências de dígitos como 97, 192768 ou 1, bem como números com representação decimal, desde que contenha apenas zeros, como 95.00, 127.0000 e assim por diante.

8.2.1 Atalhos para classes de caracteres

O metacaractere [ ] permite que criemos classes de caracteres, porém algumas classes são tão comuns que foram criados atalhos no Perl para elas. Assim temos:

Atalho	Significado	Agrupamento:
\d	Todos os dígitos	[0-9]
\D	Exceto dígitos	[^0-9]
\w	Letras, números e _	[a-zA-Z0-9_]
\W	Exceto letras e números	[a-zA-Z0-9_]
\s	Espaços	[\t\n\f\r]
\S	Exceto espaços	[^ \t\n\f\r]

Ao estudar esta tabela você vai perceber que o metacaractere **^** tem seu significado modificado quando está entre colchetes: **[^]**. Ele significa **exclusão** dos elementos dentro da classe.

Assim, nossa regex do exemplo anterior, **m(^[0-9]+(\.0+)?),** pode ser simplificada como **m{^\d+(\.0+)?}**.

8.3 Os operadores m/// e s///

Os operadores de regex do Perl têm certas características que devem ser ressaltadas. A mais interessante é a capacidade de concatenar strings, permitindo que expressões regulares sejam criadas dinamicamente. Assim, se eu criar um algoritmo que pesquise determinados textos em um arquivo, como são feitos os mecanismos de pesquisa dos editores de texto, eu poderia fazer:

```
 1 #exemplo8.16
 2 use 5.12.0;
 3 use FileHandle;
 4
 5 #caminho do arquivo
 6 my $path = '/etc/apache2/apache2.conf';
 7 my $file = FileHandle->new($path, "r");
 8 print "Informe o texto a ser pesquisado: ";
 9 chomp (my $to_search = <STDIN>);
10 my $line = 0;
```

```
 11 while($_ = $file->getline()){
 12         $line++;
 13         print ":$line: $_" if /$to_search/;
 14 }

saída:

Informe o texto a ser pesquisado: ServerRoot
:27: # with "/", the value of ServerRoot is prepended -- so
"/var/log/apache2/foo.log"
:28: # with ServerRoot set to "" will be interpreted by the
:40: # ServerRoot: The top of the directory tree under
which the server's
:50: ServerRoot "/etc/apache2"
```

Veja que, ao informarmos a expressão ServerRoot, o programa pesquisou todas as linhas do arquivo e informou seu valor. Porém, este programa vai além e permite que o usuário informe expressões regulares com metacaracteres:

```
Informe o texto a ser pesquisado: ^ServerRoot
:50: ServerRoot "/etc/apache2"
```

Pesquisando por **^ServerRoot** nosso programa retorna apenas as linhas que começam com este valor.

Não sendo o bastante, ainda temos, adicionada a toda essa flexibilidade, a opção de modificar o delimitador da regex. Assim, em vez de fazer **/regex/** podemos fazer **m#regex#** ou **m(regex)**. E para o operador **s** podemos usar **s#regex# #string#** ou **s{regex} {string}**.

Os operadores também possuem modificadores que atuam na forma como a expressão é analisada, ou como a string é percorrida. Os modificadores mais importantes são:

Modificador	Função
i	Ignora o case dos caracteres
g	Verificação global
o	Compila a regex apenas uma vez
s	O meta . também efetua match com \n
x	Permite comentários dentro da regex

Vamos ver alguns exemplos:

O modificador **i** ignora o case da **regex**:

```
 1 #exemplo8.5
 2 use 5.12.0;
 3
 4 my @strs = qw/casa carro Casa CArro/;
 5 say "Matches sem o modificador i:";
 6 for(@strs){
 7         print "$_ " if /ca/;
 8 }
 9 say "\n";
10 say "Matches com o modificador i";
11 for(@strs){
12         print "$_ " if /ca/i;
13 }

saída:
Matches sem o modificador i:
casa carro

Matches com o modificador i
casa carro Casa CArro
```

O modificador **g** verifica todos os matches de uma expressão, não apenas o primeiro. Este modificador também retorna um array com todos os matches de uma expressão:

```
 1 #exemplo8.18
 2 use 5.12.0;
 3
 4 my $str = "Frodo Baggins e Bilbo Baggins são parentes";
 5 my @match = $str =~ /b/i;
 6 my @matches = $str =~ /b/gi;
 7 say "Sem o modificador g: @match";
 8 say "Com o modificador g: @matches";

saída:

Sem o modificador g: 1
Com o modificador g: B B b B
```

E com o modificador **x** podemos adicionar comentários a expressões regulares muito longas ou complexas:

```
  1 #exemplo8.19
  2 use 5.12.0;
  3
  4 print "Digite uma data (formato dd/mm/aaaa): ";
  5 chomp (my $date = <STDIN>);
  6 if($date =~ m[  #inicia a regex
  7         (\d\d)/         #dia
  8         (\d\d)/         #mes
  9         (\d{4})         #ano]x){
 10         say "$date está no formato correto"
 11 }
 12 else{
 13         say "$date está com formato incorreto"
 14 }

saída:

Digite uma data (formato dd/mm/aaaa): 05/06/1979
05/06/1979 está no formato correto

Digite uma data (formato dd/mm/aaaa): 13/2/80
13/2/80 está com formato incorreto
```

Este modificador é extremamente interessante pois permite que deixemos as regex mais legíveis e deve ser introduzido na sua lista de boas práticas.

O operador **m** também alimenta as variáveis ocultas **$1, $2, $3...$n** quando usamos agrupamentos, **( )**. Assim podemos adicionar uma funcionalidade no programa anterior adicionando estas linhas ao final:

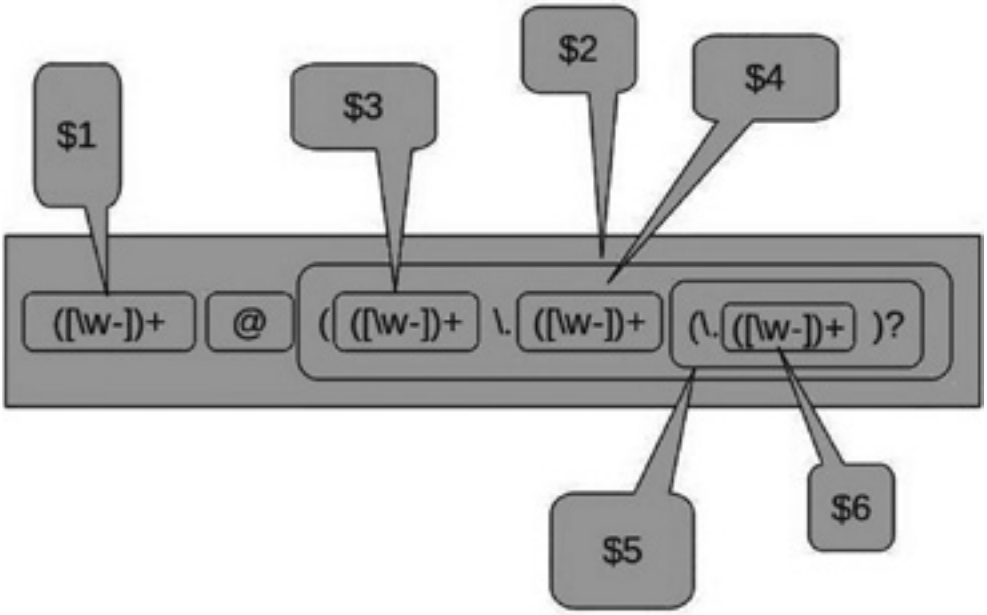
```
15 say "Dia informado: $1";
16 say "Mês informado: $2";
17 say "Ano: $3";

saída:

Digite uma data (formato dd/mm/aaa): 13/02/1980
13/02/1980 está no formato correto
Dia informado: 13
Mês informado: 02
Ano: 1980
```

O que fizemos aqui é chamado agrupamento. Os agrupamentos são subelementos das regex, que podem ser obtidos a fim de facilitar algoritmos, como obter as partes de uma data, hora ou e-mail. Mais um exemplo: vamos supor um formulário Web que verifica o formato de um e-mail e separa o nome de usuário do domínio.

```
  1 #!/usr/local/bin/perl -T
  2 #exemplo8.20
  3 #Verifique com seu provedor a versão suportada do Perl.
  4
  5 use strict;
  6 use warnings;
  7 use CGI;
  8 use CGI::Carp qw/fatalsToBrowser/; #Não usar esta linha
em produção
  9 use Data::Validate::Email qw(is_email is_email_rfc822);
 10
 11 my $cgi = CGI->new();
 12 print $cgi->header(-type => 'Content-Type', -charset =>
'iso-8859-1'),
 13         $cgi->start_html(-title => "Verifica e-mail");
 14
 15 if ( not $cgi->param("Email") ) {
 16          print $cgi->start_form(-action => $cgi-
>url()),
 17           $cgi->p("Informe seu e-mail " .
 18           $cgi->textfield(-name => "Email") . " " .
 19           $cgi->submit(-name => "testar")),
 20           $cgi->end_form;
 21 }
 22 else{
 23         my $mail = $cgi->param("Email");
 24         if (not is_email($mail)){
 25                 die "Email informado inválido\n";
 26         }
 27         $mail =~ m{^([\w-]+)\@(([\w-]+)\.([\w-]+)(\.([\
w-]+))?)};
 28         print $cgi->p("Username = $1"),
```

```
29                    $cgi->p("Domain = $2"),
30                    $cgi->p("\t Parte 1 = $3"),
31                    $cgi->p("\t Parte 2 = $4");
32     print $cgi->p("\t Parte 3 = $6") if $5;
33 }
```

Neste exemplo a regex **m{^([\w-]+)\@(([\w-]+)\.([\w-]+)(\.([\w-]+))?)}** divide a string em duas partes, uma antes do @ e outra depois. Então **$1** irá conter o nome do usuário e ***$2*** o nome completo do domínio. **$3** irá conter o valor imediatamente depois de @ e antes do primeiro ponto que divide o nome do domínio na internet. A variável **$4** irá conter o restante do nome, se for apenas .com ou .net. Como existem domínios .com assim como .com.br ou co.uk, tivemos que fazer mais um agrupamento. Então, em um domínio .com.br, ***$5*** irá conter .br e ***$6*** apenas br.



8.4 Verificando a posição na string

Às vezes o importante não é apenas efetuar o match, mas também saber as posições onde o match ocorre. Para isto usamos a função **pos( )** juntamente com o modificador **g**. O operador **g** deixa um cursor informando a posição onde ocorreu

o último match. O operador **pos** retorna a posição deste cursor. Leve em consideração que a posição inicial do cursor é sempre **0.**

```
  1 #exemplo8.21
  2 use 5.12.0;
  3
  4 my $string = "Frodo e Bilbo Baggins são parentes";
  5 while(my $match = $string =~ /b/gi){
  6     say "$& encontrado na posição ", pos($string) - 1;
  7 }

saída:

B encontrado na posição 8
b encontrado na posição 11
B encontrado na posição 14
```

Utilizamos **pos($string) – 1** porque o cursor fica posicionado no próximo caractere da string sendo percorrida. Não se esqueça de que ele é posicionado logo após o final do padrão. Portanto, se o padrão contiver diversos caracteres, o cursor estará após o último caractere.

```
  1 #exemplo8.22
  2 use5.12.0;
  3
  4 my $string = "Frodo e Bilbo Baggins são parentes";
  5 while(my $match = $string =~ /\wo/gi){
  6         say "$& encontrado na posição ", pos($string) -
1;
  7 }

saída:

ro encontrado na posição 2
do encontrado na posição 4
bo encontrado na posição 12
```

8.5 O metacaractere âncora \b

Você percebeu que as expressões regulares têm uma certa dificuldade em encontrar a palavra exata. Por exemplo, nossa pesquisa por **b** retorna todas as palavras que contenham **b**. Os delimitadores **^ e $** verificam início e final de string. Mas uma string pode conter diversas palavras. Se desejar encontrar uma palavra exata em um string você deve utilizar as âncoras **m(\bexpressao\b**).

```
  1 #exemplo8.23
  2 use 5.12.0;
  3
  4 my $string = "Frodo e Bilbo Baggins são parentes";
  5 while(my $match = $string =~ /\bBi\b/gi){
  6         say "$& encontrado na posição ", pos($string) -
1;
  7 }
```

Este exemplo não gera saída porque não temos uma palavra exata que atenda o padrão **/\bBi\b/**. Mas se modificarmos o padrão para **/\bBilbo\b/** teremos uma saída:

```
Bilbo encontrado na posição 12
```

8.6 O que ler agora?

Expressões regulares são extremamente amplas e poderosas. Seria quase impossível explicar todas as suas funcionalidades em um único capítulo de um livro. Assim, ficam como referências para estudos posteriores:

1. As páginas da documentação do Perl, ***perldoc perlrequick*** e ***perldoc perlretut***;
2. Mastering Regular Expressions, de Jeffrey E.F. Friedl, editora O'Reilly;
3. Mastering Perl, de Brian D'Foy (capítulo 2: Advanced Regular Expressions), editora O'Reilly.

8.7 Receita especial – match aproximado

Existe uma coisa muito importante a respeito das regex que é pouco comentada em livros, mas fundamentalmente em sistemas profissionais. Às vezes precisamos de realizar match em expressões próximas. Por exemplo, é muito comum que as pessoas escrevam meu nome errado, omitindo o **h** em T**h**iago ou substituindo o **z** por s em Sanche**z.** Uma busca por padrão exato nunca encontraria meu nome, mas existe uma solução para isso. O módulo **string::Approx** permite que façamos pesquisa e substituição em valores próximos. Ele utiliza um algoritmo Fuzzy, escrito em C, para fazer as pesquisas.

```
 1 #exemplo8.24
 2 use 5.12.0;
 3 use String::Approx qw(amatch);
 4
 5 while(<DATA>) {
 6         print if amatch("Thiago");
 7 }
 8 __DATA__
 9 Thiago
10 Tiago
11 Diago
12 Iago
13 Thiogo
```

Exercícios

1. Monte uma expressão regular que valide e-mails informados pelo usuário.

2. Crie uma função **trim** que remova os espaços no final e no início de uma string informada.

3. Monte um programa onde o usuário informa um arquivo e um palavra. O programa deve pesquisar o arquivo e imprimir as linhas onde esta palavra ocorre, bem como o número da linha.

4. Monte um programa onde o usuário informa o arquivo e uma palavra para ser substituída, e a palavra que substituirá a ocorrência. Substitua a palavra e mostre a saída na tela.

Capítulo 9. Manuseando Arquivos e Outros Filehandles

Fizemos muitas coisas legais com o Perl até agora, inclusive programas que fizeram manuseio básico de arquivos com o módulo ***FileHandle***. Mas para aprender de verdade a manusear arquivos com o Perl a fim de poder entender como esse módulo e outros funcionam é preciso ir, como dizem os americanos, "**down and dirty**". Ou seja, temos que diminuir nossa abstração e fazer um pouco do serviço em baixo nível.

Primeiro, temos que levar em conta exatamente o que são os filehandles. **Um filehandle é uma variável que faz referência a um arquivo**. Um objeto do módulo ***FileHandle***, além da referência à variável, fornece uma interface mais amigável e legível das funções de mais baixo nível do Perl.

9.1 Por que manusear arquivos?

Bom, essa conversa começa com um simples fato: se você quiser guardar as informações para acesso futuro, análise ou relatórios elas precisam ser armazenadas. Hoje existem as soluções em banco de dados, mas muitas vezes as informações são simples demais para exigir a necessidade de um engenho de banco de dados.

Outro fator importante... se você conhece o Linux, já deve ter reparado que tudo no Linux é baseado em arquivos. As interfaces de hardware, de sockets e de processos funcionam como arquivos. Aliás, um arquivo é uma forma muito simples e básica de comunicação entre processos (IPC). Você pode utilizar um arquivo de texto simples como interface básica para troca de informações entre dois processos. Além de que módulos CPAN como ***GD*** e ***Imagemagick*** usam filehandles em modo binário para manusear arquivos de imagem. E você pode um dia precisar de um ***filehandle*** binário para acessar as informações de algum tipo de arquivo que ainda não tenha um módulo Perl (espero que não, mas é perfeitamente possível).

Vamos começar criando um filehandle para um arquivo de texto simples.

9.2 Criando um filehandl

A função básica do Perl para criar um filehandle é ***open <variável>, <modo>, <arquivo>***, onde <variável> é o nome da variável que será a referência ao arquivo, <modo> se refere às operações que serão feitas no arquivo e <arquivo> é o nome do arquivo, incluindo extensão e caminho, se ele não estiver na pasta local.

9.2.1 Modos de acesso

Um filehandle pode abrir um arquivo para leitura apenas, para escrita apenas ou para leitura e escrita. E por que não abrir sempre para leitura e escrita? Um dos princípios da segurança é: forneça apenas as credenciais necessárias. Se o processo vai apenas adicionar conteúdo a um arquivo, para que abri-lo com permissão de leitura? Isso seria apenas um possível ponto a ser explorado por um hacker.

Símbolo	Modo	Descrição
<	Leitura	Abre um arquivo **para leitura apenas.** Se o arquivo não existe, retorna um erro.
>	Escrita	Abre o arquivo **para escrita apenas.** Se o arquivo não existir, ele será criado. Se existe, os dados anteriores são perdidos.
>>	Apêndice	**Abre o arquivo para escrita apenas**, conserva os dados existentes e **adiciona novos dados ao final do arquivo**, preservando os dados anteriores. Se o arquivo não existir, ele é criado.
+<	Leitura e Escrita	**Lê e grava dados no arquivo desde o início do arquivo.** Se o arquivo não existe, gera erro. Se existir, os dados são preservados ou alterados programaticamente.
+>	Leitura e Escrita	**Lê e grava dados no arquivo desde o início do arquivo.** Se o arquivo não existe, cria o arquivo. Se existir, os dados anteriores serão perdidos.
+>>	Leitura e Apêndice	**Lê desde o início e grava dados apenas no fim do arquivo.** Se o arquivo não existe, cria o arquivo. Se existir, os dados anteriores não serão perdidos.

9.2.2 Lendo os dados de um arquivo

Para ler um arquivo o caminho é bem simples. Abra o arquivo com um filehandle e use uma das funções: ***getc***, para ler caractere por caractere, ***readline*** para ler linha a linha ou o operador diamante <> também para ler linha a linha. Vamos tomar como exemplo um arquivo CSV cujas linhas seriam como linhas de uma tabela e os caracteres ponto-e-vírgula (;) serão os separadores de colunas – um formato muito comum em arquivos de texto.

"Sobrenome";"Nome";"Empresa";"Contato";

"Longa";"Perna";"Warner";"pernalonga@warner.com";

"Colméia";"Zé";"Hanna-Barbera";"ze_colmeia@hannabarbera.com";

"Flintstone";"Wilma";"Hanna-Barbera";"wilminha@hannabarbera.com";

"Troca-letra";"Hortelino";"Warner";"hortelino@warner.com";

Agora vamos abrir este arquivo com o script a seguir:

```
  1 #exemplo10.1
  2 use 5.12.0;
  3 use constant {PATH => "C:\\Users\\thiago\\Desktop\\
Thiago\\Curso Perl\\",
  4               ARQUIVO => "enderecos.csv",
  5               LEITURA => '<'};
  6
  7 open (my $file,LEITURA,PATH.ARQUIVO) or die "Erro ao
abrir o arquivo: $!";
  8
  9 while ($_ = readline $file){
 10 #A função readline lê uma linha de um arquivo de texto
e retorna
 11 #como um string.
 12     chomp (my @line = split /;/, $_);
 13     map{s/"//g; printf "%15s\t", $_} @line;
 14 #s/EXPRESSÃO/STRING/ substitui o primeiro elemento de
string que combinar
 15 #com a regex em EXPRESSÃO por STRING. O modificador g
indica que todos
```

```
 16 #os matchs devem ser modificados.
 17     say "";
 18 }

saída:

Sobrenome   Nome       Empresa        Contato
Longa       Perna      Warner         pernalonga@warner.com
Colméia     Zé         Hanna-Barbera  ze_colmeia@hannabarbera.com
Flintstone  Wilma      Hanna-Barbera  wilminha@hannabarbera.com
Troca-tapa  Hortelino  Warner         hortelino@warner.com
```

Porém, ***while ($_ = readline $file)*** pode ser substituída por ***while(<$file>)***. O operador diamante lê uma linha por vez do arquivo e envia para a variável implícita **$_.** Assim, nosso pequeno loop de forma mais abreviada ficaria:

```
  9 while(<$file>){
 10  chomp (my @line = split /;/);#$_ também é argumento
implícito de split
 11  map{s/"//g; printf "%15s\t",$_} @line;
 12  say "";
 13 }
```

Usamos um arquivo CSV cujos elementos são separados por ponto-e-vírgula e os textos definidos por aspas ("). A primeira linha representa o título das colunas e cada linha representa um registro de uma tabela. Essas definições significam um padrão de arquivo e é importante o conhecimento dos padrões que definem como os dados são armazenados nos arquivos que vamos utilizar, principalmente se formos utilizar arquivos binários.

9.2.3 Abrindo arquivos para adicionar apêndice

Às vezes queremos adicionar algo apenas ao final de um arquivo, preservando os dados anteriores – um sistema de logs, por exemplo. Então arquivos são abertos apenas para adicionar os últimos eventos ao final do arquivo.

```
  1 #exemplo9.2
  2 use 5.012;#outra forma de fazer use 5.12.0;
  3
  4 use constant {PATH => "C:\\Users\\thiago\\Desktop\\
Thiago\\Curso Perl\\",
  5               ARQUIVO => "enderecos.csv",
  6               APPEND => '>>'};
  7
  8 #Abre o arquivo para adicionar apêndice apenas
  9 open (my $file,APPEND,PATH.ARQUIVO) or die "Erro ao
abrir o arquivo: $!";
 10 say "Digite as informações do contato conforme soli-
tadas.";
 11
 12 #Utilizo o array com os nomes das colunas para economi-
zar digitação. Assim,
 13 #com apenas uma função print e um loop eu posso soli-
tar e gravar
 14 #todas as informações
 15 my @dados = qw/Sobrenome Nome Empresa Contato/;
 16
 17 for (@dados){
 18     print "Digite o $_ do contato: ";
 19     chomp (my $dado = <STDIN>);
 20 #adicionar uma linha para cada novo registro
 21     print $file "\n" if $_ eq 'Sobrenome';
 22     print $file qq/"$dado";/;
 23 }
 24
 25 close $file; #fecho o arquivo
```

saída:

Digite as informações do contato confome solicitadas.
Digite o Sobrenome do contato: Charmosa
Digite o Nome do contato: ***Penélope***
Digite o Empresa do contato: ***Hanna Barbera***
Digite o Contato do contato: ***penelope@hannabarbera.com***

Vai modificar o arquivo, adicionando a linha "Charmosa";"Penelope";"Hanna--Barbera";"penelope@hannabarbera.com"; como o último registro do arquivo.

9.2.4 Abrindo arquivo para leitura e escrita

Podemos abrir um arquivo, ler todo seu conteúdo e então adicionar outros dados ao arquivo. A vantagem dos arquivos de texto reside no seu tamanho. A maioria é relativamente pequena e não é problemático para os computadores modernos carregar todo seu conteúdo para a memória e processá-lo de uma única vez. Mesmo um arquivo de alguns milhares de bytes seria processado rapidamente e você não teria que se preocupar com posicionamento do cursor, como veremos mais adiante. Para exemplo, vou carregar todo meu arquivo anterior para um array, ordená-lo de forma alfabética, apagar o conteúdo anterior e sobrescrever o arquivo todo. Minha única preocupação com o cursor será para definir o fim do cabeçalho, para não sobrescrevê-lo.

```
  1 #exemplo9.3
  2 use 5.012;
  3 use constant {PATH => "C:\\Users\\thiago\\Desktop\\
Thiago\\Curso Perl\\",
  4               ARQUIVO => "enderecos.csv",
  5               LEITURA => '+<'};
  6
  7 open (my $file,LEITURA,PATH.ARQUIVO) or die "Erro ao
abrir o arquivo: $!";
  8 #tell retorna a posição (em bytes) do cursor no fi-
lehandle.No início é zero
  9 my $pos = tell $file;
 10 say $pos;
 11
 12 #lê o header
 13 my $header = <$file>;
 14 #Depois de ler o header o cursor tem nova posição,
 15 # o início da próxima linha
 16 my $pos_header = tell $file;
 17 #Veja que a posição do cursor é o caractere após o fi-
nal do cabeçalho.
 18 #Assim, se $header tem 40 bytes, o cursor está no iní-
cio do byte 41.
```

```
 19 say $pos_header;
 20 say scalar split //, $header;
 21
 22 #Agora eu carrego o restante do arquivo para um array e
organizo o array
 23 #em ordem alfabética.
 24 chomp(my @linhas = <$file>);
 25 @linhas = sort @linhas;
 26
 27 #A função seek reposiciona o cursor para a nova posição
informada, em bytes
 28 #seek FILEHANDLE,POSITION,WHENCE
 29 #onde WHENCE = 0 (Posição em relação ao início do ar-
quivo)
 30 #            = 1 (Posição em relação ao cursor);
 31 #            = 2 (Posição em relação ao EOF);
 32
 33 seek ($file, $pos_header, 0) or die "Não foi possível
mover o cursor. $!";
 34 say $file $_ for @linhas;
 35 close $file;

Arquivo Modificado:

"Sobrenome";"Nome";"Empresa";"Contato";
"Charmosa";"Penelope";"Hanna-Barbera";"penelope@hannabarbe-
ra.com";
"Colmeia";"Zé";"Hanna-Barbera";"ze_colmeia@hannabarbera.
com";
"Flintstone";"Wilma";"Hanna-Barbera";"wilminha@hannabarbe-
ra.com";
"Longa";"Perna";"Warner";"pernalonga@warner.com";
"Troca-letra";"Hortelino";"Warner";"hortelino@warner.com";
```

Neste exemplo nos preocupamos com o cursor apenas para posicioná-lo após o cabeçalho da função. Para manusear o cursor o Perl fornece as funções ***seek*** e ***tell***, onde ***tell*** informa a posição do cursor em um filehandle e ***seek*** reposiciona o cursor. Assim, podemos nos mover pelo arquivo arbitrariamente. Note que os

dados escritos a partir de uma posição são sobrepostos, e para não perder dados carreguei todo o arquivo, com exceção do cabeçalho, para um array.

9.3 A função select

Você já está cansado de saber o tamanho da minha preguiça, e como eu odeio digitar coisas repetidas. Então a função ***select*** é mais um pequeno truque do Perl que poupa um pouco de digitação, tornando seu argumento, o filehandle, o padrão para saída de dados, no lugar de STDOUT. Assim, quando você for inserir diversos dados em um filehandle, em vez de especificar em diversas funções o filehandle que você está trabalhando, use a função ***select <filehandle>.***

```
  1 #exemplo9.4
  2 use 5.012;
  3 use constant {  PATH => "C:\\Users\\thiago\\Desktop\\
Thiago\\Curso Perl\\",
  4                 ARQUIVO => "enderecos.csv",
  5                 APPEND => '>>'};
  6 open (my $file, APPEND, PATH.ARQUIVO) or die "Erro ao
abrir o arquivo: $!";
  7 say "Digite as informações do contato conforme solici-
tadas.";
  8 my @dados = qw/Sobrenome Nome Empresa Contato/;
  9 select $file; #torna Select o default STDOUT
 10 for (@dados){
 11         print STDOUT "Digite o $_ do contato: ";
 12         chomp (my $dado = <STDIN>);
 13         print "\n" if $_ eq 'Sobrenome'; #não preciso
especificar $file
 14         print qq/"$dado";/; #não preciso especificar
$file
 15 }
 16 close; #não preciso especificar $file
```

9.4 Arquivos binários

Para trabalhar com arquivos binários o processo é basicamente o mesmo, porém existem adicionais: as funções ***pack*** e ***unpack.*** A função ***pack*** recebe como argumentos uma máscara e uma variável escalar e retorna uma string cujos valores da variável foram modelados de acordo com a máscara. Porém, o valor de retorno utiliza a representação binária do seu sistema, não podendo ser visualizado com ***print*** ou ***say***. Por exemplo:

```
  1 #exemplo9.5
  2 use 5.012;
  3
  4 my $var = pack  'I', 255;
  5 #Variável representa um valor binário que não pode ser
usado diretamente!
  6 say $var;
  7 $var = unpack 'A4', 255; #reverte o processo
  8 say $var;

saída no Windows:

ÿ
255

Saída no Linux/BSD:

255
```

Mas o que aconteceu? O template ***I*** indica um tipo de dado inteiro sem sinal, que no meu sistema tem 4 bytes. Então o Perl deixou de tratar minha $var como um escalar comum. Ela agora é um dado de baixo nível, inteiro sem sinal com 4 bytes de extensão. Quando eu desempacotei com ***unpack*** eu solicitei uma representação em um string de 4 bytes (4 caracteres ASCII), um para cada byte. Você pode obter uma lista de todos os templates em ***perldoc pack*** e pode ler mais sobre estas funções estranhas em ***perldoc perlpacktut.***

E por que eu estou falando destas funções mesmo? E de programação baixo nível? Eu não quero complicar sua vida nem assustá-lo. Apenas desejo que você perce-

ba que programar em baixo nível é um verdadeiro sacrifício. E que você pense bem antes de tentar fazê-lo sem necessidade. Você vai realmente precisar se surgir algum padrão de arquivo novo no mercado, que não tenha ainda um módulo Perl ou biblioteca C para fazer esse trabalho sujo por você. Bom, continuando... vamos ler o cabeçalho de um arquivo binário qualquer. Escolhi um arquivo PNG porque tem uma especificação simples e é fácil de entender. Veja um exemplo:

```
  1 #exemplo9.6
  2 use 5.012;
  3 use constant {PATH => "C:\\Users\\thiago\\Pictures\\",
  4               ARQUIVO => "mysql1.png",
  5               LEITURA => '<'};
  6 #o formato PNG especifica que o arquivo PNG deve ini-
ciar com 8 octetos
  7 #cujos valores convertidos para decimal serão: 137 80
78 71 13 10 26 10.
  8 #Isso define a assinatura do arquivo.
  9
 10 open my $teste, "<", PATH . ARQUIVO or die "Could not
open $!";
 11
 12 #binmode informa ao Perl que o filehandle $teste é bi-
nário
 13 binmode $teste;
 14
 15 for (1..8){
 16         my $byte = getc $teste;
 17         #Aqui eu desempacoto o valor de baixo nível
para uma representação
 18         #decimal de 8 bits sem sinal
 19         my $byte_unpacked = unpack ('C', $byte);
 20         say $byte_unpacked;
 21 }
 22
 23 #Seria muito demorado ler, em um loop, byte a byte de
um arquivo binário
 24 #longo, por isso a função read lê, de um dado filehand-
le, a quantia de
```

```
 25 #bytes informados e armazena os dados em uma variável
de buffering
 26 seek $teste, 1, 0;      #reposiciono o cursor
 27
 28 #leio 3 bytes de uma vez, posso ler o quanto minha RAM
permitir
 29 read ($teste, my $buff, 3);
 30
 31 #O header de um arquivo PNG tem uma característica in-
teressante: se lermos
 32 #do segundo ao quarto byte do arquivo binário e conver-
termos os valores
 33 #para o formato texto teremos a palavra PNG
 34 my @unpacked_header = unpack 'C3', $buff;
 35 print chr $_ for @unpacked_header;
 36 close $teste;

saída:

137
80
78
71
13
10
26
10
PNG
```

Você não vai ter que se preocupar muito com ***pack/unpack e binmode*** enquanto programador web ou sysadmin. Mas se pretender utilizar IPC (porém o CPAN tem muitas facilidades para isso) ou criar módulos para disponibilizar no CPAN para padrões de arquivo novos, pode precisar destas funções...

9.5 File locking

Existem situações em que mais de um processo ou programa podem solicitar acesso a um arquivo ao mesmo tempo. Isso é muito inconveniente e pode causar problemas de segurança e de integridade da informação utilizada. Para isso o Perl

fornece a função ***flock,*** que informa para um outro processo que o arquivo em questão está em uso. A função ***flock*** tem dois argumentos: o primeiro é o filehandle do arquivo a ser bloqueado e o segundo é o modo. Os modos de flock são:

Nome	Valor	Descrição
lock_sh	1	Cria um bloqueio compartilhado.
lock_ex	2	Cria um bloqueio exclusivo.
lock_nb	4	Cria um bloqueio que apenas alerta.
lock_un	8	Desbloqueia.

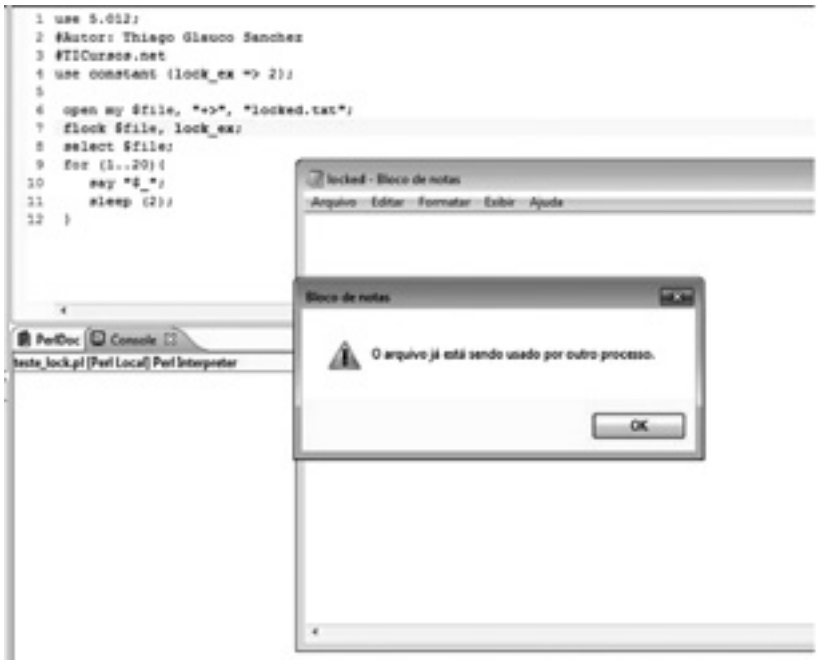
Um detalhe importante sobre ***flock*** é que ele não impede que o arquivo seja alterado, apenas gera um alerta. Outros programas que fazem verificação de uso do arquivo veem se ele está ou não sendo utilizado. Então você deve implementar ***flock*** em programas que utilizam multiprocessos ou cujos arquivos podem ser acessados de outras formas. Se um programa que não utilizar a função ***flock*** tentar escrever em seu arquivo, então os dados serão alterados. Mas isso não é uma limitação do Perl e sim dos sistemas operacionais. Mais infomações sobre ***flock*** podem ser encontradas nas páginas dos manuais com os comandos ***man flock*** (linux e bsd) e ***perldoc -f flock.***

Veja um exemplo de utilização:

```
 1 #exemplo9.7
 2 use 5.012;
 3
 4 #crio as constantes de flock
 5 use constant {lock_ex => 2, lock_un => 8};
 6
 7 #crio o filehandle
 8 open my $file, "+>", "locked.txt";
 9 #bloqueio o arquivo
10 flock $file, lock_ex;
11 select $file;
12
13 for (1..20){
14    say "$_";
15    sleep (2); #delay adicional para podermos visualizar
```

```
o efeito de flock
 16 }
 17
 18 flock $file, lock_un;
 19 close;
```

Aqui eu tentei alterar o arquivo bloqueado com o Bloco de notas:



O Bloco de notas verifica se o arquivo está sendo utilizado por outro processo antes de gravar, gerando o erro visto.

9.6 Testes de arquivos e diretórios

Antes de finalizar nosso papo sobre arquivos, vamos falar sobre os testes de arquivo. O Perl possui alguns operadores que fazem verificações nos arquivos, como tamanho, data de criação, tipo de arquivo, etc. Estas verificações são úteis porque ajudam na administração de sistemas ou ainda para determinar o tipo de filehandle a ser criado.

Operador de teste	Utilidade
`-r`	O arquivo ou diretório pode ser lido por este usuário/grupo efetivo
`-w`	O arquivo ou diretório pode ser alterado por este usuário/grupo efetivo
`-x`	O arquivo ou diretório pode ser executado por este usuário/grupo efetivo
`-o`	Este usuário/grupo efetivo tem permissão de proprietário no arquivo/diretório
`-R`	O arquivo ou diretório pode ser lido por este usuário/grupo real
`-W`	O arquivo ou diretório pode ser modificado por este usuário/grupo real
`-X`	O arquivo ou diretório pode ser executado por este usuário/grupo real
`-O`	Este usuário/grupo real tem permissão de proprietário no arquivo
`-e`	Arquivo/diretório existe
`-z`	Arquivo existe e tem tamanho de zero bytes (sempre retorna falso se utilizado em diretórios)
`-s`	Arquivo existe e tem tamanho zero bytes ou mais. Retorna o tamanho do arquivo em bytes ou falso se o arquivo não existe
`-f`	Verifica se é um arquivo de texto
`-d`	Verifica se é um diretório
`-l`	Verifica se é um link simbólico
`-S`	Verifica se é um socket (lembre, tudo é um arquivo...)
`-p`	Verifica se é um pipe (IPC)
`-b`	Arquivo representa um dispositivo de armazenamento em blocos (como um disco)
`-c`	Arquivo representa um dispositivo de hardware – um modem ou outro tipo de dispositivo de entrada e saída
`-t`	Este filehandle é um terminal TTY
`-T`	Parece um arquivo de texto
`-B`	Parece um arquivo binário
`-M`	Número de dias desde a última modificação
`-A`	Número de dias desde o último acesso

Podemos, por exemplo, testar se um arquivo existe e o seu tamanho e depois decidir qual modo de filehandle criar. A seguir temos um trecho de código que exemplifica um uso dos testes de arquivo:

```
  1 #exemplo9.8
  2 say "Digite o caminho completo do arquivo que deseja
abrir: ";
  3 chomp (my $arquivo = <STDIN>);
  4 if ((-z $arquivo) > 0){
  5         print "Arquivo já existe e possuí dados. O que
deseja fazer?\n",
  6                 "1: Abrir para leitura\n",
  7                 "2: Abrir para adicionar dados ao final
(apendice)\n",
  8                 "3: Abrir para leitura e apêndice\n",
  9                 "4: Ler e modificar um registro\n",
 10                 "Digite o número da opção desejada: ";
 11         chomp (my $escolha = <STDIN>);
 12         given ($escolha) {
 13                 when 1 {leitura()}
 14                 when 2 {append()}
 15                 when 3 {le_append()}
 16                 when 4 {ler_modificar()}
 17                 default {cria_arquivo()}
 18         }
 19 }
```

Outro exemplo:

```
  1 #exemplo9.9
  2 use 5.012;
  3
  4 my $dir;
  5 if (-e ".."){
  6         #o mesmo que open, só que se aplica a um dire-
tório
  7         #"." é o diretório local
```

```
  8         opendir $dir, ".";
  9 }
 10 else {
 11         say "Arquivo não existe!!!";
 12         exit (0); #nunca vai executar pois sempre exis-
te um diretório "."
 13 }
 14
 15 #readdir retorna o nome relativo de cada arquivo do di-
retório.
 16 while (readdir $dir){
 17         next if $_ eq "." or $_ eq "..";
 18         printf "Arquivo %s tem %s bytes\n", $_, -s $_;
 19 }
```

saída:

```
Arquivo .project tem 379 bytes
Arquivo autoload.pl tem 204 bytes
Arquivo binary_search.pl tem 494 bytes
Arquivo circular.pl tem 227 bytes
Arquivo circular_list.pl tem 324 bytes
Arquivo duvida tem 0 bytes
Arquivo duvida.pl tem 200 bytes
Arquivo duvida2.pl tem 633 bytes
Arquivo duvidas.pl tem 258 bytes
Arquivo exemplo3.pl tem 77 bytes
Arquivo Exemplos.pl tem 1047 bytes
Arquivo exemplo_file_handle.pl tem 273 bytes
#Saída omitida.
```

No meu FreeBSD, executei este comando:

```
# perl -e 'print qq(E um dispositivo) if (-c qq(/dev/acd0)'
E um dispositivo
```

Aprendemos a solicitar entradas dos usuários através de uma informação via ***print*** ou **say** e lendo a informação via ***STDIN***. Porém comandos Unix-Like úteis fazem mais que isto. Bem mais. E para continuar sua filosofia de manter as coisas simples em sua simplicidade e tornar as coisas difíceis possíveis, o Perl nos fornece a variável especial @ARGV, que recebe automaticamente argumentos de linha de comando. Mas para tornar fácil criar projetos complexos com linha de comando, foram criados os módulos ***Getopt::Std e Getopt::Long***.

9.7 A variável @ARGV

A variável @ARGV é um array, podendo ser então manipulado com qualquer técnica de array vista até agora. Podemos pensar no @ARGV como o @_ de ***main.*** Ele recebe a lista de argumentos e utiliza apenas os que forem solicitados em seu programa, dificultando que um hacker consiga corromper seu programa com entradas fuzzy. Para entender bem o @ARGV vamos criar um programa de ***echo.*** Mas antes, para deixar as coisas mais interessantes, vamos aprender a compilar um programa Perl, assim podemos rodar nossos programas em máquinas que não têm o interpretador Perl instalado.

Para isso, instale o módulo ***PAR::Packer***, que traz consigo um executável, o ***pp***, que permite a compilação de uma aplicação Perl em um executável único. Para compilar um programa Perl com ***pp*** basta usar:

```
pp -o <arquivo_executavel> <arquivo.pl>
```

Então vamos criar o programa ***echo.pl***:

```
  1 #exemplo9.10
  2 use 5.012;
  3
  4 chomp $ARGV[$#ARGV];
  5
  6 #primeiro verificamos se $ARGV[0] é um switch de coman-
do
  7 my $switch = shift if ($ARGV[0] =~ /^-/); #O array pa-
drão de main é @ARGV
  8 my $str = join " ",@ARGV;
```

```
 9 chomp $str;
10
11 given($switch){
12         when($_ eq "-e"){
13 #As entradas de caracteres de escape em @ARGV têm uma
barra invertida
14 #adicionada a elas. Assim, se você colocar no seu
string de entrada o
15 #caractere \n, o Perl irá automaticamente substituí-lo
por uma barra dupla
16 #\\n. No manual da função echo (man echo), vemos que a
opção -e permite a
17 #utilização de caracteres de escape. Então checamos e
substituímos todos
18 #os caracteres de espaço permitidos por echo.
19                 $str =~ s/\\n/\n/gsi;
20                 $str =~ s/\\t/\t/gsi;
21                 $str =~ s/\\f/\f/gsi;
22                 $str =~ s/\\r/\r/gsi;
23                 $str =~ s/\\a/\a/gsi;
24                 $str =~ s/\\v/\n\n\n\n/gsi;
25                 $str =~ s/\\b/\b/gsi;
26                 $str =~ s/\\\\/\\/gsi;
27                 say $str;
28                 exit (0);
29         }
30         when($_ eq "-n"){
31                 print $str;
32         }
33         default {say "@ARGV"}
34 }
```

E vamos compilá-lo. ***pp -o echo.exe echo.pl***. Agora temos um echo executável, que poderemos, por exemplo, utilizar em máquinas Windows que não tenham o Perl instalado.

```
C:>echo.exe -e teste de\n echo\v\tcom tabulacao horizontal
e vertical!!!\a\nE beep\a
teste
de
echo

     com tabulacao horizontal e
vertical!!! E beep
```

Caso eu queira que meu programa permita utilização de diversos switches de comando, conforme definido no padrão POSIX, a complexidade irá aumentar. Veja mais um exemplo simples e como a complexidade aumenta conforme aumenta o número de switches:

```
    1 #exemplo9.11
  2 use 5.12.0;
  3
  4 #programa mk.pl
  5 #Hash::Util qw/lock_keys/; é um módulo que ajuda a pre-
definir quais chaves
  6 # e/ou valores um hash deve ter, tornando-o estático.
  7 use Hash::Util qw/lock_keys/;
  8 lock_keys (my %args_opt, qw/-dir -file/);
  9
 10 %args_opt = @ARGV;
 11 #A opção -dir cria um diretório informado
 12 if (defined $args_opt{-dir}){
 13         if (not -e $args_opt{-dir}){
 14                 mkdir $args_opt{-dir}, 075
 15         }
 16         else{ warn "Diretório $args_opt{-dir} já exis-
te"; }
 17 }
 18 #A opção -file cria um arquivo em branco, tal qual o
comando touch.
 19 elsif (defined $args_opt{-file}){
 20         if (not -e $args_opt{-file}){
```

```
 21                 open my $file, ">", $args_opt{-file}
 22         }
 23         else{ warn "Arquivo $args_opt{-file} já exis-
te"; }
 24 }
 25 else { warn "usage md.pl -dir dirname | -file file-
name"}

saída:

c:>perl mk.pl -dir meu_dir
Diretorio meu_dir já existe at mk.pl line 18.

c:>perl mk.pl -dir meu_dir
Diretorio meu_dir já existe at mk.pl line 18.
c:>perl mk.pl -dir meu_outro_dir
c:>perl mk.pl -dir meu_outro_dir
Diretorio meu_outro_dir já existe at mk.pl line 18.

c:>perl mk.pl -f testefile
Attempt to access disallowed key '-f' in a restricted hash
at mk.pl line 9.

c:>perl mk.pl -f testefile
Attempt to access disallowed key '-f' in a restricted hash
at mk.pl line 9. c:>perl mk.pl -file testefile
c:>perl mk.pl -file testefile
Arquivo testefile já existe at mk.pl line 26.

c:>perl mk.pl
usage md.pl -dir dirname | -file filename at mk.pl line 29.
```

9.8 O módulo Getopt::Std

Para manter a facilidade em criar programas de linha de comando, foram desenvolvidos os módulos ***Getopt::Std e Getopt::Long***. O ***Getopt::Std*** permite criarmos comandos com relativa simplicidade, porém com menos recursos que o ***Getopt::Long***. O ***Getopt::Std*** nos dá a função ***getopts(<switches>, $hash_ref)***, que permite usar-

mos diversas opções de argumento para o comando. Por padrão elas são booleanas. E a função ***getopts*** cria uma chave com valor 1 em ***%hash_ref*** para cada switch ativada pelo usuário. ***Getopt::Std*** permite que o programador defina duas funções, HELP_MESSAGE e VERSION_MESSAGE, que são executadas se o usuário utilizar as opções --help e --version. Veja um programa de exemplo, getopt.pl:

```
  1 #exemplo9.12
  2 use 5.12.0;
  3 use Getopt::Std;
  4
  5 #$VERSION é requerida por Getopt::Std para informar a
versão do comando
  6 #para o usuário:
  7 my $VERSION = 1.0;
  8
  9 #%opts irá armazenar as switches utilizadas. Neste pro-
grama as switches de
 10 #comando são armazenadas como booleanos my %opts;
 11 getopts('neE', \%opts);
 12 say $_," => ", $opts{$_} for keys %opts;
 13 #@ARGV mantém-se com um valor. Apenas as switches são
removidas
 14 print "@ARGV";
 15
 16 sub VERSION_MESSAGE{
 17 #Se o usuário informar --version, a função VERSION_MES-
SAGE é evocada.
 18     say "This is my teste of Getopt::Std version 1.0";
 19 }
 20 sub HELP_MESSAGE{
 21 #Se o usuário informar --help, a função HELP_MESSAGE é
evocada por padrão.
 22     say "Usage: $0 -(n|e|E) message";
 23 }

saídas:

>perl getopt.pl -ne Teste
e => 1
n => 1
```

```
Teste
>perl getopt.pl -n Teste
n => 1
Teste
>perl getopt.pl -e Teste
e => 1
Teste
>perl getopt.pl Teste
Teste
>perl getopt.pl --ste
Unknown option: -
Unknown option: s
Unknown option: t
 e => 1

>perl getopt.pl --version
This is my teste of Getopt::Std version 1.0
[Now continuing due to backward compatibility and excessive
paranoia.
See ``perldoc Getopt::Std'' about $Getopt::Std::STANDARD_
HELP_VERSION.]

>perl getopt.pl --help
This is my teste of Getopt::Std version 1.0
Usage: getopt.pl -(n|e|E) message
[Now continuing due to backward compatibility and excessive
paranoia.
See ``perldoc Getopt::Std'' about
$Getopt::Std::STANDARD_HELP_VERSION.]
```

9.8.1 O novo echo

Agora podemos criar uma versão robusta de um comando echo sem tanto trabalho de programação, com um help razoável para os usuários e opções mais profissionais e semelhantes ao echo real:

```
  1 #exemplo9.13
  2 use 5.12.0;
  3 use Getopt::Std;
  4
  5 my $VERSION = 0.1; #Define a versão da aplicação, exi-
gido pelo Getopt::Std
  6 my %opts; #define as opções booleanas de echo
  7 getopts('neE', \%opts);
  8
  9 my $str = join " ",@ARGV;
 10 chomp $str;
 11
 12 if (defined $opts{e} and defined $opts{E}){
 13   warn "Conflito: -E desabilita caracteres de escape,
-e habilita\n";
 14  HELP_MESSAGE();
 15 }
 16
 17 #Modifica $str para atender os caracteres de escape de-
finidos em man echo.
 18 if(defined $opts{e}){
 19  $str =~ s/\\n/\n/gsi;
 20  $str =~ s/\\t/\t/gsi;
 21  $str =~ s/\\f/\f/gsi;
 22  $str =~ s/\\r/\r/gsi;
 23  $str =~ s/\\a/\a/gsi;
 24  $str =~ s/\\v/\n\n\n\n/gsi;
 25  $str =~ s/\\b/\b/gsi;
 26  $str =~ s/\\\\/\\/gsi;
 27 }
 28
 29 if (defined -n) {print $str}
 30 else {say $str}
 31
 32 sub VERSION_MESSAGE{
 33  say "Este é o meu comando echo particular, versão
0.1";
 34 }
```

```
 35
 36 sub HELP_MESSAGE{
 38 NAME
 39 echo - display a line of text
 40
 41 SYNOPSIS
 42 echo [OPÇÃO]... [STRING]...
 43
 44
 45
 46 DESCRIPTION
 47
 48 Ecoa a(s) STRING(s) para a saída padrão.
 49
 50 -n      não ecoa a saída de caracteres de nova linha
 51
 52 -e      habilita a interpretação de caracteres de esca-
pe
 53
 54 -E      desabilita a interpretação de sequências nas
strings
 55
 56 --help  mostra a ajuda e sai do programa
 57 HELP
 58 }
```

saídas:

```
$ perl exemplo9.13.pl -e "Ola\n"
Ola
$ perl exemplo9.13.pl --version
Este é o meu comando echo particular, versão 0.1
```

Apesar de usarmos switches de comando apenas em modo booleano, podemos criar switches com valor associado. Veja o ***sping.pl***:

9.8.2 sping.pl

Para reforçar nosso trabalho, vamos ver agora mais um exemplo de programa de linha de comando. O ***sping.pl*** utiliza o módulo Net::Ping para testar o uptime de diversas portas e hosts. O comando possui tanto switches com argumentos (-p, -h, -t, -n) quanto booleanas (- r). Observe que realizei um extenso teste nos valores de entrada.

```
  1 #exemplo9.14
  2 use 5.12.0;
  3 use Net::Ping;
  4 use Getopt::Std;
  5
  6 my $VERSION = 0.9;
  7 my %opts;
  8 getopts("p:h:t:n:r", \%opts);
  9
 10 #variáveis usadas pelo programa:
 11 #$time = tempo de espera. Switch -t
 12 #@ports_to_test = array de portas TCP para verificar
estado. Switch -p
 13 #@hosts_to_test = array de IPs/hostnames para verifi-
car. Switch -h
 14 #$number_os_tests = número de testes a efetuar por host
e porta. Switch -n
 15 my ($time, @ports_to_test, @hosts_to_test, $number_of_
tests);
 16
 17 #verifica e define tempo limite para teste
 18 if(defined $opts{t}){
 19     if ($opts{t} =~ /^\d+$/){
 20             $time = $opts{t};
 21     }
 22     else{
 23             warn "-t is not ainteger";
 24             HELP_MESSAGE() and die;
 25     }
 26
```

```
 27 }
 28 else {
 29     $time = 3;
 30 }
 31
 32 #verifica e define número de testes a serem executados
 33 if (defined $opts{n}){
 34     if ($opts{n} =~ /^\d+$/){
 35             $number_of_tests = $opts{n};
 36     }
 37     else{
 38             warn "-n is not a integer"; HELP_MESSAGE()
and die;
 39     }
 40 }
 41 else{
 42     $number_of_tests = 1;
 43 }
 44
 45 #verifica e define as portas que serão testadas
 46 if (defined $opts{p}){
 47     @ports_to_test = split ',', $opts{p};
 48     for my $ports(@ports_to_test){
 49             if (not $ports =~ /^\d+$/ ){
 50                     warn "-p (tcp port) must be a
number\n";
 51                     HELP_MESSAGE() and die;
 52             }
 53     }
 54 }
 55 else{
 56     @ports_to_test = (80);
 57 }
 58
 59 #verifica e define os hosts que serão testados
 60
 61 if (defined $opts{h}){
 62     @hosts_to_test = split ',', $opts{h};
```

```
 63     for my $hosts(@hosts_to_test){
 64     if(not($hosts =~ /^(\d{1,3}\.){3}(\d{1,3})$/ or
 65            $hosts =~ /^\w+\.?\w+/)){
 66        warn "-h must be a host name or ip address\n";
 67        HELP_MESSAGE() and die; } }
 68 }
 69 else {
 70        warn "Needs a host to test"; HELP_MESSAGE() and
die;
 71 }
 72
 73 #usuário solicita repetições indefinidas;
 74 if($opts{r}){
 75     while (1){
 76         test_loop(hosts => \@hosts_to_test, ports => \@
ports_to_test);
 77     }
 78 }
 79
 80 #usuário especificou repetições:
 81 for (my $i = 1; $i <= $number_of_tests; $i++){
 82     test_loop(hosts => \@hosts_to_test, ports => \@
ports_to_test);
 83 }
 84
 85 sub VERSION_MESSAGE{
 86 #Mensagem de versão exibida caso solicitado pela opção
--version
 87     say "My special ping test version 0.9";
 88 }
 89
 90 sub HELP_MESSAGE{
 91 #Mensagem de ajuda exibida caso solicitado pela opção
--help
 92 say <<HELP;
 93 This is my special ping test, now running on version
$VERSION
 94 for upness and response time testing of hosts
```

```
 95 and tcp ports
 96
 97 USAGE:
 98 sping.pl [option][value]
 99
100 OPTIONS:
101 -h: host or list of hosts to ping (required)
101 -h: host or list of hosts to ping (required)
102 -p: port or list of ports to test, if ommited sping
will send a TCP ECHO
103 -t: time, in seconds, to wait the echo reply. Default
is 3 sec.
104 -n: number of tests to send, if ommited sping will send
one test.
105 -r: send tests until user stops the process.
106 HELP
107 }
108
109 sub ping{
110 #A função ping efetivamente testa o host e a porta.
111     my $ping = Net::Ping->new();
112     $ping->port_number($_[1]);
113     $ping->hires();
114     my @return = $ping->ping($_[0], $_[2]);
115     $ping->close();
116     return @return;
117 }
118
119 sub test_loop{
120 #Executa um loop de testes, sendo um teste para cada
par host:porta
121     my %args = @_;
122     my @hosts = @{$args{hosts}};
123     my @ports = @{$args{ports}};
124
125     for my $host (@hosts){
126        for my $port(@ports_to_test){
127            my ($ret, $answer_time, $ip) = ping($host,
```

```
$port, $time);
128             print "Host $ip:$port answered in $answer_
time sec.\n" if $ret;
129             print "Host $ip:$port is blocked or not
up\n" if not $ret;
130         }
131     }
132 }

saída:

$ perl exemplo9.14.pl -h www.perl.org,www.freebsd.org,www.
bbc.co.uk -p 80 -n 2
Host 207.171.7.51:80 answered in 0.202269077301025 sec.
Host 69.147.83.34:80 answered in 0.223164081573486 sec.
Host 212.58.246.90:80 answered in 0.22022008895874 sec.
Host 207.171.7.51:80 answered in 0.199004888534546 sec.
Host 69.147.83.34:80 answered in 0.23108696937561 sec.
Host 212.58.246.90:80 answered in 0.224642992019653 sec.
```

Se compilarmos este programa com ***pp -o sping sping.pl*** teremos um executável de linha de comando que poderemos adicionar ao nosso path e usar como qualquer outro comando de nosso sistema.

O CPAN possui diversas outras facilidades de console, como o **UI::Dialog::Console** e o **Getopt::Long,** que podem ser muito úteis em projetos que necessitem de uma interface console mais amigável.

9.9 IPC

A comunicação interprocessos (IPC) é uma forma de ampliar as capacidades de seu programa para se comunicar com processos de outros programas, ou mesmo gerar processos filhos de seu programa. A forma mais simples de IPC para o programador Perl são as ***funções internas exec,*** expressões entre `` e ***system.***

9.9.1 A função system

A função ***system*** criará um processo filho para o nosso programa. Este processo filho executará o comando informado no argumento de ***system***. Veja:

```
  1 #exemplo9.15
  2 use 5.12.0;
  3
  4 if ( !system "ls" ){
  5         say "Programa externo ls executado";
  6 }

saída:

err.pl     pipe_in.pl  pipe.pl
Programa externo executado
```

Observando o código e seu resultado, podemos concluir que o ***perl*** buscou o comando ***ls*** em nosso ***Path*** e o executou. Foi criado um processo filho e, após o processo filho executar, o ***perl*** prosseguiu com o seu código. Agora veja o código a seguir, onde adicionamos alguns argumentos no ls.

```
  1 #exemplo9.16
  2 use 5.12.0;
  3
  4 system q/ls -l/;
  5 say "Programa externo executado";

saída:

total 12
-rw-rw-r--. 1 thiago thiago  61 Set 11 15:25 err.pl
-rw-rw-r--. 1 thiago thiago 174 Set  9 22:00 pipe_in.pl
-rw-rw-r--. 1 thiago thiago 182 Set  9 21:26 pipe.pl
Programa externo executado
```

Quando o comando se torna mais complexo, a função ***system*** evoca o interpretador de comandos do seu sistema (no caso o ***bash***) para executar o comando. Normalmente isso seria insignificante, e em muitos casos é, mas você sabe quantos ***exploits*** existem por aí esperando uma falha em uma execução de ***shell***? É muito mais seguro e rápido utilizar o formato a seguir, que não terceiriza a execução para o interpretador do seu sistema:

```
$ perl -E'
>system qw/ls -l -a -f/;
>say $!;
>'
```

Recebendo um array como argumento, a função ***system*** executa os comandos sem terceirizar o processo para o ***shell***, tornando seu programa mais rápido e seguro. Mas a função ***system*** tem um grande ponto negativo e dois grandes pontos positivos.

O grande ponto negativo é a incapacidade de obtermos a saída do programa e processá-la, visto que ela é enviada direto para a ***STDIN*** e para a ***STDERR*** do seu sistema pelo comando. Os grandes pontos positivos são: a segurança, pois podemos executar o comando sem o intermédio do interpretador do seu sistema; e a possibilidade de receber o status da execução do comando. Todo comando executado retorna um valor, de acordo com o padrão ***Posix,*** sendo **0** para terminado com êxito. A lista completa de erros do padrão pode ser obtida em http://linux.die.net/man/3/errno.

Veja:

```
1 #exemplo9.17
2 use 5.12.0;
3
4 my @command = ("netstat", "-t");
5 my $errno = system @command;
6 say "Valor da execução: $errno";
```

```
saída:

Conexões Internet Ativas (sem os servidores)
Proto  Recv-Q  Send-Q  Local Address  Foreign Address            State

tcp    0       0       thiago:41557   sites.die.net:http         ESTABELECIDA
tcp    0       0       thiago:59471   eze03s01-in-f100.1e100:http ESTABELECIDA
tcp    0       0       thiago:41556   sites.die.net:http         ESTABELECIDA
tcp    0       0       thiago:41558   sites.die.net:http         ESTABELECIDA
tcp    0       0       thiago:42827   jorma.blender.org:http     ESTABELECIDA
```

Para experimentar a saída gerada por um erro, substitua a linha 4 do exemplo 9.17 por ***my @command = ("netstat", "-y")*** e veja a saída:

```
netstat: invalid option - 'y'
...
Valor da execução: 1024
```

Aqui temos um caso de erro onde o argumento é inválido. A mensagem ***"netstat: invalid option"*** é gerada pelo comando ***netstat*** e enviada para a ***STDERR*** do sistema; o Perl não teve acesso a ela. Ele apenas recebe o valor 1024 como controle do erro. Se substituirmos a linha 4 do mesmo exemplo por ***my @command = ("gato", "-y")***, teremos um erro diferente:

```
Valor da execução: -1
```

Neste caso o comando ***"gato"*** não existe em nosso ***PATH***, no caso ***$ENV{PATH}***, portanto a execução não é possível, retornando ***-1***. Uma vez que você sabe disto, pode tornar os exemplos mais complexos com o módulo ***Errno.pm***, na biblioteca padrão do Perl, e com a variável ***$!***. O comando a seguir mostra a lista de erros exportada por ***Errno.pm***.

```
$ perl -MErrno -E 'say for keys %!'
EBADR
ENOMSG
ENOTSUP
ESTRPIPE
```

```
EADDRINUSE
EL3HLT
...
```

O módulo **Errno.pm** exporta todas as constantes definidas pela biblioteca **C** *errno.h,* permitindo que possamos criar tratamentos de erro mais elaborados que **$!**.

Vamos fazer um código simples para verificar a execução de um comando externo.

```
  1 #exemplo9.18
  2 use 5.12.0;
  3 use Errno (":POSIX");
  4
  5 #Errno exporta um hash, %! que tem em suas chaves os
erros definidos por
  6 #errno.h. Quando um programa retorna um erro definido
por errno.h a chave
  7 #equivalente de $! tem valor verdadeiro. @command defi-
ne o comando.
  8 my @command = qw(ls -l -l -h /);
  9 system @command; #O código retornado por ls é armazena-
do em $?
 10
 11 #se o comando de system for executado com sucesso, $?
será falso.
 12 if ($?){
 13     say mensagens_amigaveis(\%!);
 14     exit 0;
 15 }
 16 else{
 17     say "executado normalmente";
 18 }
 19 sub mensagens_amigaveis{
 20 #Gera mensagens amigáveis para erros definidos em
errno.h ou
 21 #o número do erro retornado se este não estiver defini-
do em errno.h
 22     my ($err_hash_ref)= @_;
```

```
 23     my $erro_ativo;
 24
 25 #Aqui verificamos o erro informado e o seu valor
 26     foreach my $erro (keys %$err_hash_ref){
 27             $erro_ativo = $erro if $err_hash_ref-
->{$erro};
 28     }
 29     say $erro_ativo;
 30     given ($erro_ativo || $?){
 31             #geramos a mensagem
 32        when ($_ eq "ENOENT") {
 33        return "O comando não existe em PATH ou na loca-
lidade especificada";
 34        }
 35        #colocar todos os casos desejados ou possíveis
em cláusula when.
 36        default { return "Erro não especificado, $_"}
 37     }
 38 }
```

saída:

```
total 110K
dr-xr-xr-x.  23 root root 4,0K Set 12 09:59 .
dr-xr-xr-x.  23 root root 4,0K Set 12 09:59 ..
-rw-r--r--.   1 root root    0 Set 12 09:59 .autofsck
dr-xr-xr-x.   2 root root 4,0K Set 11 08:32 bin
dr-xr-xr-x.   5 root root 1,0K Set 10 16:28 boot
drwxr-xr-x.   2 root root 4,0K Ago  2 11:50 cgroup
drwx------.   3 root root 4,0K Set  2 21:18 .dbus
drwxr-xr-x.  22 root root 4,0K Set 12 10:50 dev
drwxr-xr-x. 118 root root  12K Set 12 10:03 etc
drwxr-xr-x.   4 root root 4,0K Set  2 21:20 home
dr-xr-xr-x.  17 root root  12K Set 11 08:32 lib
drwx------.   2 root root  16K Mai 12 22:18 lost+found
drwxr-xr-x.   2 root root 4,0K Set 10 05:57 media
drwxr-xr-x.   4 root root 4,0K Set  2 21:11 mnt
drwxr-xr-x.   2 root root 4,0K Out  1  2009 opt
```

```
dr-xr-xr-x. 198 root root    0 Set 12 09:59 proc
dr-xr-x---.   7 root root 4,0K Set 11 15:42 root
dr-xr-xr-x.   2 root root  12K Set 11 08:32 sbin
drwxr-xr-x.   7 root root    0 Set 12 09:59 selinux
drwxr-xr-x.   2 root root 4,0K Out  1  2009 srv
drwxr-xr-x.  12 root root    0 Set 12 09:59 sys
drwxrwxrwt.  72 root root 4,0K Set 12 17:45 tmp
drwxr-xr-x.  13 root root 4,0K Mai 12 22:20 usr
drwxr-xr-x.  21 root root 4,0K Set  2 20:36 var
executado normalmente
```

Para verificarmos se nosso código funciona, vamos substituir **'ls'** por **'gato'**:

```
Can't exec "gato": Arquivo ou diretório não encontrado at /
home/thiago/workspace/TICursos/errno.pl line 12. ENOENT O
comando não existe em PATH ou na localidade especificada.
```

Outra coisa interessante que pode ser feita com a função ***system*** é controlar a execução de scripts em Perl, que retornam para um código significativo por meio da função ***exit.*** A função ***exit*** tem como argumento um número que é o código retornado para o programa que chamou o script Perl.

9.9.2 A função exec

A função **exec**, tal qual a função system, executa um programa externo, porém ela não cria um processo filho e sim utiliza o próprio processo do seu programa para executar o programa externo. Então seu programa encerra juntamente com o fim do comando evocado por ***exec***.

```
  1 #exemplo9.19
  2 use 5.12.0;
  3 exec "netstat -t";
  4 say "Nunca executa!!!";
saída:
```

9.9.3 Obtendo a saída de um comando

Existe mais uma forma de executar um programa externo e obter uma iteração interessante. Ao executar os comandos do sistema com as funções **system** ou **exec**, nosso programa não pode verificar as saídas fornecidas pelos comandos – a não ser que estivessem direcionadas para um arquivo e o nosso script abrisse o arquivo depois da execução. Para que possamos ler as saídas em tempo real o Perl permite os comandos executados entre acentos graves ``.

```
  1 #exemplo9.20
  2 use 5.12.0;
  3
  4 my $date = `date`;
  5 print "O Perl recebe a saída do comando date: ", $date;

saída:

O Perl recebe a saída do comando date: Sun Oct 30 21:24:59
BRST 2011
```

Além de ler a saída padrão, os comandos entre acentos graves podem obter a saída de erro ***(STDERR)*** e o código de erro retornado pelo comando externo. Bom, né? Porém, as desvantagens dos comandos entre acentos grave são: eles são executados pelo interpretador do seu sistema; são mais lentos e geram mais ***overhead*** que os comandos executados por ***system.***

Vamos ver um programa assim:

```
  1 #exemplo9.21
  2 use 5.12.0;
  3
  4 #executa o comando e envia a saída pada @lista_de_ar-
quivos em vez da
  5 #saída padrão, permitindo que o valor seja processado.
  6 my @lista_de_arquivos = `ls -l /home/thiago`;
  7
  8 #Processa a lista de arquivos de forma personalizada.
  9 for (@lista_de_arquivos){
```

```
 10     if ($_ =~ /^d/){
 11         my ($permission, $int, $owner, $user,
$size,$date, $dir) =
 12         $_ =~
 13  /(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+\s+\S+\
s+\S+)\s+(\S+.*)/;
 14         say "$permission\t$owner\t$dir"; }
 15 }
```

saída:

```
drwxr-xr-x.      thiago      Área de Trabalho
drwxr-xr-x.      thiago      Documentos
drwxr-xr-x.      thiago      Download
drwxr-xr-x.      thiago      Imagens
drwx--x--x.      root        minicpan
drwxr-xr-x.      thiago      Modelos
drwxr-xr-x.      thiago      Música
drwx--x--x.      thiago      Programming
drwxr-xr-x.      thiago      Público
drwxr-xr-x.      thiago      Vídeos
drwxrwxr-x.      thiago      workspace
```

Podemos ainda direcionar a saída de erros padrão do comando externo com **2>&1**:

```
  1 #exemplo9.22
  2 use 5.12.0;
  3
  4 my @saida = `ls -la /home/frodo 2>&1`;
  5 say "Meu comando com STDOUT controlado:\n@saida";
  6 say "Erro número $?";
```

saída:

```
Meu comando com STDOUT controlado:
ls: impossível acessar /home/frodo: Arquivo ou diretório
não encontrado

Erro número 512
```

A importância do controle da saída, incluindo a saída de erro, é o poder de processar a informação fornecida pelo comando externo e adaptá-la à sua real necessidade e não confundir o seu usuário com saídas sobrepostas.

Existe no CPAN o projeto ***Perl Power Tools*** http://search.cpan.org/~cwest/ppt-0.14/, que contém uma versão em Perl dos principais comandos Linux, o que pode ser interessante para garantir portabilidade em programas que usem comandos externos. Evocar comandos externos deixa seu programa mais lento, sim, mas permite que a prototipagem de seu produto seja muito mais rápida, podendo apresentar um produto funcional em menos tempo e com pouca codificação.

9.10 Pipes

Os pipes são comandos ou processos filhos que recebem ou enviam dados em tempo de execução. Os comandos entre `` e os comandos executados por ***system*** devem receber os argumentos ou enviar a saída sem iteratividade. Para criar um pipe usamos o comando ***open***, informamos um filehandle e colocamos um pipe (|) no segundo argumento da função. Para criarmos um pipe de escrita colocamos | no início do comando, e para criar um PIPE de leitura colocamos | ao final do comando.

Quando iniciamos o comando ***cat*** do Linux, sem nenhum arquivo no argumento, ele passa a ler os valores digitados em STDIN. No caso do exemplo9.23, os valores que ***cat*** deve ler são enviados pelo filehandle aberto com um pipe de escrita.

```
  1 #exemplo9.23
  2 use 5.12.0;
  3
  4 open my $cat, "| cat -n";
  5 print $cat "Primeira Linha\n";
  6 print $cat "segunda linha\n";
  7 print $cat "terceira linha\n";
  8 say "Veja a saída de cat:";
  9 close $cat;

saída:

Veja a saída de cat:
     1     Primeira Linha
     2     segunda linha
     3     terceira linha
```

Um pipe de leitura aumenta o desempenho de um programa que processa a saída de um comando externo, pois permite que os dados sejam processados à medida que são gerados.

```
1 #exemplo9.24
2 use 5.12.0;
3
4 open my $connections, "netstat |";
5 while(<$connections>){
6         say if /tcp|udp/;
7 }
```

saída:

```
tcp   0  0  thiago:58416  iad04s01-in-f101.1e100:http  ESTABELECIDA
tcp   0  0  thiago:59787  bb43c187.virtua.com.b:23630  ESTABELECIDA
tcp   0  0  thiago:35858  66.117.43.23:http            ESTABELECIDA
tcp   0  0  thiago:45487  linux.pucpr.br:http          ESTABELECIDA
tcp   0  0  thiago:52983  sn1msg3020107.sn1.gate:msnp  ESTABELECIDA
tcp   0  0  thiago:51474  iad04s01-in-f102.1e100:http  ESTABELECIDA
```

Nestes exemplos o interpretador do meu sistema executou os comandos. Para definirmos um pipe que não é intermediado pelo interpretador podemos usar o seguinte formato:

```
open my $filehandle, "|-", "comando", @argumentos; #escrita
open my $filehandle, "-|", "comando", @argumntos; #leitura
```

Refaça o exemplo9.23 com esta nova forma de pipe:

```
1 #exemplo9.23
2 use 5.12.0;
3
4 open my $comando,  "|-", "cat", "-n";
```

Assim como na função ***system***, o comando executado por um pipe modifica o valor de *$?* e pode ser usado com Errno.pm.

Exercícios

1. Monte uma versão simplificada do comando ls (que lista diretórios).

2. Monte uma versão simplificada do comando cp (que copia arquivos).

3. Monte uma versão simplificada do comando touch (que cria arquivos vazios).

4. Monte uma versão do comando grep.

5. Monte uma versão simplificada do comando expand (que substitui tabulações por agrupamentos de quatro espaços).

6. Pesquise nomes de comandos no CPAN e veja o que encontra.

7. Qual a vantagem que um utilitário de linha de comando feito em Perl teria sobre os utilitários originais?

9. Monte um programa que liste os arquivos de um diretório organizados por data de criação.

10. Refaça o exemplo9.8 de forma que sejam implementadas as funções leitura(), append(), le_append() , ler_modificar(), cria_arquivo().

Capítulo 10. Técnicas de Debug - The Perl Debugger

10.1 Antes de caçar insetos...

Insetos são seres nojentos que gostam de sujeira, gordura, cantos escuros e pouco arejados. Para diminuir sua incidência mantemos nossa casa limpa, arejada e de tempos em tempos dedetizamos o ambiente... mas eles são inevitáveis e sempre uma baratinha aparece num dia de chuva ou uma mosca entra pela nossa janela.

Na programação e no desenvolvimento de sistemas devemos manter um ambiente que evite que nosso sistema seja infestado de bugs. E quando estes inevitáveis seres aparecerem devemos ter um ambiente que permita localizá-los. As primeiras regras de programação higiênica são: **definir o escopo; definir as funcionalidades e metodologias do seu programa; codificar de forma limpa e organizada; testar e debugar o programa; distribuir; fazer novos ajustes e manutenção do programa.**

10.1.1 Definir o escopo de atuação do programa

Qual problema eu quero resolver ou qual facilidade eu quero criar com este programa? Alguns novatos acham estranho, mas eu sempre oriento: **antes de criar um programa, documente o problema**. Você não pode resolver com eficácia um problema que não entende bem. Numa abstração simples, imagine que sua casa foi invadida por formigas. Você pode tentar pisar nas formigas. Alguns vão comprar venenos nos supermercados. Mas um profissional vai pesquisar sobre as formigas e descobrir que você deve matar a rainha a fim de eliminar a infestação em definitivo. Depois vem a decisão de como matá-la. Com um revólver? Uma bazuca? Com um veneno especializado para formigas que deve ser posto dentro do formigueiro? A última parece mais sensata.

Parece cômica, mas a comparação é real. Algumas vezes grandes corporações tentam matar insetos com revólveres, enquanto pequenas empresas querem caçar tigres armados com estiletes. Entender o problema é importante para que você não crie uma aplicação de intranet num servidor Apache com MySQL para resolver um problema que pode ser resolvido com uma planilha e uma macro. Ou tentar resolver um problema que precise um banco de dados relacional com uma planilha e uma macro.

É comum programadores verem um problema superficialmente e já darem respostas como: "ah, podemos resolver isso com uma aplicação web, um banco de dados e um servidor dedicado!" apenas porque os desenvolvedores têm afinidade com esta ou aquela tecnologia, sem realmente verificarem a necessidade do cliente. Eu já vi um serviço Apache levantado para uma aplicação usada em apenas uma máquina. O desenvolvedor preferiu manter-se na sua zona de conforto e criar um aplicativo web para ser usado por apenas um usuário a aprender o paradigma do desenvolvimento desktop e desenvolver uma interface gráfica. Isso na verdade insere mais problemas do que resolve. A máquina fica exposta a ataques internos de outros usuários da rede ou mesmo consumo exagerado de recursos.

Definir o escopo do problema vai mantê-lo no foco do que deve ser resolvido. E vai ajudar a definir o traçado que o projeto deve seguir.

10.1.2 Definir as funcionalidades do p ograma

Após definir o escopo da sua aplicação, vamos definir e documentar como ela deve fazer isso. Ainda aqui, antes de entrar na tecnologia, vá para a abstração. Definir quais botões de interface gráfica ou opções de linha de comando devemos oferecer para nossos usuários. Quais telas e como elas devem ser. Uma vez bem definido o problema e o que devemos oferecer ao usuário para solucioná-lo, enfim podemos estudar quais tecnologias, paradigmas e metodologias vamos usar para atacar este problema. Veja que até agora não falei de Perl. Entenda que os melhores erros que um analista ou programador pode enfrentar são os erros em tempo de compilação. Caçar erros de lógica em um programa em funcionamento é pior. E pior ainda é ter que modificar uma aplicação porque, bem, ela não resolve o problema a que se propõe solucionar. Ou além: trouxe novos problemas para o cliente. O programa também pode ser reescrito porque usou uma abordagem errada. O analista decidiu usar uma abordagem orientada a funções em uma aplicação que sofre muitas modificações ao longo do tempo e sempre tem novas funcionalidades incorporadas... ou gastou muito tempo desenvolvendo um sistema escrito com orientação a

objetos, e o programa tem apenas dois arquivos de 100 linhas cada... e bom... está rodando há dois anos sem nenhuma modificação e nenhum módulo do código foi reaproveitado em outro projeto. Precisava ser OOP?

Quando definimos quais metodologias e tecnologias usar, não podemos ser como terroristas ideológicos, que preferem morrer por esta ou aquela causa. Legal, você acha que o MySQL é o melhor banco de dados relacional do mundo; mas vai levantar um servidor novo por quê, se o seu cliente já tem um MS-SQL rodando e funcionando? Será que um SQLite não será mais do que o suficiente neste caso? Acho terrível chegar em clientes com quatro servidores de bancos de fabricantes diferentes, porque cada consultor que vai lá decide rodar a aplicação em um servidor diferente. Apenas devemos fazer o que é melhor para o objetivo final do programa. Escolher uma abordagem adequada para o problema vai evitar que seu sistema seja um bug para seu cliente.

Antes de codificar, pense no problema a ser resolvido. Depois, pense em como seu programa vai solucionar os problemas. Então defina as funções e como elas devem funcionar. Só então comece a codificar.

10.1.3 Depois de tudo isso... codifica

Escrever um código de forma coerente, com boas práticas, vai facilitar a sua vida. Comente regularmente seu código, utilize espaços e identação... enfim, adote um padrão coerente para identação, nomes de variáveis etc. Você pode usar como referência o manual ***perlstyle*** em ***perldoc perlstyle***.

10.1.4 Executar o programa

Esta parte é simples. Como eu disse antes, os melhores erros ocorrerão nesta fase. E o próprio compilador irá ajudá-lo a encontrar os erros que impedem que o programa compile.

10.1.5 Teste e debug

Nesta fase você irá caçar erros menos evidentes. Vai aprender a buscar erros e testar o programa. Antes de distribuir um programa você deve testá-lo amplamente, buscando possíveis falhas de lógicas e de segurança ou mesmo erros que impedem que o programa tenha a saída desejada, até passando pela fase da compilação. Isso seria o equivalente a fazer a dedetização de sua casa regularmente ou sair à caça de alguma mosca desavisada que entrou pela sua janela...

10.1.6 Entregar e ajustar o programa

Tenha certeza de que depois de tudo isso, ao implantar o programa, você ainda encontrará alguns pequenos problemas ou mesmo um cliente mais chato que vai dizer: "puxa... mas eu achei que essa funcionalidade seria diferente." ou "Ah... pensando bem, não tem como modificar esta pesquisa para ser por nome, além de CPF?". Sim, seu programa sofrerá pequenas mudanças na própria fase de distribuição.

10.1.7 Modifica ões e manutenções

Sim, nós testamos o programa para diversas possibilidades de entrada. Mas sempre vamos enfrentar um bug novo ou falha de segurança. Às vezes você terá que debugar algo no próprio ambiente do cliente. Por isso é interessante que seu código tenha linhas comentadas que, ao remover o token de comentário, permita ao programa trazer informações úteis para localizar possíveis erros além das possibilidades de mudança de ambiente ou mesmo solicitações de novas funcionalidades ao sistema.

10.2 O debugger do Perl

O Perl não é apenas uma linguagem de programação. É uma ferramenta completa de desenvolvimento. O debugger que acompanha o Perl pode ser evocado com o comando ***perl -d nome_do_programa.pl***. Vamos estudar o debugger com um pequeno algoritmo de pesquisa binária.

```
      1 #exemplo10.1
      2 use 5.12.0;
      3 use integer;
      4
      5 my @arr = ('a' .. 'z');
      6 my $indice = binary_search(to_search => 'h', array
=> \@arr);
      7
      8 say $arr[$indice] if defined $indice;
      9
     10
```

```
11 sub binary_search{
12         my %args = @_;
13         my $str = $args{to_search};
14         my $array_ref = $args{array};
15         my $low = 0;
16         my $high = scalar @$array_ref;
17
18         while( $low < $high){
19                 my $middle = ($low + $high)/2;
20                 if ($array_ref->[$middle] lt $str){
21                         $low = $middle + 1;
22                 }
23                 else {
24                         $high = $middle;
25                 }
26         }
27         return $low;
28 }
```

```
$ perl -d exemplo10.1.pl

Loading DB routines from perl5db.pl version 1.33
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more
help.

main::(exemplo10.1.pl:5):       my @arr = ('a' .. 'z');
  DB<1>
```

Observe que o debugger mostra qual linha irá executar na sequência, a linha 5, pois as primeiras três linhas são cláusulas ***use*** e comentários. A linha 4 está em branco. Se você digitar ***h*** irá acessar o ***help*** do debugger. Você pode obter ajuda específica de algum comando do debugger com ***h <opção>***. Veja:

```
 DB<1> h v
v [line]    View window around line.
```

O comando ***v*** mostra o código próximo à linha atual:

```
  DB<2> v
2:      use 5.12.0;
3:      use integer;
4
5==>    my @arr = ('a' .. 'z');
6:      my $indice = binary_search(to_search => 'h', array
=> \@arr);
7
8:      say $arr[$indice] if defined $indice;
9
10
11      sub binary_search{
```

Observe que o debugger informou a posição da próxima linha a ser executada com a seta '**==>**'. Se quiser ver ao redor de uma linha específica, pode informar a linha junto com o comando v:

```
  DB<2> v 13
10
11      sub binary_search{
12:             my %args = @_;
13:             my $str = $args{to_search};
14:             my $array_ref = $args{array};
15:             my $low = 0;
16:             my $high = scalar @$array_ref;
17
18:             while( $low < $high){
19:                     my $middle = ($low + $high)/2;
```

Voltando à nossa execução. Para executar uma linha temos as opções ***n*** e ***s***; ambas executam a próxima linha. Mas, caso a próxima linha seja uma chamada a uma

função ou método de uma classe, a opção ***s*** fará o debugger entrar na função e poderemos analisá-la linha a linha, mesmo se a função ou método estiver em outro arquivo. A opção ***n*** executa a função toda em um único passo, de forma rápida. A opção ***n*** é interessante quando você sabe que a função executa perfeitamente e não precisa analisar o que ocorre no interior dela.

```
  DB<3> n
main::(exemplo10.1.pl:6):       my $indice = binary_
search(to_search => 'h', array => \@arr);
```

Vamos executar o ***s*** para acessarmos a função que criamos. Mas antes vamos ver o conteúdo do array ***@arr***. Para ver o conteúdo de uma variável usamos o comando **p <variavel>**.

```
  DB<3> p @arr
abcdefghijklmnopqrstuvwxyz
```

Porém, ***p*** retorna o valor em contexto ***scalar*** e pode ser muito mais informativo verificar um array ou hash em contexto de lista. Para isso usamos o ***x.***

```
  DB<4> x @ar
0  'a'
1  'b'
2  'c'
3  'd'
... (linhas omitidas)
21  'v'
22  'w'
23  'x'
24  'y'
25  'z'
```

Agora vamos acessar a função binary_search com o comando ***s:***

```
  DB<5> s
main::binary_search(exemplo10.1.pl:12):
12:             my %args = @_;
```

Como ***binary_search*** não evoca outras funções, podemos avançar com o comando ***n***.

```
  DB<5> n
main::binary_search(exemplo10.1.pl:13):
13:             my $str = $args{to_search};
```

Vamos enfatizar o fato: o debugger informa o nosso namespace sempre. A função ***binary_search*** fica dentro de ***main***.

```
main::binary_search(exemplo10.1.pl:13):
```

Agora podemos ver um hash, %args, com ***p*** e ***x.***

```
  DB<1> p %args
arrayARRAY(0x1426464)to_searchh
```

Veja que o retorno é o mesmo que o de um array em contexto escalar. Com o ***x*** teremos:

```
0  'array'
1  ARRAY(0x1426464)
   0  'a'
   1  'b'
   2  'c'
   3  'd'
 ... linhas omitidas.
   23  'x'
   24  'y'
   25  'z'
2  'to_search'
3  'h'
```

O mesmo retorno de um array. Mas temos a chave "array" como elemento 0, a referência ao array de pesquisa como elemento 1, a chave to_search como ele-

mento 2 e o valor a ser pesquisado, h, como elemento 3. O comando x também informou todo o conteúdo do array de pesquisa. Mas se temos um hash, por que ele é retornado como um array? Lembra que internamente um hash é como um array, porém o Perl mantém alguns algoritmos internos que permitem que usemos as chaves como indexadores dos elementos? Pois é. Mas e se quisermos ver um hash como, bem... como um hash? Basta fazermos ***x \%args.***

```
  DB<4> x \%args
0  HASH(0x19636a4)
   'array' => ARRAY(0x1426464)
      0  'a'
      1  'b'
      2  'c'
  ... linhas omitidas ...
      24  'y'
      25  'z'
   'to_search' => 'h'
```

Agora sim, o debugger relaciona as chaves com seus respectivos valores e também informa os valores do array de pesquisa. Podemos avançar mais uma linha no nosso programa. Aliás, se você digitar v agora, vai ver que não temos nada interessante para analisar até a linha 17. Então podemos avançar diversas linhas com o comando ***c <linha>***. Se não informarmos a linha, o debugger irá avançar até encontrar um breakpoint. No caso, vamos tentar avançar até a linha 17: ***c 17***. E oops...

```
  DB<5> c 17
Line 17 not breakable.
```

O que isso quer dizer? Linhas exclusivas de comentário, perldoc ou linhas em branco não podem ser usadas para parada ou breakpoint. Um breakpoint, ou ponto de parada, é uma linha do programa que você pode usar como um ponto estratégico para analisar o estado do programa. Então faremos ***c 18.***

```
  DB<6> c 18
main::binary_search(exemplo10.1.pl:18):
18:             while( $low < $high){
```

Eu quero que o debugger pare a execução toda vez que chegar neste ponto, o início do loop. Este ponto é interessante, pois permite que eu analise o estado das variáveis a cada iteração do loop ***while***. Para isto preciso de um breakpoint. Toda vez que o debugger for executar o código nesta linha ele para. Veja o que diz a ajuda do comando ***b h***.

```
  DB<7> h b
b        Sets breakpoint on current line
b [line] [condition]
        Set breakpoint; line defaults to the current execu-
tion line;
        condition breaks if it evaluates to true, defaults
to '1'.
b subname [condition]
        Set breakpoint at first line of subroutine.
```

Vamos criar uma parada obrigatória na linha 19:

```
  DB<8> b 19
  DB<9> c
main::binary_search(exemplo10.1.pl:19):
19:                     my $middle = ($low + $high)/2;
```

Então toda vez que rodarmos com o comando c, iremos executar até a linha 19. Cada iteração irá parar na linha 19.

```
  DB<2> c
main::binary_search(binary_search.pl:19):
19:                     if ($array_ref->[$middle] lt $str){

  DB<2>
```

E podemos solicitar os valores das variáveis.

```
  DB<9> p $high
26
  DB<10> p $middle

  DB<11> p $low
0
```

Ou ainda marcar variáveis para serem assistidas com ***w***. Assim o debugger irá nos avisar sempre que elas mudarem de valor.

```
  DB<12> h w
w expr        Add a global watch-expression.
w             Does nothing
```

Então marcamos as variáveis que queremos monitorar, e sempre que elas forem modificadas o debugger irá informar o valor delas:

```
  DB<13> w $high
  DB<14> w $low
  DB<15> w $middle
  DB<16> c
Watchpoint 2:   $middle changed:
    old value:  ''
    new value:  '13'
main::binary_search(exemplo10.1.pl:20):
20:                     if ($array_ref->[$middle] lt $str){
```

No nosso caso, é muito interessante vigiar $array_ref->[$middle], pois é ela que verifica o caractere atual da pesquisa.

```
  DB<16> w $array_ref->[$middle]
  DB<18> p $array_ref->[$middle]
n
```

Então acompanhamos os valores em cada iteração:

```
  DB<19> c
Watchpoint 0:   $high changed:
    old value:  '26'
    new value:  '13'
Watchpoint 2:   $middle changed:
    old value:  '13'
```

```
    new value:   ''
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:   'n'
    new value:   'a'
main::binary_search(exemplo10.1.pl:19):
19:                     my $middle = ($low + $high)/2;

  DB<19> c
Watchpoint 2:   $middle changed:
    old value:   ''
    new value:   '6'
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:   'a'
    new value:   'g'
main::binary_search(exemplo10.1.pl:20):
20:                     if ($array_ref->[$middle] lt $str){

  DB<19> c
Watchpoint 1:   $low changed:
    old value:   '0'
    new value:   '7'
Watchpoint 2:   $middle changed:
    old value:   '6'
    new value:   ''
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:   'g'
    new value:   'a'
main::binary_search(exemplo10.1.pl:19):
19:                     my $middle = ($low + $high)/2;
  DB<19> c
Watchpoint 2:   $middle changed:
    old value:   ''
    new value:   '10'
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:   'a'
    new value:   'k'
main::binary_search(exemplo10.1.pl:20):
20:                     if ($array_ref->[$middle] lt $str){
```

```
  DB<19> c
Watchpoint 0:   $high changed:
    old value:  '13'
    new value:  '10'
Watchpoint 2:   $middle changed:
    old value:  '10'
    new value:  ''
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:  'k'
    new value:  'a'
main::binary_search(exemplo10.1.pl:19):
19:                     my $middle = ($low + $high)/2;
  DB<19> c
Watchpoint 2:   $middle changed:
    old value:  ''
    new value:  '8'
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:  'a'
    new value:  'i'
main::binary_search(exemplo10.1.pl:20):
20:                     if ($array_ref->[$middle] lt $str){
  DB<19> c
Watchpoint 0:   $high changed:
    old value:  '10'
    new value:  '8'
Watchpoint 2:   $middle changed:
    old value:  '8'
    new value:  ''
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:  'i'
    new value:  'a'
main::binary_search(exemplo10.1.pl:19):
19:                     my $middle = ($low + $high)/2;
  DB<19> c
Watchpoint 2:   $middle changed:
    old value:  ''
    new value:  '7'
Watchpoint 3:   $array_ref->[$middle] changed:
```

```
    old value:   'a'
    new value:   'h'
main::binary_search(exemplo10.1.pl:20):
20:                     if ($array_ref->[$middle] lt $str){
  DB<19> c
Watchpoint 0:   $high changed:
    old value:   '8'
    new value:   '7'
Watchpoint 2:   $middle changed:
    old value:   '7'
    new value:   ''
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:   'h'
    new value:   'a'
main::binary_search(exemplo10.1.pl:27):
27:             return $low;
```

Até que finalmente chegamos em 'h' como valor de $array_ref->[$middle]. Então o debugger irá sair da sub-rotina atual para o caller automaticamente no próximo passo. Quando você estiver analisando uma sub-rotina com muitos loops e sua análise estiver concluída antes do final da sub-rotina, você pode dizer ao debugger para concluir a sua execução com o comando ***r***.

```
  DB<12> h r
r        Return from current subroutine.
rerun           Rerun session to current position.
rerun n         Rerun session to numbered command.
rerun -n        Rerun session to number'th-to-last command.

    DB<19> r
scalar context return from main::binary_search: 7
Watchpoint 0:   $high changed:
    old value:   '7'
    new value:   ''
Watchpoint 1:   $low changed:
    old value:   '7'
    new value:   ''
```

```
Watchpoint 3:   $array_ref->[$middle] changed:
    old value:  'a'
    new value:  ''
main::(exemplo10.1.pl:8):       say $arr[$indice] if de-
fined $indice;
```

O debugger informou que estamos de volta em main. O comando ***v*** mostra que estamos na linha 8.

```
  DB<19> v
5:      my @arr = ('a' .. 'z');
6:      my $indice = binary_search(to_search => 'h', array
=> \@arr);
7
8==>    say $arr[$indice] if defined $indice;
9
10
11      sub binary_search{
12:             my %args = @_;
13:             my $str = $args{to_search};
14:             my $array_ref = $args{array};
```

Podemos conferir os valores de ***$indice*** e ***$arr[$indice]***.

```
  DB<19> p $indice
7
  DB<21> p $arr[$indice]
h
```

Veja que o Perl automaticamente eliminou as variáveis que eram internas à função ***binary_search*** assim que saímos de seu escopo. Se executarmos mais um ***n***, o programa será encerrado. Mas antes vejamos mais alguns recursos do debugger.

Você pode executar comando e funções do Perl dentro do debugger:

```
  DB<15> print "teste"
teste
  DB<16> print sin 2
0.909297426825682
 DB<17> mkdir meu_dir, 0755
```

Modificar valor de variáveis do seu programa durante a execução:

```
  DB<18> p $indice
7
  DB<19> $indice = 8

  DB<20> p $indice
8
```

Ou mesmo reiniciar o debugger:

```
  DB<21> h R
R        Pure-man-restart of debugger, some of debugger
state
        and command-line options may be lost.
        Currently the following settings are preserved:
        history, breakpoints and actions, debugger Options
        and the following command-line options: -w, -I, -e.
```

Agora podemos nos divertir caçando os insetos que porventura entrarem em nossos programas.

Exercícios

1. A melhor forma de exercitar o uso do Perl Debugger é debugar programas. Execute o debugger nos exercícios dos capítulos anteriores para praticar.

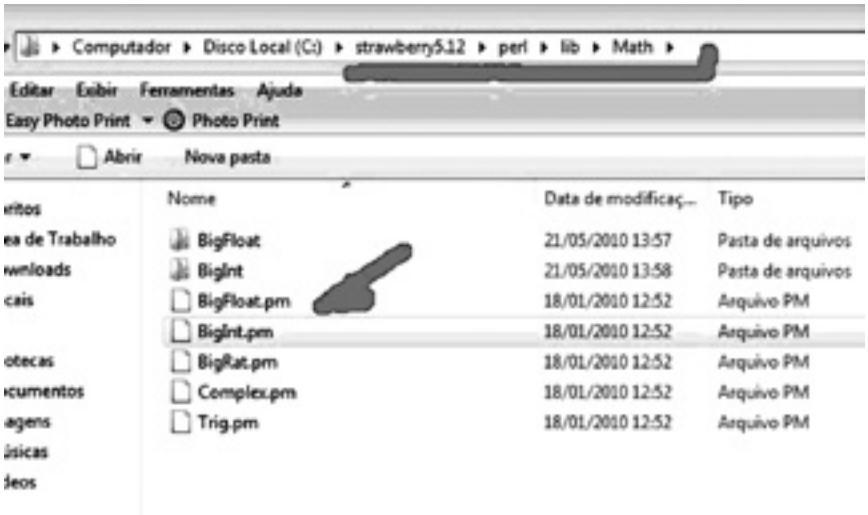
2. Execute o script exemplo9.14 no modo debugger. Utilize o comando **s** para verificar o conteúdo das sub-rotinas e dos módulos que são carregados pelo Perl.

Capítulo 11. Utilizando Módulos do CPAN e Criando Seus Próprios Módulos

Já falamos um pouco de módulos. Mas ainda há algo mais para estudarmos. Como já foi dito, os módulos são a forma mais adequada de reusabilidade no Perl. Para criarmos um módulo, devemos iniciar um novo arquivo com ***package Nome_do_modulo;***. O arquivo deve ser nomeado como ***Nome_do_modulo.pm***.

NOTA: Você pode reutilizar código de diversas formas, afinal isto é o Perl. Por exemplo, utilizar diversos scripts separados evocados com ***require, system*** ou ***exec.*** Mas na programação moderna, os módulos são mais adequados, uma vez que utilizam apenas um interpretador perl, enquanto ***system*** e ***exec*** utilizam um interpretador para cada script que evocarem. A vantagem da cláusula ***use*** sobre a ***require*** reside no fato da verificação no tempo de compilação ser feita pelo ***use,*** enquanto ***require*** realiza a verificação do código evocado apenas durante a execução.

Para que o seu programa utilize um módulo usamos o pragma ***use*** seguido pelo ***Nome do Módulo;***. O operador ***use*** faz com que o compilador Perl carregue o módulo para uso no programa. Em alguns casos você pode ver os nomes de módulos como ***Pasta1::Pasta2::Modulo***. Isto significa que o seu módulo está em subpastas dentro da pasta onde se encontram as bibiotecas. Então, ***use Math::BingInt;*** se refere ao módulo **BigInt.pm** dentro da pasta **Math** da minha biblioteca do Perl.



NOTA: Nos sistemas Linux e Unix as pastas dos módulos Perl podem ter o ***path*** diferente conforme a distribuição e o fabricante, se você estiver utilizando a instalação padrão da distribuição. Caso você esteja utilizando uma instalação local, os arquivos dos módulos estarão dentro do diretório **<sua_instalação>/lib**:

Instalação local no meu Solaris:
$ ls localperl/lib/5.12.2/Math
BigFloat BigInt BigRat.pm Trig.pm
BigFloat.pm BigInt.pm Complex.pm

Instalação padrão do meu Solaris:
$ ls /usr/perl5/5.8.4/lib/Math
BigFloat BigInt BigRat.pm Trig.pm
BigFloat.pm BigInt.pm Complex.pm

Como o Perl é uma linguagem antiga que vem evoluindo continuamente com o tempo e mantém compatibilidade com as versões anteriores, você vai encontrar no CPAN módulos com interfaces orientadas a funções, módulos com interfaces orientadas a objetos e módulos que permitem ambas as interfaces.

Nos módulos orientados a funções, teremos duas situações distintas: os que exportam os nomes das funções e os que não exportam. Caso um módulo não exporte suas funções podemos acessá-las com o operador '**::**'. Ficando assim o uso:

```
use Modulo;

my $a = Modulo::funcao1;
my $b = Modulo:: funcao2;
```

Caso o módulo exporte funções, elas podem ser acessadas como já vimos em diversas situações anteriores:

```
use Modulo qw(funcao1 funcao2);

my $a = funcao1; #evoca Modulo::funcao1
my $b = funcao2;
```

Alguns módulos exportam as funções automaticamente:

```
use Modulo;

my $a = funcao1( ); #evoca Modulo::funcao1
```

Em módulos cuja interface é exclusivamente orientada a objetos, vamos criar uma instância do objeto com o método construtor fornecido pela classe, na maioria dos casos ***new.*** Então teremos acesso à interface fornecida.

```
use Modulo;

my $obj_ref = Modulo->new; #nem sempre o construtor é new
$obj_ref->metodo1;
$obj_ref->metodo2;
$var1 = $obj_ref->propriedade;
```

11.1 Pacotes e namespaces no Perl

A abrangência de uma variável, em Perl, é definida pelas funções ***my, our*** e ***local***. Como já dissemos, as variáveis declaradas com ***my*** têm visibilidade apenas no bloco onde são criadas. Já as variáveis criadas com ***local*** têm visibilidade no bloco

onde foram criadas e em qualquer função que é chamada dentro daquele bloco. E ***our*** é válida dentro de todo pacote. Mas o que é um pacote em Perl?

Vamos iniciar discutindo o pacote ***main*** em mais detalhes. Se você já programou em ***C*** ou ***Java***, deve estar familiarizado com a função ***main***. Esta é a primeira função a ser executada em um programa. É a função chamada pelo próprio sistema operacional para executar um programa. E no Perl? O Perl, como foi desenvolvido em C, herdou o conceito de ***main***; assim, o pacote principal de todo programa Perl é ***main***.

NOTA: O uso de variáveis com o operador ***local*** foi desencorajado, de forma que são gerados alertas caso você utilize variáveis declaradas com ***local*** quando o pragma ***use 5.12.0*** está ativado. Quando uma linguagem começa a desencorajar o uso de alguma de suas características significa que em um futuro próximo ela não mais será suportada.

```
  1 #exemplo11.1
  2 #use 5.12.0; #uso de local desencorajado, se ativar
5.12.0 o
  3 #operador "local" irá gerar um aviso.
  4
  5 my $var1 = 10;
  6 our $var2 = "My local";
  7 local $var3 = "Wow...";
  8
  9 print "$_ => ", $main::{$_}, "\n" for keys %main::;
 10 #O hash main possui uma referência para todas as variá-
veis e funções...
 11 print $main::var2, "\n";
 12
 13 #teste(); # mesmo que &main::teste;
 14 &main::teste;
 15
 16 sub teste{
 17  print '#' x 40,"\n";
 18  print "$_ \n" for keys %main::;
 19  print ($main::var1 or "Variável \$var1 não é visível
aqui!\n");
```

```
20  print $main::var2,"\n"; #$var2 é visível.
21  print $var3,"\n";
22 }
```

saída :

```
#Parte da saída foi omitida
var3 => *main::var3
version:: => *main::version::
stderr => *main::stderr
utf8:: => *main::utf8::
stdout => *main::stdout
stdin => *main::stdin
ARGV => *main::ARGV
INC => *main::INC
ENV => *main::ENV
Regexp:: => *main::Regexp::
UNIVERSAL:: => *main::UNIVERSAL::
teste => *main::teste
$ => *main::$
main:: => *main::main::
var2 => *main::var2
PerlIO:: => *main::PerlIO::
BEGIN => *main::BEGIN
! => *main::!
var1 => *main::var1
STDOUT => *main::STDOUT
IO:: => *main::IO::
Exporter:: => *main::Exporter::
STDERR => *main::STDERR
Internals:: => *main::Internals::
STDIN => *main::STDIN
DB:: => *main::DB::
My local
#######################################
var3
version::
/
```

```
stderr
utf8::
stdout
stdin
ARGV
INC
ENV
Regexp::
UNIVERSAL::
teste
$
main::
var2
Win32::
PerlIO::
BEGIN
var1
STDOUT
IO::
Exporter::
STDERR
STDIN
DB::
Variável $var1 não é visível aqui!
My local
Wow...
```

Você deve perceber, executando o script do exemplo11.1, que os nomes das funções e variáveis acessíveis em ***main*** são chaves no hash ***%main::*** . Caso você adicione um módulo com a cláusula ***use***, poderá ver que o hash ***%main::*** passa a conter o módulo, suas dependências e as funções que seu programa importar do módulo. E assim como ***%main::***, cada módulo criado com ***package*** tem seu namespace próprio, mesmo quando o pacote é criado no mesmo arquivo:

```
1 #exemplo11.2
2 package main;{
3  use 5.12.0;
```

```
  4
  5  say "hello";
  6  my $var1 = 1000;
  7  our $var2 = "Sou uma variável de main";
  8 }
  9 package teste; {
 10  use 5.12.0;
 11  say "Hello?";
 12  say $var2;
 13 }

saída:

Global symbol "$var2" requires explicit package name at H:/
Users/thiago/workspace/Curso/exemplo3.pl line 13.
Execution of H:/Users/thiago/workspace/Curso/exemplo3.pl
aborted due to compilation errors.
```

Veja que ***$var2***, mesmo declarada como global, não é visível em outro módulo, a não ser explicitamente. Modifique o código de **package teste** do exemplo11.2 para:

```
  9 package teste; {
 10  use 5.12.0;
 11  say "Hello?";
 12  say $main::var2;
 13 }

saída:

hello
Hello?
Sou uma variável de main
```

Porém, as variáveis declaradas com ***my*** são invisíveis mesmo com a utilização do nome do pacote. Logo, ***say $main::var1;*** gera um erro. E assim como existe **%main::**, existe **%teste::**. Haverá um hash ***%nome_do_pacote::*** para cada pacote do seu programa.

11.2 O hash %INC

Outra variável especial a considerar é o hash ***%INC***, que contém todos os módulos carregados pelo seu programa com a cláusula ***use*** ou como dependência dos módulos solicitados.

```
1 #exemplo11.3
2 use 5.012;
3 use Net::FTP;
4
5 say "$_ => $INC{$_}" for sort keys %INC;
```

saída:

```
Exporter/Heavy.pm => C:/strawberry5.12/perl/lib/Exporter/
Heavy.pm
Fcntl.pm => C:/strawberry5.12/perl/lib/Fcntl.pm
IO.pm => C:/strawberry5.12/perl/lib/IO.pm
IO/Handle.pm => C:/strawberry5.12/perl/lib/IO/Handle.pm
IO/Socket.pm => C:/strawberry5.12/perl/lib/IO/Socket.pm
IO/Socket/INET.pm => C:/strawberry5.12/perl/lib/IO/Socket/
INET.pm
IO/Socket/UNIX.pm => C:/strawberry5.12/perl/lib/IO/Socket/
UNIX.pm
Net/Cmd.pm => C:/strawberry5.12/perl/lib/Net/Cmd.pm
Net/Config.pm => C:/strawberry5.12/perl/lib/Net/Config.pm
Net/FTP.pm => C:/strawberry5.12/perl/lib/Net/FTP.pm
SelectSaver.pm => C:/strawberry5.12/perl/lib/SelectSaver.pm
Socket.pm => C:/strawberry5.12/perl/lib/Socket.pm
Symbol.pm => C:/strawberry5.12/perl/lib/Symbol.pm
Time/Local.pm => C:/strawberry5.12/perl/lib/Time/Local.pm
XSLoader.pm => C:/strawberry5.12/perl/lib/XSLoader.pm
autoflush_epic.pm => H:/Users/thiago/workspace/.metadata/.
plugins/org.epic.debug/autoflush_epic.pm
constant.pm => C:/strawberry5.12/perl/lib/constant.pm
feature.pm => C:/strawberry5.12/perl/lib/feature.pm
strict.pm => C:/strawberry5.12/perl/lib/strict.pm
```

```
utf8.pm => C:/strawberry5.12/perl/lib/utf8.pm
vars.pm => C:/strawberry5.12/perl/lib/vars.pm
warnings.pm => C:/strawberry5.12/perl/lib/warnings.pm
warnings/register.pm => C:/strawberry5.12/perl/lib/warnin-
gs/register.pm
```

11.3 O array @INC

O array @INC contém o ***path*** das bibliotecas do Perl.

NOTA: **Path** é o caminho ou local onde se encontra um determinado arquivo. Os paths em ***@INC*** indicam onde estão os arquivos de módulos de que o seu programa precisa.

```
C:\Users\thiago>perl -e "print qq/$_ \n/, for @INC"
C:/strawberry5.12/perl/site/lib
C:/strawberry5.12/perl/vendor/lib
C:/strawberry5.12/perl/lib
```

Era comum, em programas escritos antes do pragma ***use lib*** fazer parte do Perl, programadores utilizarem um bloco ***BEGIN*** onde eles adicionavam valores ao array ***@INC***.

```
BEGIN{
     push @INC, "path_da_biblioteca_particular"
}
```

Mas hoje é mais recomendado que utilize ***use lib ("path_da_minha_lib").*** Exemplo:

```
use lib ("/home/thiago/perl_lib");
use MeuModulo;
```

NOTA: BEGIN é utilizado para executar código em tempo de compilação para setups, como checar se módulos existem em seu sistema, a versão do seu sistema operacional e fazer a carga de módulos de forma seletiva.

Ah, apenas para não passar em branco: se quiser usar um módulo em um liner, basta utilizar o argumento M e o nome do Módulo:

```
>perl -MCGI=:all -e "print header,start_html"

Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
         "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitio-
nal.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>
```

No exemplo anterior carregamos o módulo CGI juntamente com suas funções e geramos o ***header*** de uma página ***html***.

11.4 Criando seus próprios módulos

Agora que você está familiarizado com os módulos do CPAN e com a criação de funções, está na hora de criar seus próprios módulos. O processo é bem simples: todo arquivo de módulo deve seguir a seguinte receita:

1. O arquivo deve ter a extensão ***.pm;***
2. A primeira declaração do arquivo será ***package Nome_do_Módulo;***
3. O nome do arquivo será o mesmo que o do módulo;
4. Se você quiser exportar as funções deve adicionar a linha: ***use base (Exporter)***;
5. Identificação da versão;

6. Indicar quais funções serão exportadas, se alguma o for;
7. Por questões históricas, a última linha do arquivo deve conter um valor booleano verdadeiro, tradicionalmente 1.

Então, para criar seu primeiro módulo, digamos, ***MyListUtils***, basta criar o arquivo ***MyListUtils.pm***:

```
package MyListUtils;
use 5.12.0;
use base (Exporter); #se você quiser exportar as funções

our $VERSION = 0.1; #Versão do módulo
our @EXPORT = (' my_sub_utils1'); #exportada automaticamente

#Exportados com a solicitação do usuário ex:
# MyListUtils ('my_sub_utils2');
our @EXPORT_OK = (' my_sub_utils2',  'my_sub_utils1');

#Exportados por meio de tags, ex:
#MyListUtils (":all");

our %EXPORTER_TAGS = (all => [ qw/my_sub_utils1 my_sub_
utils2 my_sub_utils3/]);

sub my_sub_utils1{...}
sub my_sub_utils2{...}
sub my_sub_utils3{...}
1
```

NOTA: No Perl 5.12 a variável our ***$VERSION*** pode ser substituída declarando-se a versão do módulo juntamente com o nome na cláusula ***package.***

Eis um exemplo prático:

```
1 #exemplo11.4
2 package MeuPing 0.01;
3 use base('Exporter');
4 use Scalar::Util qw/looks_like_number/;
```

```
  5 use Net::Ping;
  6 use Params::Validate qw(validate ARRAYREF SCALAR);
  7
  8 #As variáveis abaixo são especiais e devem ser globais
mesmo!!!
  9
 10 our $VERSION = 0.01;
 11 our @EXPORT = ('ping_http');
 12 our @EXPORT_OK = qw/ping_mail ping_ftp/;
 13 our %EXPORT_TAGS = (all => [qw/ping_http ping_ftp ping_
mail/] );
 14
 15 sub ping_http{
 16  check_args(@_);
 17  my %args = @_;
 18  my %retorno;
 19  for my $host(@{$args{hosts}}){
 20        @{$retorno{$host}}  = ping($host, 80,
$args{tempo});
 21  }
 22  return \%retorno;
 23 }
 24
 25 sub ping_ftp{
 26  check_args(@_);
 27  my %args = @_;
 28  my %retorno;
 29  for my $host(@{$args{hosts}}){
 30        @{$retorno{$host}}  = ping($host, 21,
$args{tempo});
 31  }
 32  return \%retorno;
 33 }
 34
 35 sub ping_mail{
 36  check_args(@_);
 37  my %args = @_;
 38  my %retorno;
```

```
 39  for my $host(@{$args{hosts}}){
 40        @{$retorno{$host}}  = ping($host, 25,
$args{tempo});
 41  }
 42  return \%retorno;
 43 }
 44
 45 sub ping{
 46  my $ping = Net::Ping->new();
 47  $ping->port_number($_[1]);
 48  $ping->hires();
 49  my @return = $ping->ping($_[0], $_[2]);
 50  $ping->close();
 51  return @return;
 52 }
 53
 54 sub check_args{
 55  validate(@_,{hosts => {type => ARRAYREF},
 56                    tempo => {type => SCALAR}} );
 57  %args = @_;
 58  die "Tempo deve ser numérico" unless looks_like_number
$args{tempo};
 59 }
 60
 61 1
```

E o arquivo que usa o módulo, que eu chamei de pingador.pl:

```
  1 #exemplo11.5
  2 use lib ".."; #aqui vai o path da sua biblioteca parti-
cular
  3 use MeuPing qw/ping_ftp/; #ping_ftp não é importada au-
tomaticamente.
  4
  5 $resultado = ping_ftp(hosts => ['ticursos.net', 'ftp.
freebsd.org'],
  6             tempo => 1); #A função deve ser importada
```

```
  7 for my $hosts (keys %$resultado){
  8  printf "$hosts, %s:21 está %s, tempo de resposta é
%.3f\n",
  9        $resultado->{$hosts}[2], $resultado->{$hosts}[0]
? 'up' : 'down',
 10        $resultado->{$hosts}[1];
 11 }
 12 print "A versão é: ", $MeuPing::VERSION, "\n";

saída:

ftp.freebsd.org, 149.20.64.73:21 está up, tempo de resposta
é 0.211
ticursos.net, 174.120.239.219:21 está up, tempo de resposta
é 0.231
A versão é: 0.01
```

Até aqui tudo legal, não é mesmo? Agora você sabe como criar seus próprios módulos... mas... esse módulo que eu criei tem muito código repetido, e programadores Perl realmente não gostam de código repetido. Todas as funções têm:

```
check_args(@_);
my %args = @_;
my %retorno;
for my $host(@{$args{hosts}}){
```

Por isso os desenvolvedores do Perl criaram mais uma pequena mágica... o AUTOLOAD:

11.5 A função AUTOLOAD

A função ***AUTOLOAD*** é carregada quando uma função não declarada é evocada dentro do pacote. Veja a mágica:

```
  1 #exemplo11.6
  2 use 5.012;
  3
  4 teste(1,2,3);
```

```
  5 carro('volks', 'fiat');
  6
  7 sub AUTOLOAD{
  8  my $sub_name = our $AUTOLOAD;
  9  say "Você evocou a função $sub_name(@_), que ainda não
foi definida ";
 10 }

saída:

Você evocou a função main::teste(1 2 3), que ainda não foi
definida
Você evocou a função main::carro(volks fiat), que ainda não
foi definida
```

Claro que você notou que o ***AUTOLOAD*** também aceita os argumentos... e se nós refizermos as funções do nosso pingador e utilizarmos o ***AUTOLOAD***? Vamos fazer um teste:

```
  1 #exemplo11.7
  2 use 5.12.0;
  3
  4 use Net::Ping;
  5 use Params::Validate qw(validate ARRAYREF SCALAR);
  6 use Scalar::Util qw/looks_like_number/;
  7
  8 my $resultado = ping_mail(hosts => ['ticursos.net',
'codeagro.sp.gov.br'],
  9  tempo => 1); #A função deve ser importada
 10
 11 for my $hosts (keys %$resultado){
 12  printf "$hosts, %s:21 está %s, tempo de resposta é
%.3f\n",
 13        $resultado->{$hosts}[2], $resultado->{$hosts}[0]
? 'up' : 'down',
 14        $resultado->{$hosts}[1];
 15 }
 16
```

```
 17 sub AUTOLOAD{
 18
 19  my @sub_name = split /_/, (our $AUTOLOAD);
 20  die "Erro, função $AUTOLOAD desconhecida" if not
'main::ping' eq  $sub_name[0];
 21
 22  my %args = @_;
 23  my (%retorno, $port_number);
 24
 25  given($sub_name[1]){
 26        when($_ eq 'http'){$port_number = 80}
 27        when ($_ eq 'ftp'){$port_number = 21}
 28        when ($_ eq 'mail'){$port_number = 25}
 29        default {die "Função $AUTOLOAD não definida!\n"}
 30  }
 31
 32  for my $host(@{$args{hosts}}){
 33        @{$retorno{$host}}  = ping($host, $port_number,
$args{tempo});
 34  }
 35
 36  return \%retorno;
 37 }
 38
 39
 40 sub ping{
 41  my $ping = Net::Ping->new();
 42  $ping->port_number($_[1]);
 43  $ping->hires();
 44  my @return = $ping->ping($_[0], $_[2]);
 45  $ping->close();
 46  return @return;
 47 }
```

saída:

codeagro.sp.gov.br, 200.144.28.66:21 está up, tempo de resposta é 0.015
ticursos.net, 174.120.239.219:21 está up, tempo de resposta é 0.231

O que aconteceu aqui? ***ping_mail*** não é uma função definida em ***main***! Então a função ***AUTOLOAD*** foi acionada com o nome ***ping_mail***. Transformamos a string ***main::ping_mail*** em um array ***(main::ping, mail)*** com a função ***split*** e verificamos se existe a string ***main::ping*** no nome da função chamada. Se não existir, o programa encerra com erro.

Em seguida, verificamos o segundo termo do nome da função. São válidos ***mail***, ***http*** e ***ftp.*** Qualquer nome diferente é considerado inválido e o programa encerra. Como ***mail*** é válido, o Perl executa a estrutura ***give/when*** e associa um valor para ***$port_number.*** A função ***ping*** é chamada e executada com os valores de porta, host e tempo fornecidos. Legal, né?

Agora vamos reescrever nosso módulo com AUTOLOAD:

```
  1 #exemplo11.8
  2 package MeuPing 0.02; #Nova versão
  3 use 5.12.0;
  4
  5 use base('Exporter');
  6 use Scalar::Util qw/looks_like_number/;
  7 use Net::Ping;
  8 use Params::Validate qw(validate ARRAYREF SCALAR);
  9
 10 our $VERSION = 0.1;
 11 our @EXPORT = ('ping_http');
 12 our @EXPORT_OK = qw/ping_mail ping_ftp/;
 13 our %EXPORT_TAGS = (all => [qw/ping_http ping_ftp ping_
mail/] );
 14
 15 sub AUTOLOAD{
 16  my @sub_name = split /_/, (our $AUTOLOAD);
 17  die "Erro, função $AUTOLOAD desconhecida"
 18        if not 'MeuPing2::ping' eq $sub_name[0];
 19
 20  my %args = @_;
 21  my (%retorno, $port_number);
 22
 23  given($sub_name[1]){
 24        when($_ eq 'http'){$port_number = 80}
```

```
 25        when ($_ eq 'ftp'){$port_number = 21}
 26        when ($_ eq 'mail'){$port_number = 25}
 27        default {die "Função $AUTOLOAD não definida!\n"}
 28  }
 29  for my $host(@{$args{hosts}}){
 30        @{$retorno{$host}}  = ping($host, $port_number,
$args{tempo});
 31  }
 32  return \%retorno;
 33 }
 34
 35 sub ping{
 36  my $ping = Net::Ping->new();
 37  $ping->port_number($_[1]);
 38  $ping->hires();
 39  my @return = $ping->ping($_[0], $_[2]);
 40  $ping->close();
 41  return @return;
 42 }
 43 1;
```

Bem menor! A função ***AUTOLOAD*** é um recurso muito bacana do Perl, mas tem suas desvantagens:

- Uma função inválida só é descoberta em tempo de execução;
- A execução é (imperceptivelmente) mais demorada, uma vez que implica um número maior de checagens durante a execução;
- Pode elevar o tempo de debug de um programa;
- Evite usar em profundidade... imagine um módulo que usa AUTOLOAD chama uma função que é AUTOLOAD em módulo de suas dependências e assim por diante. Pode elevar o tempo de execução do programa e o tempo de debug.

Exercícios

1. Incremente os scripts exemplo11.7 e exemplo11.8 para que eles possam testar as portas telnet, ssh e https.

2. Monte um script que utilize sua nova versão do MeuPing para testar hosts e portas de sites na internet ou na rede local informados pelo usuário.

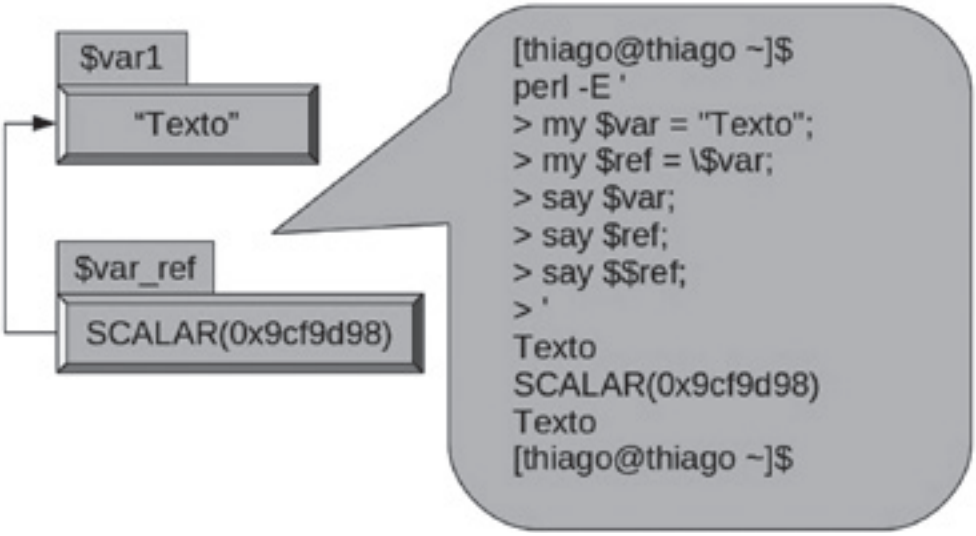
3. Monte um pequeno módulo que exporte as funções latir e comer. Fica mais bacana se você utilizar um sintentizador de voz suportado pelo módulo PerlSpeak, como o festival (veja a lista no CPAN). Caso não seja possível, as funções devem imprimir apenas "au-au" e "chomp-chomp".

4. Monte um módulo que exporte as funções: area_do_quadrado, area_do_triangulo, area_do_retangulo. Monte um script que use essas funções.

Capítulo 12. Referências

Já trabalhamos um bom bocado no decorrer deste livro com referências, de forma que você consegue intuitivamente trabalhar com arrays multidimensionais, hashes e objetos criados por módulos do CPAN. Mas existem ainda alguns truques que precisam ser apresentados, como funções que retornam outras funções ou que recebem referências a outras funções como argumentos. Assim, este capítulo visa consolidar os conhecimentos em referências e preparar você para criar seus módulos orientados a objetos.

A primeira coisa a reforçar sobre as referências é seu funcionamento. As referências são variáveis cujo valor é o endereço de outra variável:

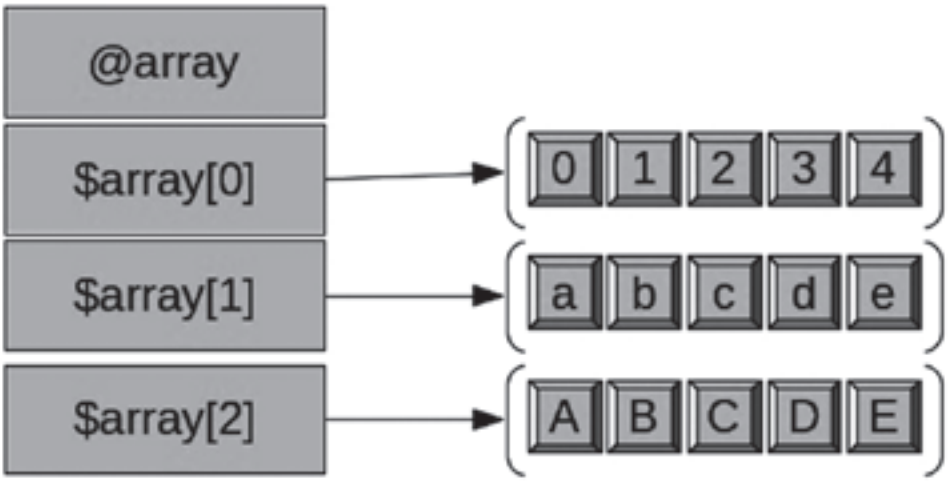


Neste exemplo é fácil perceber que a variável **$ref** contém como valor uma posição de memória e não um valor. Se quisermos acessar o valor referenciado utilizamos **$$ref**. Mas referenciar escalares que contenham números ou strings não é muito produtivo. A criação de referências é interessante para trabalharmos com objetos, filehandles, estruturas complexas ou mesmo para enviar código como argumento de uma função ou como valor de retorno de outra função.

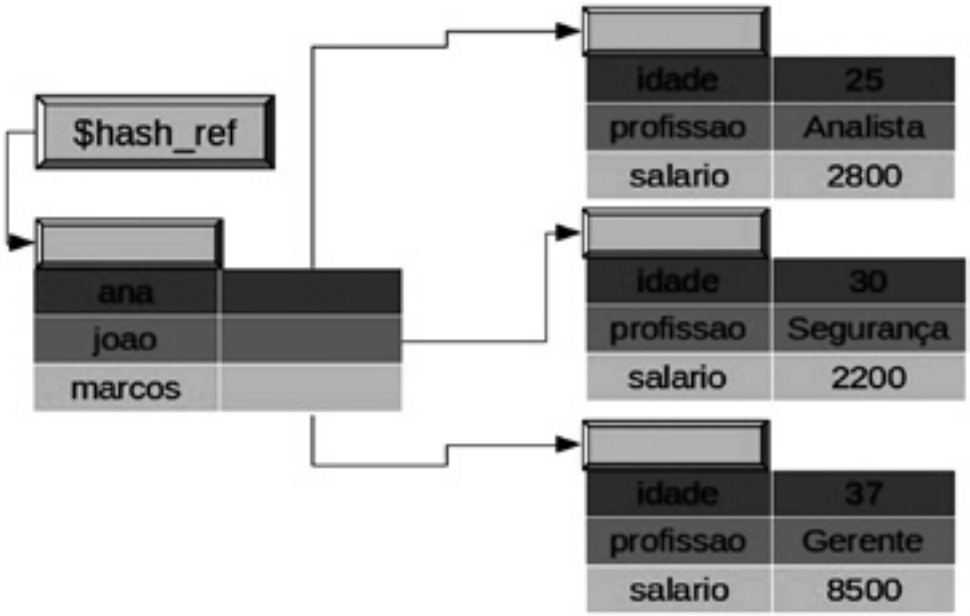
O Perl, internamente, trabalha apenas com dados planos. Então, quando criamos **arrays** multidimensionais, estamos criando um **array de referências**:

```
  1 #exemplo12.1
  2 use 5.12.0;
  3 my @array = ([0..9], ["a"..."z"],["A"..."Z"]);
  4 say $array[2][23];

saída:

X
```

'X' é o caractere armazenado no elemento 23 do segundo array interno a @array.



E quando criamos uma referência a um tipo mais complexo, ou mesmo um **hash** anônimo, o Perl cria uma estrutura de referências na memória.



Vamos fazer um exemplo:

```
  1 #exemplo12.2
  2 use 5.12.0;
  3
  4 my $hash_ref = {
  5 ana => { idade=>25, profissao=>"Analista",salar
io=>2800},
  6 joao => {idade =>30, profissao=>"Segurança",salar
io=>2200},
  7 marcos => {idade=>37, profissao=>"Gerente",salar
io=>8500}
  8 };
  9 1;
```

Se analisarmos o código a seguir no **debug** veremos como esta estrutura é montada na memória:

```
 $ perl -d exemplo12.2.pl

Loading DB routines from perl5db.pl version 1.33
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more
help.

main::(exemplo12.2.pl:4):       my $hash_ref = {
main::(exemplo12.2.pl:5):       ana => { idade=>25, profiss
ao=>"Analista",salario=>2800},
main::(exemplo12.2.pl:6):       joao => {idade =>30, profis
sao=>"Seguran?a",salario=>2200},
main::(exemplo12.2.pl:7):       marcos => {idade=>37, profi
ssao=>"Gerente",salario=>8500}
  DB<1>

  DB<1> n
main::(exemplo12.2.pl:9):       1;
```

Repare que ***$hash_ref*** é uma referência para HASH(0x3be4e0), que contém as chaves ana, joao e marcos. Cada chave armazena uma referência para outro hash.

```
  DB<1> x $hash_ref
0  HASH(0x3be4e0)
   'ana' => HASH(0x162140)
      'idade' => 25
      'profissao' => 'Analista'
      'salario' => 2800
   'joao' => HASH(0x162180)
      'idade' => 30
      'profissao' => 'Seguran?a'
      'salario' => 2200
```

```
    'marcos' => HASH(0x3be3f0)
       'idade' => 37
       'profissao' => 'Gerente'
       'salario' => 8500
  DB<2>
```

As referências permitem que montemos estruturas de dados complexas na memória do programa. Estas estruturas complexas podem ser exportadas para arquivos e importadas novamente com o módulo Storable.

12.1 Referências a funções

O Perl permite que criemos referências a funções, nos dando a possibilidade de passar funções como argumentos para outras funções ou mesmo como valor de retorno de uma função, uma vez que uma referência é um escalar como outro qualquer. Um módulo muito utilizado no CPAN, que usa referência a funções como argumento, é o ***File::Find***. Este módulo exporta uma função chamada ***find,*** que recebe um diretório ou um **array** com nomes de diretórios e uma referência a uma função, o que chamamos de callback.

Veja:

```
  1 #exemplo12.3
  2 use 5.12.0;
  3 use File::Find;
  4
  5 sub refered { #Imprime nome de arquivos que combinam
com a regex
  6  say "$File::Find::name" if /\.pl$/;
  7 }
  8
  9 find(\&refered, "/home/thiago");#Informa o diretório de
pesquisa
```

Este **script** irá mostrar na tela todos os arquivos terminados com .pl que estejam em /home/thiago.

```
/home/thiago/workspace/LinuxInRio/simplefcgi.pl
/home/thiago/workspace/LinuxInRio/oldcgi.pl
/home/thiago/workspace/LinuxInRio/simplecgi.pl
/home/thiago/workspace/TICursos/mthread.pl
/home/thiago/workspace/TICursos/pipe_in.pl
/home/thiago/workspace/TICursos/pipe.pl
/home/thiago/workspace/TICursos/tid2.pl
/home/thiago/workspace/TICursos/cook.pl
/home/thiago/workspace/TICursos/semaforo.pl
/home/thiago/workspace/TICursos/race.pl
/home/thiago/workspace/TICursos/mthread_cook.pl
/home/thiago/workspace/TICursos/semaphores.pl
/home/thiago/workspace/TICursos/tid.pl
/home/thiago/workspace/TICursos/queue.pl
/home/thiago/workspace/TICursos/errno.pl
/home/thiago/workspace/TICursos/singletr.pl
/home/thiago/workspace/TICursos/shared.pl
/home/thiago/workspace/TICursos/exec.pl
```

Podemos ainda criar uma função que retorne outra função:

```
  1 #exemplo12.4
  2 use 5.12.0;
  3 use File::Find;
  4
  5 sub refered {
  6   my $count = 0;
  7   return sub { say ++$count, ": $File::Find::name" if
/\.pl$/};
  8 }
  9
 10 my $func_ref = refered(  );
 11 say $func_ref;
 12 find($func_ref, '/home/thiago');
```

```
saída:

39: /home/thiago/workspace/LinuxInRio/simplefcgi.pl
40: /home/thiago/workspace/LinuxInRio/oldcgi.pl
41: /home/thiago/workspace/LinuxInRio/simplecgi.pl
42: /home/thiago/workspace/TICursos/mthread.pl
43: /home/thiago/workspace/TICursos/pipe_in.pl
44: /home/thiago/workspace/TICursos/pipe.pl
45: /home/thiago/workspace/TICursos/tid2.pl
46: /home/thiago/workspace/TICursos/cook.pl
47: /home/thiago/workspace/TICursos/semaforo.pl
48: /home/thiago/workspace/TICursos/race.pl
49: /home/thiago/workspace/TICursos/mthread_cook.pl
50: /home/thiago/workspace/TICursos/semaphores.pl
```

Veja que a função **refered** cria **count**, que é incrementada pela função retornada por **refered**.

12.2 Verificando estruturas com o Data::Dumper

O módulo Data::Dumper permite que verifiquemos em detalhes o conteúdo interno das estruturas complexas em tempo de execução, de forma semelhante ao debugger:

```
  1 #exemplo12.5
  2 use 5.12.0;
  3
  4 use Data::Dumper;
  5 my @array = ([0..9], ["a"..."f"],["A"..."F"], "casa",
"cachorro", "ap");
  6 say Dumper @array;

saída:

$VAR1 = [
          0,
```

```
            1,
            2,
            3,
            4,
            5,
            6,
            7,
            8,
            9
          ];
$VAR2 = [
            'a',
            'b',
            'c',
            'd',
            'e',
            'f'
          ];
$VAR3 = [
            'A',
            'B',
            'C',
            'D',
            'E',
            'F'
          ];
$VAR4 = 'casa';
$VAR5 = 'cachorro';
$VAR6 = 'ap';
```

12.3 Armazenando estruturas complexas

O módulo ***Storable*** permite armazernar em arquivos estruturas de dados complexas como arrays de arrays, hashes de arrays ou arrays de hashes para depois recarregar a estrutura na memória em execuções futuras ou exportá-las para outro programa Perl. Vamos fazer um pequeno programa que gera uma tabela de registro de funcionários. Este programa irá armazenar esta tabela em um arquivo com o módulo Storable e também irá carregar os elementos na memória sob forma de uma **ARRAY_REF:**

```
  1 #exemplo12.6
  2 use 5.12.0;
  3 use Storable;
  4 use Data::Dumper;
  5 use constant ARQUIVO => 'registro_funcional'; #arquivo
com os dados
  6
  7 #Vamos montar uma estrutura similar a uma tabela de
banco de dados com as
  8 #informações: chave, funcionário, cargo e salário. Esta
estrutura será um
  9 #array e cada registro um array anônimo interno ao ar-
ray de registros
 10
 11 my $table;
 12 my $lastkey = 0; #Última chave válida
 13
 14 if (-e ARQUIVO){
 15 #Se o arquivo já existir, sua estrutura será carregada
na memória com a
 16 #função retrieve, exportada pelo módulo Storable
 17  $table = retrieve ARQUIVO;
 18  #my $elements_number = scalar @$table;
 19  $lastkey = $table->[1 - scalar @$table][0];
 20 #primeiro elemento do último array de $tabela é a últi-
ma chave utilizada
 21 }
 22
 23 do{
 24  my $choose ; #Escolha do usuário no menu de opções
 25
 26  #Imprime o menu do usuário
 27  print <<MENU;
 28 O que você deseja:
 29 (a) Adicionar um elemento
 30 (l) Ver a lista de elementos
 31 (d) Deletar um elemento
 32 (s) Salvar alterações
```

```
 33 (p) Imprimir Dumper
 34 (q) Sair
 35 MENU
 36  chomp ($choose = <STDIN>);
 37
 38  #Lê a opção do usuário e executa
 39  given ($choose){
 40        when ($_ eq 'a') {&add_elemento}#adiciona item
ao registro
 41
 42        when ($_ eq 'l') { foreach my $registro (@$ta-
ble){
 43                          #imprime lista de itens
 44                          print "@{$registro}\n";
 45                          }
 46                       say ""} #imprime uma linha em
branco
 47
 48        when ($_ eq 'd') { &remove_elemento} #remove
item
 49        when ($_ eq 's') {store $table , "registro_fun-
cional"} # Salva
 50        when ($_ eq 'p') {print Dumper $table} #Imprime
dump da lista
 51        when ($_ eq 'q') {exit}
 52        default {say "Valor inválido. Escolher um item
do menu."}
 53  }
 54 }while(1);
 55
 56 sub add_elemento{
 57  my @registro;#registro de elementos
 58  print "Informe o nome do funcionário: ";
 59  chomp(my $funcionario = <STDIN>);
 60
 61  print "Informe o cargo: ";
 62  chomp(my $cargo = <STDIN>);
 63
```

```
 64  print "Informe o salário: ";
 65  chomp(my $salario = <STDIN>);
 66
 67  #Adiciona item à lista e incrementa a chave
 68  push @$table, [++$lastkey, $funcionario, $cargo, $sa-
lario];
 69 }
 70
 71 sub remove_elemento{
 72  print "Informe a chave do elemento a ser removido ou 0
para cancelar: ";
 73  chomp (my $chave = <STDIN>);
 74  given ($chave){
 75        when (0) {say "operação cancelada"; return}
 76        when (/\d+/) { splice @$table, --$chave, 1;
 77                              &recalc_keys;
 78        }
 79        default{print "chave informada não é numérica";
return;}
 80  }
 81 }
 82
 83 sub recalc_keys{
 84  my $key = 0;
 85  for my $registro (@$table){
 86        $registro->[0] = ++$key;
 87  }
 88  $lastkey = $key;
 89 }
```

saída:

```
O que vc deseja:
(a) Adicionar um elemento
(l) Ver a lista de elementos
(d) Deletar um elemento
(s) Salvar alterações
(p) Imprimir Dumper
```

```
(q) Sair
l
1 Thiago Glauco Sanchez Analista Sr. 4500.00
2 Luciana Sanchez Pedagoga 5600.00
3 Frodo Baggins Bolseiro 3700.00

O que vc deseja:
(a) Adicionar um elemento
(l) Ver a lista de elementos
(d) Deletar um elemento
(s) Salvar alterações
(p) Imprimir Dumper
(q) Sair
p
$VAR1 = [
          [
            '1',
            'Thiago Glauco Sanchez',
            'Analista Sr.',
            '4500.00'
          ],
          [
            '2',
            'luciana sanchez',
            'Pedagoga',
            '5600.00'
          ],
          [
            '3',
            'Frodo Baggins',
            'Bolseiro',
            '3700.00'
          ]
        ];
O que vc deseja:
(a) Adicionar um elemento
(l) Ver a lista de elementos
(d) Deletar um elemento
```

```
(s) Salvar alterações
(p) Imprimir Dumper
(q) Sair
q
```

Uma boa ideia de uso do ***Storable*** são sistemas cujos dados devem ser armazenados para uso futuro, principalmente arquivos que serão lidos apenas por aplicativos escritos em Perl, como o estado de um jogo onde o usuário pode continuar da mesma fase onde parou na última execução. Com este módulo nós economizamos o trabalho de manusear os diferentes formatos de arquivos com expressões regulares e já carregamos os dados no formato de trabalho.

12.4 Um exemplo com Tk

Outro módulo importante do Perl que trabalha com referências é o **Tk**. O módulo Tk usa referências de funções para tratar dos eventos da interface gráfica.

```
  1 #exemplo12.7
  2 use 5.12.0;
  3 use Tk;
  4 use strict;
  5
  6 #Cria a janela principal do programa
  7 my $mw = new MainWindow;
  8
  9 #O botão executa uma função referenciada no argumento
-command
 10 my $btn = $mw->Button(-text=>"Exibe um Dialog Box",
 11                       -command => \&sair)->pack();
 12 MainLoop;
 13
 14 #função enviada via referência para o objeto $btn
 15 sub sair {
 16 #Exibe uma caixa de mensagem para o usuário.
 17  my $resposta = $mw -> messageBox(-message=>"Deseja Re-
almente Sair??",
```

```
 18                                  -type=>'yesno',-
icon=>'question');
 19  if( $resposta eq "Yes" ) {
 20        exit 0;
 21  }
 22  else {
 23        $mw->messageBox(-type=>"ok",
 24        -message=>"Gostou desta Aplicação, hein!!!");
 25  }
 26 }
```

Exercícios

1. Refaça o exemplo12.4. Desta vez procure no CPAN um módulo que controle a tela do console como o Curses::UI.

2. Monte um aplicativo como o comando find do Linux, que pesquisa por um nome de arquivo em um diretório, ambos informados pelo usuário.

Capítulo 13. Programação Orientada a Objetos com Perl

13.1 Por que usar programação orientada a objetos?

OOP confunde muitos programadores. Isso acontece por causa de abordagens excessivamente teóricas e por quebra do paradigma da programação procedural, que por muito tempo dominou o mundo da programação com linguagens como C, Pascal e Basic. Se a programação procedural durou tanto, e ainda existe, é porque ela é boa. Então... por que você vai usar **OOP**?

Primeiro: desde que você nasceu, você entra em contato com objetos. Seu cérebro é acostumado a trabalhar com coisas que tenham propriedades, comportamentos e funcionalidades. Um carro tem suas funcionalidades: acelera, freia, vira, muda de marcha. Tem comportamento: faz 12 km por litro, acelera até 230 km/h, é estável nas curvas. E tem propriedades: cor, modelo, fabricante. As coisas são assim no mundo real e podem ser assim no mundo da programação.

Segundo: a programação **OOP** permite uma aproximação mais flexível e uma análise **top-down** do seu projeto, ou seja, permite que você facilmente defina todas as funcionalidades do seu programa, as separe em elementos e transforme esses elementos em objetos. Um programa web, por exemplo, pode ter um objeto especializado em gerar telas, um objeto especializado em cuidar das sessões dos usuário e um terceiro objeto para cuidar dos dados do seu banco.

Terceiro: você pode se concentrar na modelagem do dado e deixar os seu dados complexos no mesmo pacote que as funções que tratem estes dados. Veja o trecho de código a seguir:

```
use Carro;
my $carro = Carro->new(Cor => "Black", Fabricante => "Gur-
geo");
```

```
$carro->acelerar(+10);
$carro->virar("esquerda");
```

A velocidade atual do carro está encapsulada no objeto (guardada dentro do objeto). O código é mais fácil de ler e de manter. As estruturas de dados e as funções que tratam estes dados estão juntas dentro do módulo ***Carro.pm*** e tem um relacionamento óbvio para o programador que utiliza o módulo. Isso é muito mais fácil para um ser humano entender do que este trecho de código que utiliza código procedural:

```
use FuncoesParaCarro qw(acelerar virar);

#Você tem que criar as variáveis do carro em seu main.
#Aqui vamos montar um HASH_REF
my $carro = { Color => "Black",
          Manufacturer => "Gurgeo",
          Velocidade => 0,
          Direcao = "Em_frente"};

#E trabalhar separadamente as funções e os dados
$carro->{Velocidade} = acelerar($carro->{Velocidade} + 10);
$carro->{Direcao} = virar("esquerda", $carro->{Direcao});
```

No paradigma orientado a objetos, os dados e as funções estão encapsulados na mesma estrutura, ou seja, eu não preciso me preocupar em informar a velocidade atual para dizer que eu quero acelerar. A informação já está no objeto. Num módulo orientado a funções, os dados e as funções estão em locais diferentes, então eu tenho que dizer qual é minha velocidade atual antes de acelerar, ou minha direção atual, para solicitar ao objeto que vire.

Quarto: programas feitos com OOP são comprovadamente mais fáceis de expandir. Como os objetos encapsulam as funções e os dados num mesmo módulo, os programas que manipulam estes objetos podem facilmente ser modificados sem se preocupar com os outros módulos, e funções podem ser adicionadas por meio de herança em classes sem modificar o módulo que define a classe.

13.2 Defini ões de um objeto

Assim como no mundo real, os objetos têm certas características que devemos considerar:

Propriedades ou atributos: propriedades são as qualidades do objeto, como cor, tamanho, velocidade, que definem o estado atual do objeto. Em termos comuns, são variáveis com um significado especial dentro daquele objeto. Voltando a nossa comparação inicial com a linguagem humana, atributos são os adjetivos que caracterizam um substantivo.

Comportamento: O que um objeto pode fazer? Acelerar, frear, virar, ligar, desligar, nascer, morrer. Quando ele pode fazer? Acelerar somente se estiver abaixo da velocidade máxima. Morrer apenas se ainda estiver vivo. Frear apenas se estiver em movimento. Como fazer: frear diminuindo a velocidade. Um método é feito criando-se uma função dentro do pacote que define a classe.

Identidade: O meu carro não é o seu carro!!! Mesmo que meu carro tenha a mesma cor, modelo e fabricante que o seu, o meu carro não é o seu. Ele tem uma existência independente da existência do seu carro. E uma forma de identificação que mostra que eles são elementos distintos: um número de placa e chassi que nunca podem ser iguais. Em programação Perl, esta identidade é garantida pelo endereço de memória ocupado pelo objeto. Eis um exemplo simples de identidade:

```
 1 #exemplo13.1
 2 use 5.12.0;
 3 {
 4 package Carro;
 5 sub new{
 6  my $car = {cor => undef,
 7           modelo => undef,
 8           fabricante => undef};
 9  my $class = shift;
10  return bless $car, $class;
11  }
12 }
13
14 my $meu_carro = Carro->new( );
15 say "A identidade do meu carro eh: $meu_carro";
```

```
 16 my $seu_carro = Carro->new( );
 17 say "A identidade do seu carro eh: $seu_carro";

saída:

A identidade do meu carro eh: Carro=HASH(0x152f28)
A identidade do seu carro eh: Carro=HASH(0x153b38)
```

13.3 Existe alguma desvantagem em usar OOP?

Não existe almoço grátis neste mundo cruel. E o paradigma OOP tem suas desvantagens. A primeira desvantagem: usualmente, o módulo de uma classe pai tem um pouco mais de código que um módulo comum. A segunda desvantagem é uma degradação, ao menos em teoria, de desempenho.

Porém devo ressaltar duas coisas: mesmo quando o módulo que define a classe tiver mais código que um módulo orientado a funções, os programas que utilizarão esta classe terão menos código. Eu quero dizer que o programador que consome as classes (que usar o seu módulo) normalmente irá utilizar menos código que um programador que usou um módulo orientado a funções. Afinal, **$carro->acelerar(+10)** é mais enxuto que **$carro->{Velocidade} = acelerar($carro->{Velocidade} + 10)**.

Da mesma forma, você ou outro programador que utilizar sua classe terá menos trabalho para criar um modelo de dados. No final você economiza código. Mais adiante, quando você aprender a usar o Moose, essa sobrecarga de código será perdida.

A outra coisa que eu devo alertar diz respeito à perda de desempenho. A criação de um objeto exige processamento extra. Manter um objeto na memória, que é uma estrutura de dados bem complexa no baixo nível, também consome recurso a mais. Mas posso garantir que essa perda de desempenho é totalmente insignificante em qualquer hardware moderno (fabricado depois de 2001). Você pode ler sobre essa perda de desempenho em livros antigos, antes dos processadores Pentium 4 e Dual Core... agora Quad Core... enfim, esta perda de desempenho será imperceptível para seu usuário.

A prática diz: se você for criar um sistema pequeno, que terá poucas modificações e dificilmente alguém que não seja você irá realizar manutenções no código, use módulos orientados a funções. Para qualquer coisa diferente disso, use OOP.

NOTA: Classe é o termo OOP para definir o código que descreve o objeto. Em Perl, uma classe é um Módulo ou Pacote que contém o código do objeto. **Herança** é a capacidade de criarmos uma classe nova referenciando uma classe já existente.

13.4 Criando um objeto

Vamos iniciar, criando a classe **Person,** que representa uma pessoa. Para isso vamos definir algumas propriedades e comportamentos: uma pessoa tem **nome, idade e estado civil**, além de muitas outras características. Estas são suas propriedades. Uma pessoa pode ter diversas outras propriedades, mas vamos ficar com estas por enquanto. Definir quais propriedades de um objeto que estamos modelando são importantes é como definir quais propriedades são importantes em um banco de dados. E como comportamento, as pessoas **envelhecem, casam, divorciam-se, crescem, morrem**. Este é o comportamento do objeto. Então vamos criar nosso módulo **Person**, que definirá a classe:

```
  1 #exemplo13.2
  2 package Person;
  3 use strict;
  4 use warnings;
  5 use Carp; #Esta classe serve para facilitar o debug em
objetos e classes.
  6
  7 sub new{
  8 #quando criarmos o objeto o comando será "my $referent
= Person->new();"
  9 #O operador -> coloca a string que o antecede, no caso
Person, como
 10 #primeiro argumento do método new, ou seja, Person será
o primeiro elemento
 11 #do array @_ do método new( ). Assim a variável $class
vai conter 'Person'.
 12
 13  my $class = shift; #Puxa o primeiro elemento de @_, no
caso Person
```

```
 14  my %args = @_;
 15
 16  #Agora nós criamos as propriedades no HASH_REF $self
 17  my $self = {
 18        Name => ($args{Name} or “”),
 19        Age => ($args{Age} or 0),
 20        Marital_state => ($args{marital_state} or “sin-
gle”),
 21  };
 22  bless $self, $class; #Cria o objeto Person em $self e
retorna $self.
 23 }
 24
 25 1;
```

O método **new** é apenas um construtor. Ou seja, uma função que cria um objeto na memória por meio da função **bless.** A função **bless** recebe como argumentos: **um escalar, que contém os atributos do futuro objeto, e um segundo argumento, que é o nome da classe que estamos criando.**

Para iniciarmos o uso do nosso módulo criamos um **script** simples:

```
  1 #exemplo13.3
  2 use 5.12.0;
  3 use lib(“.”); #Procura módulos na pasta Local.
  4 use Person;
  5 use strict;
  6
  7 my $person1 = Person->new(Name => “Andrews Moore”,
  8                        Age => “single”, );
  9 #Acessando diretamente:
 10 say $person1->{Name}, “ is “ , $person1->{Marital_sta-
te};
 11 say $person1; #Mostra a identidade do objeto
CLASSE=REFERÊNCIA

saída:

Andrews Moore is single.
Person=HASH(0x83046f8)
```

Legal. Nosso objeto foi criado. O nosso construtor funcionou e podemos ver nosso objeto na memória.

Temos aqui um impasse: a nossa classe permite acesso direto às suas propriedades. Se você será o único consumidor das classes que criar, não se importará muito se o código da classe não proteger suas propriedades. Porém, é interessante que você forneça métodos para que o usuário acesse as propriedades da classe. Usualmente as interfaces fornecem um método **set** e um método **get** para cada propriedade. Adicione o código a seguir ao exemplo13.2:

```
 24
 25 sub get_name{
 26  my $self = shift;
 27  return $self->{Name};
 28 }
 29 sub set_name{
 30  my $self = shift;
 31  #A função confess é fornecida pelo módulo Carp
 32  # A regex garante que nome não tenha caracteres estra-
nhos
 33  confess "Nome informado incorretamente" if not
 34        $_[0] =~ /^[^@#\$%&*()\[\]\+\=\\~\^:;><]+$/;
 35        return $self->{Name} = $_[0];
 36 }
 37
 38 1;
```

O interessante aqui é criarmos verificações para evitar que o usuário envie dados que comprometam o funcionamento correto do seu sistema. No método ***set_name,*** por exemplo, impedimos que o usuário envie caracteres que não podem pertencer a nomes. A função ***confess*** do módulo ***Carp*** envia mensagens amigáveis ao programador que utilizar seu módulo de forma errada. Se a função ***set_name*** receber um caractere indevido, o programa vai informar ao programador:

```
Nome informado Incorretamente at Person.pm line 31
     Person::set_name('Person=HASH(0x9b256f8)', 'Jo\x{c3}\
x{a3}o da Silva#') called at /home/thiago/workspace/OOP/
populacao.pl line 15
```

A função ***confess*** retorna informações bem úteis para programadores em geral, como módulo, linha, função, argumentos enviados para as funções caso ocorra algum erro. A função ***get_name*** simplesmente retorna o valor da propriedade ***Name***. Hum... muito código somente para isso, não acha?

Vamos falar agora de outra característica da teoria da programação a objetos chamada **Sobrecarga de Métodos**. A Sobrecarga de Métodos diz que um objeto pode ter diversos métodos com o mesmo nome, desde que com argumentos diferentes, de forma que seu objeto tenha comportamentos diferentes de acordo com os argumentos de um método em si. Para o programador que consome a classe, o processo é transparente. Porém, o Perl é, na minha opinião, melhor que as outras linguagens neste ponto em particular, pois, uma vez que os argumentos são armazenados no array @_, não precisamos criar diversas funções, apenas uma que verifica as diversas possibilidades de argumentos possíveis. O trecho de código a seguir substitui as funções ***get_name*** e ***set_name***.

```
 24
 25 sub name{
 26  my $self = shift; #Retira Person de @_
 27  if(@_){
 28  #verifica se existem argumentos na chamada do método.
 29  #Se existirem, configura a variável Name com o valor
do argumento.
 30        confess "Nome informado incorretamente" if not
 31              $_[0] =~ /^[^@#\$%&*()\
[\]\+\=\\~\^:;><]+$/;
 32
 33        $self->{Name} = $_[0];
 34  }
 35 #Se não há argumentos na chamada do método, retorna o
valor da propriedade
 36 #Name
 37  return $self->{Name}
 38 }
 39
 40 1;
```

Assim, um mesmo método pode tomar diferentes ações de acordo com sua lista de argumentos, conforme definem as teorias de **Sobrecarga de Métodos.**

Agora vamos criar toda a interface da nossa classe:

```
  1 #exemplo13.4
  2 package Person;
  3 use 5.12.0;
  4 use Carp;
  5
  6 sub new{
  7 #quando criarmos o objeto o comando será "my $referent
= Person->new();"
  8 #O operador -> coloca a string que o antecede, no caso
Person, como
  9 #primeiro argumento do método new, ou seja, Person será
o primeiro elemento
 10 #do array @_ do método new( ). Assim a variável $class
vai conter 'Person'.
 11  my $class = shift;
 12  my %args = @_;
 13  my $self = { Name => ($args{Name} or ''),
 14    Age => ($args{Age} or 0), #Afinal todos nascem com 0
anos
 15    Marital_state => ($args{marital_state} or "single"),
#e solteiros.
 16  };
 17  bless $self, $class;
 18 }
 19
 20 sub name{
 21  my $self = shift;
 22  if(@_){
 23        confess "Nome informado incorretamente" if not
 24          $_[0] =~ /^[^@#\$%&*()\[\]\+\=\\~\^:;><]+$/;
 25        $self->{Name} = $_[0];
 26  }
 27  return $self->{Name}
```

```
 28 }
 29
 30 sub age{
 31  my $self = shift;
 32  if($_[0]){
 33        given($_[0]){
 34          #Caso o usuário informe algo que não seja um
número inteiro
 35          when(/\D+/) { confess "Idade deve ser um in-
teiro entre 0 e 150" }
 36          when($_ < 0 or $_ > 150) { confess "Idade deve
ser um inteiro" .
 37              "entre 0 e 150" }
 38          default{return $self->{Age} = $_}
 39        }
 40  }
 41  else{
 42        return $self->{Age};
 43  }
 44 }
 45
 46 sub get_older{
 47  my $self = shift;
 48  #Afinal as pessoas envelhecem.
 49  return ++$self->{Age};
 50 }
 51
 52 sub marital_state{
 53  my $self = shift;
 54  if($_[0]){
 55        given($_[0]){
 56          when($_ eq "married"){
 57              if($self->{Marital_state} eq "married"){
 58                    warn "Você já é casado, poligamia é
ilegal";
 59                    exit;
 60              }
 61              return $self->{Marital_state} = $_[0];
```

```
 62        }
 63        when($_ eq "single"){
 64              return $self->{Marital_state} = $_[0];
 65        }
 66        when($_ eq "divorced"){
 67              return $self->{Marital_state} = $_[0];
 68        }
 69        default{
 70              confess
 71                'Marital state must be single, married
or divorced'
 72        }
 73    }
 74  }
 75 }
 76
 77 sub get_married{
 78  my $self = shift;
 79  if($self->{Marital_state} eq "married"){
 80        warn "Você já é casado e não pode casar de
novo";
 81        exit;
 82  }
 83  return $self->{Marital_state} = "married";
 84 }
 85
 86 sub get_divorced{
 87  my $self = shift;
 88  if($self->{Marital_state} ne "married"){
 89        warn "Você não é casado, logo não pode se divor-
ciar";
 90        exit;
 91  }
 92  return $self->{Marital_state} = "married";
 93 }
 94
 95 1;
```

Agora um pequeno programa que usa a interface:

```
  1 #exemplo13.5
  2 use 5.12.0;
  3 use lib(".");
  4 use Person;
  5 use strict;
  6
  7 my $person = Person->new(Name => "Clark Kent", Age =>
30);
  8 say $person->name . " tem " . $person->age . " anos.";
  9 say $person->name . " fez aniversário e agora tem " .
$person->get_older();
 10 my $person2 = Person->new(Name => "Louis Lane", Age =>
28);
 11 $person->get_married() and $person2->get_married();
 12
 13 printf "Agora %s e %s estão casados\n", $person->name,
$person2->name
 14  if $person->marital_state() eq "married"
 15     and $person2->marital_state() eq "married";
```

13.5 Herança

Outro ponto muito importante da programação orientada a objetos é a herança. Herança é a capacidade de criar uma nova classe reaproveitando uma classe existente. A nossa classe **Person** definiu algumas propriedades e alguns métodos comuns e gerais às pessoas. Elas **nascem, ficam velhas, casam, divorciam-se, etc.**

Mas vamos imaginar um outro caso. As pessoas ficam mais velhas quando fazem aniversário. Porém, no clássico Alíce no País das Maravilhas alguns personagens faziam festas de desaniversário. Para criarmos uma classe que represente uma pessoa que faça desaniversários, não precisamos reescrever todo o código da classe **Person**, nem tampouco permitir que todas as pessoas façam desaniversários. Basta criarmos uma classe que herde todas as propriedades e métodos da classe **Person** e codificarmos as características adicionais que queremos. Para se tornar uma classe filha de uma outra classe existente basta adicionar o pragma ***use base (My_Super_Class)***, onde My_Super_Class é o nome da classe pai.

Vejamos:

```
 1  #exemplo13.6
 2  package WonderWorldPerson;
 3  use 5.12.0;
 4  use Carp;
 5  use lib(".");
 6  use base("Person"); #Aqui definimos a classe pai.
 7
 8  sub new{
 9          my $class = shift @_;
10          my %args = @_;
11
12 #Agora, chamamos o método new da classe pai, Person,
para
13 #reutilizar seu código.
14          my $self = $class->SUPER::new(@_);
15          if ( not exists $args{ Unbirth } ){
16                  $self->{ Unbirth } = 0
17          }
18          else {
19                  $self->{ Unbirth } = $args{ Unbirth };
20          }
21          return $self;
22  }
23
24  sub unbirthday{
25  #unbirth não existe em Person.
26  my $self = shift;
27
28  if ( $_[0] ) {
29        if ( ( $self->{Unbirth} + $_[0] ) >= ( 364 *
$self->{Age} ) ){
30              warn "Você só pode fazer 364 desaniversá-
rios por ano!!!";
31        }
32        else{
```

```
33                $self->{Unbirth} += $_[0];
34          }
35   }
36   else {
37          return $self->{Unbirth};
38   }
39 }
40 1;
```

Como a classe WonderWorldPerson usa Person como base, ela herda todos os métodos e propriedades de Person e sobrescreveu o construtor ***new***. Mesmo sobrescrevendo um método de uma classe pai, podemos reaproveitar o código dentro da classe filha, como fizemos na linha 11:

my $self = $class->SUPER::new(@_);

Neste caso reaproveitamos todo o código já escrito no método ***new*** de Person. Se você executar os exemplos deste capítulo com o debugger verá que ***$class->SUPER::new(@_)*** executa o método ***new*** de ***Person*** e depois retorna para o método new de ***WonderWorldPerson.*** Veja um programa que usa a classe ***WonderWorldPerson***.

```
  1 #exemplo13.7
  2 use 5.12.0;
  3 use lib(".");
  4 use strict;
  5
  6 {
  7  package RainhaDeCopas;
  8  use base "WonderWorldPerson";
  9 #Rainha de copas é uma classe filha de WonderWorldPer-
son
 10 #e herda todos o código de sua classe pai.
 11 }
 12
 13 my $queen = RainhaDeCopas->new( Name => "Rainha de Co-
pas",
 14                    Age => "35",
```

```
 15                    Marital_state => "married",
 16                    Unbirth => 15,);
 17
 18 printf"A nossa rainha tem %d de idade\n", $queen->age(
);
 19 $queen->get_older( ); #este método foi herdado
 20 printf "Essa não, a %s fez aniversário, e agora está
com %d\n",
 21  $queen->name( ), $queen->age( );
 22
 23 #Evoca o método unbirth
 24 $queen->unbirthday( 1 );
 25 printf "Três dias depois ela fez um desaniversário.\n"
.
 26        "Agora ela tem %d anos e %d desaniversarios",
 27  $queen->age( ), $queen->unbirthday(  );

saída:

A nossa rainha tem 35 de idade
Essa não, a Rainha de Copas fez aniversário, e agora está
com 36
Três dias depois ela fez um desaniversário.
Agora ela tem 36 anos e 16 desaniversarios
```

Como você viu, não é preciso modificar a classe Person nem reescrever o código já utilizado em Person para criar a classe WonderWorldPerson, que contém todos os métodos e propriedades de Person. A classe RainhaDeCopas é uma classe filha de WonderWorldPerson. A herança pode ocorrer infinitamente, em teoria.

13.6 Atributos e métodos de classe e os destrutores

Outro ponto importante na teoria da OOP são os atributos de classe, ou seja, atributos que pertencem à classe e não a objetos individuais. Os atributos de classe são interessantes para:

1. Controlar a quantidade de objetos criados de uma determinada classe;
2. **Definir dados importantes para um grupo**, como o número de indivíduos de uma população, ou a quantidade de dinheiro dispensado por uma rede de caixas eletrônicos;

3. **Criar um algoritmo para localizar objetos**;
4. **Definir uma propriedade especial baseada na ordem de criação de um elemento**, como um número de série;
5. Criar uma conexão única (a mesma para todos os objetos) com um banco de dados, socket ou arquivo de log.

Para definirmos um atributo de classe, basta criar uma variável global no módulo classe que esteja fora de qualquer função. Vamos modificar o início da nossa classe Person e criar um atributo ***população***.

NOTA: As modificações feitas em uma classe não afeta a sua classe pai, mas afeta todas as suas herdeiras. Assim, ao modificar **Person**, as classes **WonderWorldPerson** e **RainhaDeCopas** serão afetadas.

Modifique o método new de Person e adicione o método population ao final do arquivo da classe, conforme o exemplo13.8.

```
  1 #exemplo13.8
  2 package Person;
  3 use 5.12.0;
  4 use Carp;
  5
  6 our $population = 0; #Propriedade pertencente à classe
  7
  8 sub new{
  9 #quando criarmos o objeto o comando será "my $referent
= Person->new();"
 10 #O operador -> coloca a string que o antecede, no caso
Person, como
 11 #primeiro argumento do método new, ou seja, Person será
o primeiro elemento
 12 #do array @_ do método new( ). Assim a variável $class
vai conter 'Person'.
 13  my $class = shift;
 14  my %args = @_;
```

```
 15  ++$population;
 16  my $self = {
 17    Name => ($args{Name} or "”),
 18    Age => ($args{Age} or 0),
 19    Marital_state => ($args{marital_state} or "single”)
 20  };
 21  bless $self, $class;
 22 }

...

 99 sub population{ #Função pertencente à classe
100  return $population;
101 }
102
103
104 1;
```

Essa mudança é bem simples: criamos uma variável global à classe, **$population**, que é incrementada toda vez que alguma pessoa nova é criada. A variável pode ser lida pelo método de classe **population**. Como método de classe, population pode ser acessado por qualquer objeto, como também pela notação **Person->population;**.

Um pequeno programa para vermos isso em ação:

```
  1 #exemplo13.9
  2 use 5.12.0;
  3 use lib(".");
  4 use Person;
  5 my @habitantes;
  6 for(1..1000){
  7  push @habitantes, Person->new;
  8 }
  9
 10 #Métodos de classe podem ser acessados diretamente pela
classe
 11 printf "A população atual é de %d habitantes\n", Per-
son->population();
 12
 13 #Ou por qualquer objeto da classe
```

```
 14 printf "A população atual é de %d habitantes\n",
 15  $habitantes[2]->population();

saída:

A população atual é de 1000 habitantes
A população atual é de 1000 habitantes
```

No Perl, para destruirmos um objeto basta remover todas as referências que apontem para ele. Então, **$habitantes[2] = undef;** iria destruir o nosso objeto e teríamos um objeto Person a menos. Porém, nossa variável **$population** continuaria 1000. Precisamos definir um método que seja executado na limpeza do objeto da memória. O Perl define um método chamado ***DESTROY***. No método ***DESTROY*** você deve incluir todas as limpezas para excluir um objeto, como encerrar **pipes**, **filehandles ou conexões com banco de dados e manusear as variáveis de classe** que o objeto esteja utilizando.

No caso da nossa classe **Person**, basta decrementar a variável **$population**. Então vamos adicionar o método **DESTROY** à nossa classe **Person** logo após o método *population*.

```
102
103 sub DESTROY{
104  my $self = shift;
105  --$population;
106 }
107
108 1;
```

E ver como ele funciona:

```
  1 #exemplo13.10
  2 use lib(".");
  3 use Person;
  4
  5 my @habitantes;
  6
  7 for(1..1000){
  8  push @habitantes, Person->new;
  9 }
```

```
 10
 11 printf "A população é de %d habitantes\n", Person-
->population();
 12
 13 #Este loop é um serial killer...
 14 for (1..50){
 15  $habitantes[$_] = undef;
 16 }
 17
 18 printf "A população é de %d habitantes\n", Person-
->population();

saída:

A população é de 1000 habitantes
A população é de 950 habitantes
```

13.7 A classe SUPER

Toda classe no Perl é filha da classe **SUPER.** A classe super define alguns métodos importantes como:

- **isa(nome_da_classe)**: verifica se o objeto pertence ou não a determinada classe;
- **can(metodo)**: verifica se o objeto pode executar o método;
- **VERSION(versao)**: verifica a versão do módulo da classe. Retorna true se a versão for maior ou igual à versão solicitada, que é definida pela variável global **$VERSION** no módulo da classe.

```
  1 #exemplo13.11
  2 use 5.12.0;
  3 use lib(".");
  4 use WonderWorldPerson;
  5
  6 my $queen = WonderWorldPerson->new(Name => "Rainha de
Copas", Age => "25");
  7
  8 say "A rainha é uma pessoa" if $queen->isa("Person");
  9 say "A rainha pode casar-se" if $queen->can("get_married");
```

```
 10 say "A rainha pode enriquecer" if $queen->can("became_
rich");

saída:

A rainha é uma pessoa
A rainha pode casar-se
```

13.8 Em resumo

No Perl:

- **Uma classe é apenas um módulo** cujas funções funcionam como métodos, e um destes métodos é o construtor que cria uma referência para o objeto;
- **Um método é apenas uma função** relacionada ao tipo de dados que o objeto trabalha;
- **Os atributos são variáveis** do Perl, podendo estas ser escalares, hashes, arrays ou mesmo referências para outros objetos;
- **Um objeto é apenas uma referência** a uma estrutura complexa criada com a função **bless, que gera uma referência a um objeto** na memória.

Exercícios

1. Estenda a classe Person para que ela tenha as propriedades RG e tipo sanguíneo.

2. Crie uma classe pai que represente veículos automotores de forma genérica. Depois, crie classes filhas que representem carros de passeio, caminhões e ônibus.

3. Crie uma classe Ponto que tenha como propriedades suas coordenadas x e y. Depois crie uma classe filha, Ponto3D, que tenha também a propriedade z.

4. Crie a classe reta, que deve possuir como propriedades objetos Ponto, definidos no exercício 3.

5. Com a classe WonderWorldPerson crie uma classe CoelhoApressado que tenha os métodos ver_horas e correr e a propriedade atrasado, que armazena 0 para falso e 1 para verdadeiro.

Capítulo 14. Introdução ao Trabalho com Imagens no Perl

Existem diversas formas de trabalhar com imagens, afinal, estamos no Perl. Normalmente as linguagens de "script" não são consideradas capazes de trabalhar com imagens ou bancos de dados. As extensões e scripts do GIMP são feitos com Perl por padrão, o que por si só já comprovaria sua eficácia nesta área. Você tem duas escolhas para trabalhar com imagens no Perl.

1. Escrevendo arquivos binários com as funções pack e unpack. É interessante, apesar de custoso e demorado. Temos um exemplo de como obter dados binários de um arquivo de imagem no capítulo 9. Ler, processar e escrever dados binários é uma saída que eu recomendo apenas para quem quer estudar o modo como determinados formatos de imagem funcionam ou como estudo de algoritmos de tratamento de imagem.
2. A segunda forma é pesquisar no CPAN um módulo de processamento de imagens que atenda as suas necessidade. Existem vários módulos, para todos os gostos e necessidades. Porém, em sua maioria, exigem a instalação de alguma biblioteca em C, com a qual os módulos do Perl trocam informações.

Posso citar como módulos comuns do CPAN para trabalhar com imagens:

1. **GD:** O módulo ***GD*** possui diversos outros módulos, inclusive o ***GD::Simple***, ótimo para iniciantes, que iremos utilizar neste capítulo. O GD realiza o processamento das imagens a partir da biblioteca multiplataforma **GDLibrary**, que pode ser obtida em www.gdlibrary.org. Com o módulo CPAN GD você pode criar imagens e gráficos e processar imagens existentes de forma dinâmica e totalmente em tempo de execução. O GD trabalha com os forma-

tos **png, jpeg e gif**, além de outros formatos. Os módulos GD também têm classes especializadas, como o ***GD::Graph3D***.

2. **Image::Magick**: assim como o GD, o Image::Magick permite manipulação de imagens por meio de programação através da biblioteca ImageMagick (http://www.imagemagick.org/script/perl-magick.php).
3. **Imager**: mais uma extensão do Perl para trabalhar com imagens.
4. **GIMP**: módulo Perl de interface com o GIMP.

As aplicações para geração de imagens em tempo de execução são muitas. Você pode gerar imagens e gráficos diferenciados de acordo com entradas informadas pelo usuário ou de acordo com alguma variável monitorada. Por exemplo, o **MRTG** (http://oss.oetiker.ch/mrtg/) utiliza-se do **Perl + CGI + GD + SNMP** para criar gráficos "on the fly" da utilização de hardware dos equipamentos de rede monitorados e disponibilizar estas informações em uma página web.

Agora que você está motivado para apender a trabalhar com imagens, vamos aos primeiros passos: instale no seu sistema as bibliotecas do GD.

No Linux você pode utilizar o comando:

```
$ sudo aptitude install libgd2
```

No Windows, acesse o site http://www.libgd.org/releases/ e baixe o pacote ***gd-latest-win32.zip*** para instalar a biblioteca.

14.1 Desenhando com o Perl

Ao iniciar um trabalho com algo novo devemos buscar a forma mais simples de fazê-lo. Então vamos fazer este trabalho com o objeto ***GD::Simple***. ***O GD::Simple*** é um módulo do CPAN que gera uma interface simplificada ao módulo ***GD***, também do CPAN. Para criarmos um objeto ***GD::Simple*** usamos o construtor.

```
1 #exemplo14.1
2 use 5.12.0;
3 use GD::Simple;
```

```
  4 my $comprimento = 150; #$comprimento e $altura em pi-
xels
  5 my $altura = 120;
  6 my $image = GD::Simple->new($comprimento, $altura);
  7 $image->bgcolor('chocolate');
```

Depois de criado o objeto você precisa definir a cor de fundo da imagem:

```
  8 $image->rectangle(0,0,$comprimento, $altura);
  9 open my $file_png, '>', 'teste.png';
```

Isso cria um retângulo que cobre toda a imagem que será gerada. O retângulo tem a cor de fundo "chocolate". O método **rectangle** recebe como argumentos os pontos (pixels) iniciais e finais dos eixos **x** e **y**. A contagem inicia-se no canto superior esquerdo da imagem. Agora é só colocar em um arquivo:

```
10 binmode $file_png;
11 print $file_png $image->png;
```

A função ***binmode*** informa ao ***filehandle*** que ele irá criar um arquivo binário e não um arquivo de texto. O método **png** da classe ***GD::Simple*** gera a codificação da saída no formato **png**. Esse pequeno programa gera um arquivo png como o da figura:

Agora vamos aprender a usar a pena.

Pena é uma forma de desenho onde imitamos um desenho à mão, definindo tamanho, cor, posição e movimentação da pena (caneta). A pena tem as seguintes propriedades e métodos importantes:

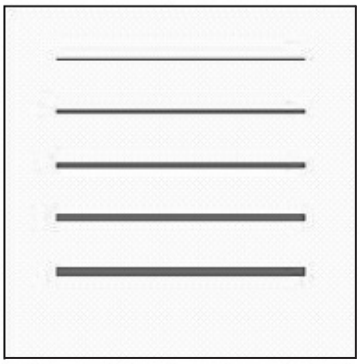
1. ***fgcolor*** e ***bgcolor***: definem a cor do traçado;
2. ***penSize*:** define o tamanho da pena (traço) em pixels;
3. ***curPos*:** posição atual da pena (pixel);
4. ***line*** e ***lineTo***: desenham uma linha;
5. ***angle*** e ***turn***: mudam a direção da pena em graus, no sentido horário.

Vamos ver dois exemplos que usam a pena:

```
  1 #exemplo14.2
  2 use GD::Simple;
  3 my $img = GD::Simple->new(200,200); #cria uma imagem de
200px por 200px
  4
  5 $img->bgcolor("whitesmoke"); #define a cor de fundo.
  6 $img->fgcolor("black");
  7 $img->rectangle(0,0,200,200); #cria um retângulo do ta-
manho da imagem
  8
  9 $img->fgcolor("red"); #Vamos definir a cor da pena para
vermelho
 10
 11 my @pos = (30,30); #e posicionar a pena em x = 30 e y =
30;
 12
 13 for (1..5){
 14  $img->penSize($_); #aqui definimos o tamanho do traço
 15  $img->moveTo(@pos); #aqui definimos a posição da pena.
 16  $img->line(140); #desenhamos uma linha de 140 pixels
 17  $pos[1] += 30 ; #posicionamos a próxima linha 30 pi-
xels abaixo (eixo y)
```

```
 18 }
 19
 20 #E então criamos o arquivo:
 21
 22 open my $file_jpg, ">", "/home/thiago/Documentos/Perl/
GD/file.jpg";
 23 binmode $file_jpg;
 24
 25 print $file_jpg $img->jpeg;
 26 #não use a função say aqui. Ela adiciona um \n ao final
da string.
```

E eis que a nossa saída será:

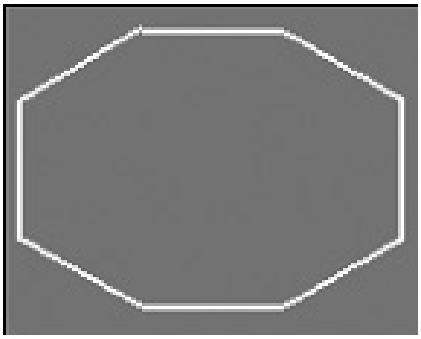


Podemos usar a pena para desenhar também, modificando a direção com ***angle*** e ***turn***:

```
  1 #exemplo14.3
  2 use GD::Simple;
  3 #criamos a imagem
  4 my $img = GD::Simple->new(150,120);
  5
  6 #definimos as cores de fundo e criamos o arquivo
  7 $img->bgcolor("chocolate");
  8 $img->rectangle(0,0,150,120);
  9
```

```
 10 #Definimos a cor da linha
 11 $img->fgcolor("white");
 12
 13 #posicionamos o cursor
 14 #e definimos o tamanho
 15 $img->moveTo(50,10);
 16 $img->penSize(2,2);
 17 $img->line(50); #desenhamos uma linha
 18 $img->angle(30); #mudamos 30 graus a posição em relação
ao eixo x
 19 $img->line(50);
 20 $img->angle(90); #mudamos 90 graus a posição
 21 $img->line(50);
 22 $img->angle(150); #mudamos 150 graus a posição
 23 $img->line(50);
 24 $img->angle(180); #mudamos 180 graus a posição
 25 $img->line(50);
 26 $img->angle(210);
 27 $img->line(50);
 28 $img->angle(270);
 29 $img->line(50);
 30 $img->angle(330);
 31 $img->line(50);
 32
 33 my $path = "/home/thiago/Documentos/GD/turtle.png";
 34 open my $file, ">", $path;
 35 binmode $file;
 36 print $file $img->png;
```

Isto nos dará a seguinte imagem:



14.2 Criando formas básicas

Para estudarmos a criação de formas básicas vamos criar uma bandeira do Brasil por meio de código. A bandeira é composta de diversas formas geométricas. As dimensões da bandeira do Brasil, de acordo com o INMETRO (http://www.inmetro.gov.br/consumidor/produtos/bandeira.asp), são:

"Para cálculos das dimensões, será tomada por base a largura, dividindo-a em 14 (quatorze) partes iguais, sendo que cada uma das partes será considerada uma medida ou módulo (M). Os demais requisitos dimensionais seguem o critério a seguir:

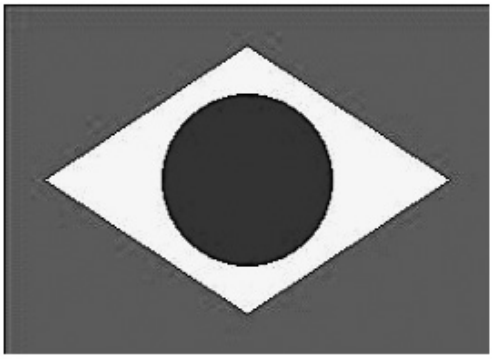
1. Comprimento será de vinte módulos (20M);
2. A distância dos vértices do losango amarelo ao quadro externo será de um módulo e sete décimos (1,7M);
3. O raio do círculo azul no meio do losango amarelo será de três módulos e meio (3,5M);"

Com estas informações podemos criar o programa que irá gerar uma imagem **jpg** com a bandeira do Brasil:

```
  1 #exemplo14.4
  2 use 5.12.0;
  3 use GD::Simple;
  4
  5 my $base = 200;
  6 my $m = int($base/14); #fator M da norma
  7 my $largura = 20 * $m; #20M de acordo com a norma
  8 my $losang_dist = int($m * 1.7); #distância do losango
com a borda
  9 #O centro da imagem
 10 my ($center_x, $center_y) = ($largura/2, $base/2);
 11 #ponto 1 do losango (mais à esquerda)
 12 my $losang_x1 = $losang_dist;
 13 my $losang_y1 = $center_y;
 14
```

```
15 #ponto 2 do losango (mais acima)
16 my $losang_x2 = $center_x;
17 my $losang_y2 = $losang_dist;
18 #ponto 3 do losango (mais à direita)
19 my $losang_x3 = $largura - $losang_dist;
20 my $losang_y3 = $center_y;
21
22 #ponto 4 do losango (mais abaixo)
23 my $losang_x4 = $center_x;
24 my $losang_y4 = $base - $losang_dist;
25 my $img = GD::Simple->new($largura, $base);
26
27 #cria o fundo verde
28 $img->bgcolor("green");
29 $img->rectangle(0,0,$largura,$base);
30
31 #Criamos o losango com um objeto Polygon
32 #especificamos cada vértice do polígono
33 #antes de desenhar
34 my $losango = GD::Polygon->new;
35 $losango->addPt($losang_x1,$losang_y1);
36 $losango->addPt($losang_x2, $losang_y2);
37 $losango->addPt($losang_x3, $losang_y3);
38 $losango->addPt($losang_x4, $losang_y4);
39
40 #desenhamos o losango em amarelo
41 $img->bgcolor("yellow");
42 $img->polygon($losango);
43 #para desenhar o círculo vamos até o centro da imagem
44 #e criamos uma elipse com os dois diâmetros iguais.
45 $img->moveTo($center_x, $center_y);
46 $img->bgcolor("blue");
47 $img->ellipse((7 * $m), (7 * $m));
48 #Agora criamos o arquivo jpeg
49 open my $file, ">",
50  "/home/thiago/Documentos/Curso Perl/GD/bandeira.jpg";
51 binmode $file;
52 print $file $img->jpeg;
```

O arquivo gerado será como o da imagem a seguir:



14.3 Adicionando texto às imagens

A classe ***GD::Simple*** também permite que trabalhemos com strings nos arquivos de imagem. Para isso temos os métodos ***font***, ***fontsize*** e ***string*.** Com o método ***font*** definimos a fonte (Arial, Times, Verdana, etc.), além das propriedades ***bold*, *italic* e *underline*.** O método ***fontsize*** define o tamanho da fonte e ***string*** o texto que será desenhado.

```
  1 #exemplo14.5
  2 use GD::Simple;
  3
  4 my $img = GD::Simple->new(300,150);
  5 $img->fgcolor('red'); #A cor da linha de contorno.
  6 $img->bgcolor('lightsteelblue'); #A cor de fundo.
  7 $img->penSize(3); #Espessura da pena que vai desenhar a
borda.
  8 $img->rectangle(0,0,300,110); #Desenha o retângulo de
fundo.
  9 $img->moveTo(10, 105);
 10 $img->fgcolor('black'); #Define a cor da fonte.
 11 $img->font('Times:italic:bold'); #Define a fonte.
 12 $img->fontsize(20); #Define o tamanho da fonte.
 13 #$img->angle(-25); #Descomente e veja o que acontece.
 14
```

```
15 #E, finalmente, texto a ser desenhado
16 $img->string("TOP SECRET!!!\nPerl Rocks Again!!!");
17
18 open my $file_jpg, ">",
19  "/home/thiago/Documentos/Livro Perl/GD/texto.png";
20 binmode $file_jpg;
21 print $file_jpg $img->png;
```

O arquivo de imagem:

14.4 Estendendo o módulo GD::Simple

O módulo ***GD::Simple*** é orientado a objetos, portanto podemos estender suas funcionalidades por meio de herança (ver capítulo 13). Assim, se eu criar um módulo MyGD que utilize ***GD::Simple*** como classe pai, poderei criar novas funções de desenho e manter as atuais. A classe ***GD::Simple*** não define um método para desenhar circunferências. Portanto, eu pretendo criar uma classe nova que o faça:

```
  1 #exemplo14.6
  2 package MyGD;
  3 use 5.12.0;
  4 use base("GD::Simple");
  5 use Carp;
  6
  7 sub circle{
  8  my $self = shift;
  9  my %args = @_;
 10  confess("Argumento raio obrigatorio") if not defined
$args{raio};
```

```
 11  $self->bgcolor($args{bgcolor}) if defined
$args{bgcolor};
 12  $self->fgcolor($args{fgcolor}) if defined
$args{fgcolor};
 13  $self->ellipse($args{raio}, $args{raio});
 14 }
 15 1
```

Podemos agora reescrever todos os programas anteriores substituindo a classe GD::Simple por MyGD. Assim teremos os mesmos métodos, mais este método que desenha circunferências. Um pequeno teste é refazer o programa bandeiras usando essa classe nova.

14.5 O que ler agora?

Se você pretende desenvolver habilidades de trabalhar com imagens de forma complexa, você pode ler:

1. A documentação dos módulos encontrados no cpan GD, Chart e Image::Magick;
2. Graphics Programming with Perl, de Martin Verbruggen, editora Manning;
3. Mastering Algorithms With Perl, de Jon Orwant, Jarkko Hietaniemi e John Macdonald, editora O'Reilly, capítulo 10.

Exercícios

1. Estenda o módulo GD::Simple de forma que o usuário do módulo possa criar figuras geométricas básicas informando um método que será o nome da figura e bgcolor e fgcolor como argumentos.

2. Crie um programa simples que gere as bandeiras do Brasil, da Irlanda e da Alemanha conforme a dimensão de base informada pelo usuário. Pesquise na internet as proporções oficiais.

3. Crie um programa que gere um banner com o texto e tamanhos informados pelo usuário.

4. Estude a documentação dos módulos ***GD***, ***GD::Chart*** e ***GD::Chart3D.***

5. Pesquise no CPAN como trabalhar com ***Image::Magick***.

Capítulo 15. Bancos de Dados Relacionais

Nos primórdios da informática, antes do DVD ser inventado, antes mesmo do CD, nossa geração antecessora de desenvolvedores de softwares tinham diversos desafios. Mas o desafio que nos vem ao caso neste capítulo é o do **armazenamento de informações para uso póstumo**, ou mesmo uma informação que deve ser **utilizada por diversos programas de forma simultânea**. No capítulo 9 vimos a função ***flock***. Travar o acesso à informação de arquivos elimina diversos problemas, mas gera uma espera por parte dos outros processos que necessitam acessar o mesmo arquivo, mesmo que para modificar registros diferentes. Nos dias modernos, essa latência pode ser fatal. Vamos imaginar sistemas de **e-commerce, bancos online, reservas de hotel, reservas de passagens... todos esses sistemas devem suportar diversos acessos simultâneos.** Imagine se você não pode fazer uma compra via internet enquanto outro usuário estiver comprando, mesmo que sejam produtos distintos, como se estivéssemos em uma fila comum de caixa de supermercado? O ***flock*** pode ser pensado assim, como uma fila de caixa, onde apenas um cliente é atendido por vez!

Para solucionar esse problema, nossos sábios antecessores desenvolveram uma tecnologia muito boa, o **Banco de Dados Relacional**. Bancos de dados relacionais têm as seguintes vantagens:

1. **Têm um engenho de administração** que toma conta do banco para você não se preocupar em travar e destravar o acesso aos dados. O engenho do banco tem formas mais sofisticadas de garantir acesso simultâneo à mesma informação.
2. **Armazenagem segura da informação.** Conhece os três pilares da segurança da informação? Eles são confidencialidade, disponibilidade e confiabilidade:
   - **Confidencialidade:** os bancos de dados modernos, como Oracle e MySQL, permitem a criação de usuários de acesso aos bancos, como acesso de

leitura, escrita e "leitura e escrita", garantindo a confidencialidade da informação armazenada;

- **Disponibilidade:** por ter um gerenciamento seguro e rápido de acesso e controle de mudanças, as informações sempre estão disponíveis;
- **Confiabilidade ou veracidade:** com seu engenho controlando **acesso somente de pessoas autorizadas** e controlando as mudanças da informação, o banco de dados **evita** condições de competição ***(race conditions),*** impedindo que a informação seja corrompida e garantindo a confiabilidade da fonte.

3. **Estrutura:** Os engenhos de RDB **evitam que dados sejam adicionados de forma incompleta ou de forma que não atendam a parâmetros**, ou seja, se você tiver uma tabela com as colunas: Nome, Endereço e Telefone e seus valores forem considerados obrigatórios, o próprio engenho do seu banco irá evitar que sejam enviados dados que não contenham todas estas informações, poupando um pouco de esforço do programador Perl, preguiçoso por natureza.

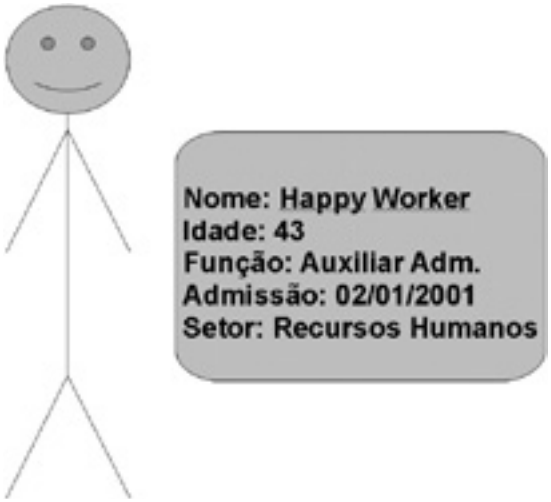
Como toda tecnologia, para trabalharmos com banco de dados temos que aprender alguns conceitos e termos que fazem parte do jargão dos administradores de banco de dados, que são: entidade, tabela, atributo, registro, chave primária, chave estrangeira e relacionamento.

Entidade: Elemento sobre o qual queremos armazenar informações. Normalmente as tabelas de um banco são representações de elementos do mundo real. Vou utilizar agora o exemplo mais clichê que existe: um banco de dados de uma organização ou empresa tem, com certeza, informações sobre empregados. Então o **empregado é uma entidade a ser modelada**. Esta entidade tem propriedades como nome, endereço, setor no qual ele trabalha, salário, cargo etc.

NOTA: Você pode perceber que as entidades de banco de dados são modeladas de forma semelhante às propriedades dos objetos na programação OOP.

Um banco de dados, normalmente, conterá informações de mais de uma entidade.

Relação ou Tabela: uma relação ou tabela é **o elemento básico de um banco de dados relacional**. A tabela é a representação de um agrupamento de entidades



no banco. Uma tabela, ou relação, de um banco SQL é como uma tabela de uma planilha, contendo linhas, colunas e células.

Nome	Idade	Função	Setor	Endereço	Salário
Happy Worker	43	Auxiliar Administrativo	R.H.	Rua dos Bobos, 0	R$ 1,500.00
Dono da Empresa	55	Chefe	Presidência	Bela Vista	R$ 100,000.00
The IT Guy	22	Analista	TIC	Av. Basil, 300	R$ 3,000.00

Colunas ou atributos: as colunas são itens de informação da entidade que modelamos. Ou seja, são as **propriedades interessantes de um elemento do mundo real**. No caso do Empregado, temos nome, idade, função, setor, endereço e salário. Se fôssemos modelar um banco de dados de informações médicas, por exemplo, o mesmo sr. Happy Worker seria visto com outra perspectiva e suas propriedades ou atributos na tabela seriam outros. Seriam, por exemplo, altura, peso, idade, tipo sanguíneo, cirurgias etc. Cada propriedade importante para o sistema é um atributo ou uma coluna na tabela.

Linhas ou registros: no caso dos empregados, temos para cada linha ou registro um indivíduo ou empregado do qual queremos guardar as informações. Então as linhas são os elementos que formam um conjunto do universo analisado. Cada

elemento individual do nosso conjunto (empregados da minha empresa) tem um registro ou linha na tabela.

Chaves: as **superchaves** são um grupo de colunas que, com seus dados, podem identificar uma linha única na tabela. Por exemplo, os atributos nome e endereço podem ser um identificador que remete a um elemento único da tabela. Uma **superchave mínima** é uma superchave otimizada. Pense: em vez de usarmos duas colunas ou três, podemos adicionar na tabela um número de identificação do funcionário, como sua matrícula na empresa ou o seu CPF. Isso porque é de certa forma comum termos mais de uma pessoa com o mesmo nome na mesma empresa. Embora seja raro que duas pessoas com o mesmo nome morem na mesma casa, não existe nada que impeça que isso ocorra. Porém, o número de matrícula ou do documento CPF são garantias de individualidade. Uma superchave escolhida para garantir a individualidade do elemento é chamada de **chave primária**.

Chaves estrangeiras: nossos empregados trabalham em setores. Se pensarmos com cuidado, o setor pode ser considerado uma entidade, de forma que tenhamos uma tabela de setores.

Setor_ID	Setor	Descrição
Pre	Presidência	Tomada de decisões globais
TIC	TIC	Suporte e desenvolvimento de TIC
RH	Recursos Humanos	Desenvolvimento e controle de pessoal
Fin	Financeiro	Controle de gastos e fluxo de caixa

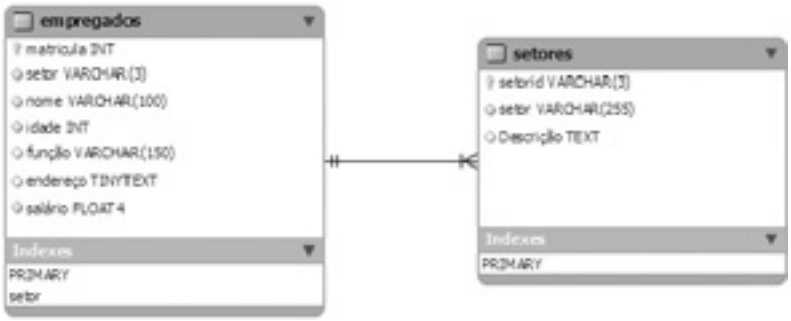
Então podemos modificar a tabela dos empregados gerando uma relação entre Empregados e Setores:

Matrícula	Setor	Nome	Idade	Função	Endereço	Salário
3	RH	Happy Worker	43	Auxiliar Administrativo	Rua dos Bobos, 0	R$ 1.500,00
1	Pre	Dono da Empresa	55	Chefe	Bela Vista	R$ 100.000,00
6	TIC	Th e IT Guy	22	Analista	Av. Brasil, 300	R$ 3.000,00

Adicionamos uma propriedade, a matrícula, que será o identificador único do elemento, ou seja, **sua chave primária**. A tabela Setores tem como chave primária o ID do Setor, que é um código de letras interno à empresa. Assim, podemos criar uma chave estrangeira na tabela empregados.

Relacionamento: um relacionamento nada mais é que uma ligação entre duas entidades. No nosso caso **existe um relacionamento entre a tabela Empregado e a tabela Setor, pois o empregado é lotado em um setor.** As relações podem ser **um para um, um para muitos ou muitos para muitos**. Em nosso exemplo estamos usando uma relação um para um, pois cada empregado é lotado em um, e apenas um, setor, e cada setor tem apenas um empregado. No mundo real as empresas têm um ou mais funcionários por setor. Assim, **a relação real é um (setor) para muitos (empregados).**

Mais adiante faremos um projeto de design de banco para que os conceitos fiquem mais claros.



15.1 Esquema

O esquema é a forma ou estrutura do banco sem que haja, necessariamente, registros armazenados nas tabelas. No caso, a imagem anterior é um exemplo de esquema, pois ela informa quais são os relacionamentos entre tabelas, as tabelas, as colunas das tabelas e o tipo de dados. Mas como o banco ainda não foi implementado, não existem registros nas tabelas.

15.2 Tipos de dados

Os atributos são dados de uma entidade que juntos formam uma informação relevante. Estes dados possuem tipos. Diferentemente do Perl, os dados têm diversos tipos possíveis. Os principais (não todos) são:

Tipo	O que armazena	Descrição
INT	Numérico	Armazena um número inteiro de 4 bytes
SMALLINT	Numérico	Armazena um número inteiro de 2 bytes
Real	Numérico	Número de ponto flutuante com 6 dígitos decimais de precisão

Tipo	O que armazena	Descrição
CHAR (n)	String	Armazena uma sequência de **n** caracteres
VARCHAR (n)	String	Armazena uma sequência de até **n** caracteres
BLOB	Arquivo	Armazena um arquivo (normalmente imagem, som ou vídeo)
DATE	Data	Armazena datas no formato aaaa-mm-dd
TIME	Hora	Armazena hora no formato hh:mm:ss
TIMESTAMP	Data e Hora	Armazena data e hora no formato aaaa-mm-dd hh:mm:ss

Dependendo do fabricante, os tipos de dados podem ser um pouco diferentes. Embora estes tipos informados sejam encontrados em qualquer banco, alguns fabricantes adicionam outros tipos. Mas sempre haverá os tipos **INT, VARCHAR (n) e DATE**. Veja uma pequena comparação dos tipos de data entre fabricantes diferentes:

- **O padrão SQL:2003** define para datas: `DATE, TIME, TIMESTAMP, TIME WITH TIME ZONE, TIMESTAMP WITH TIME ZONE` e `INTERVAL;`
- **O Orcle 11g** implementa: `DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE,` e `TIMESTAMP WITH LOCAL TIME ZONE, SMALLDATE` e `DATETIME;`
- **O DB2 9** implementa: `DATE, TIME` e `TIMESTAMP.`

15.3 Normalização e design do banco

Quando iniciamos o design de um banco, devemos definir qual o universo em estudo. Então vamos criar um banco de dados o mais profissional e simples possível para atender ao RH de uma empresa qualquer. Nosso universo de estudo compreende os funcionários e a estrutura organizacional da empresa. Logo em seguida devemos pensar em quais **queries** vamos efetuar no banco na maior parte do tempo. Ou seja, depois do banco consolidado, a maioria das **queries** são de consulta. Por exemplo, na época do pagamento são feitas as **queries** para gerar a folha. É muito raro uma pessoa mudar de nome, então não haverá muitos **updates** nesse campo. Mas funcionários entram, saem, se aposentam. Porém, com uma frequência inferior à necessidade de consultas salariais para efetuar pagamentos quinzenais ou mesmo à busca de pessoas com certas habilidades para alocar a mão de obra em um determinado projeto.

15.3.1 Quais são as informações relevantes?

De quais informações o RH precisa? Nome, data de nascimento, sexo, número dos documentos (RG, CPF, Título de Eleitor, Reservista), data de admissão, formação, idiomas, habilidades, setor, matrícula etc. Não vamos usar todas estas informações porque este banco tem apenas fins didáticos:

Empregado: Matrícula, nome, nascimento, sexo, CPF, Admissão, Salário, Cargo, Habilidades, Setor, Idiomas.

Claro que qualquer empresa tem uma lista de setores ou gerências e uma definição de habilidades importantes a considerar. Não é interessante para nossa empresa habilidades do tipo aeromodelismo, mas programação Java ou Estatística pode ser útil. Então precisamos que o RH nos forneça esta lista. Vamos supor a lista a seguir:

Habilidades importantes: Liderança, Matemática financeira, Java, C, Perl, Estatística, Contabilidade, Direito Trabalhista.

Outras listas interessantes seriam idiomas e formação. Poucas empresas estão interessadas em pessoas que falem esperanto ou tupi-guarani. E nossa empresa não precisa de pessoas formadas em Gastronomia nem tampouco em Teologia. Mas nossa empresa se interessa por Inglês, Espanhol e Francês. Então nossa tabela de usuários deveria ter as colunas:

Bem, este nosso banco mais parece uma tabela de planilha, o que por si só não seria problema se não fossem algumas situações que podemos perceber após povoar um pouco o banco:

Matrícula	Nome	Nascimento	Sexo	CPF	Admissão	Cargo	Salário	Setor	Habilidades	Idiomas
2	Happy Worker	06/06/1973	M	675201645	08/10/2001	Analista de Sistemas	3500,00	Informação	C, Java, Perl	inglês
1	The Big Boss	06/03/1975	M	2147483647	11/09/2001	Presidente	57827,30	Presidência	Direito	Inglês, francês
3	Sad Worker	07/03/1983	M	265741256	23/09/2004	Analista Financeiro	2850,36	Informação	Contabilidade e Finanças	espanhol

A primeira observação importante a fazer diz respeito à **redundância de informação**. Temos dois funcionários no Departamento de Tecnologia da Informação e as habilidades também estão repetidas em diversas linhas. Isso significa desperdício de armazenamento e lentidão nas queries. Outro defeito são as anomalias que temos no banco. Se o sr. Happy Worker e o sr. Sad Worker se demitirem e as linhas da tabela que contêm seus registros forem excluídas, bem, nosso banco não terá mais informações sobre a existência do Setor de Tecnologia da Informação, pois não existe nenhum outro registro no banco que registre a existência deste setor. Ou ainda: e se alguém inserir um novo funcionário e achar que é mais fácil digitar a sigla TI em vez de Tecnologia da Informação na coluna Setor? Vamos ter uma inconsistência. **Isto é o que chamamos de anomalias.**

As anomalias podem ser de ***inserção, update e delete***. As anomalias de inserção permitem que dados inconsistentes sejam inseridos no banco devido ao design incorreto. O mesmo vale para as anomalias de update. As anomalias de delete fazem com que percamos mais informação do que as que queremos descartar do banco.

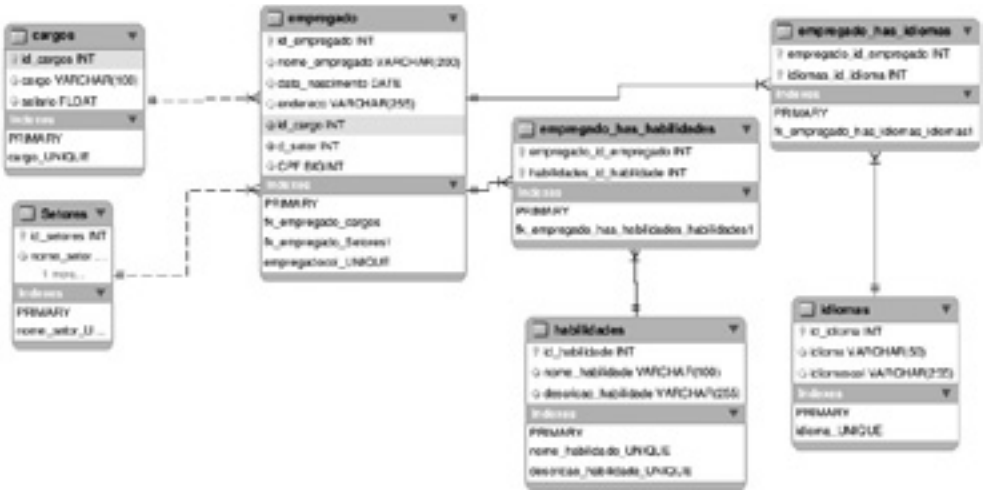
Outra regra importante diz respeito aos **Null Values**. Devemos evitar incluir propriedades que gerem grande quantidade de **Null Values**, ou seja, **células sem valor**.

Em um banco de dados pequeno como este, estes problemas podem parecer pontuais e possíveis de ser controlados manualmente, mas crescem exponencialmente conforme o volume de dados do banco aumenta. E seria trágico uma empresa, digamos, do tamanho da HP, IBM, Itaú, enfrentando problemas como estes! Alguns bytes de strings repetidas a mais em um banco de uma pequena empresa não significam nada, e com certeza seu controle de RH poderia ser feito em uma planilha e não com um banco relacional.

Mas, digamos, uma empresa com 50 ou 60 mil funcionários. Se o banco usar strings padrão Unicode que consomem 2 bytes por caractere e cada nome de setor tiver por volta de 10 caracteres, teremos 20 bytes de desperdício por repetição da string. Agora, suponha que esta empresa tenha vinte setores e o nome do setor é gravado em cada linha de registro dos empregados – são 60.000 x 20 = 1.200.000 bytes apenas para registrar vinte setores, o que poderia ser feito com apenas duzentos bytes para os nomes, mais vinte bytes para a chave primária da tabela de setores e mais sessenta mil bytes para as chaves estrangeiras na tabela de empregados. Daí teríamos 60000 + 220 = 60.220 bytes. Quase vinte vezes menor que a primeira situação. Então a primeira regra do design é: se uma informação se repete por uma tabela, ela provavelmente deve ser uma entidade e precisa de uma tabela própria. No caso dos empregados, uma tabela de setores e uma de cargos faz muito sentido.

Nossa próxima regra é evitar mais de um dado por célula. A tabela empregados tem diversas informações sobre os idiomas e habilidades dos empregados. Porém, as regras de normalização definem que cada célula deve conter informações atômicas. Portanto, precisamos de uma tabela de habilidades, uma de idiomas e de tabelas intermediárias, contendo os relacionamentos idioma x empregado e habilidades x empregado.

Então, vamos criar o banco relativamente simples do diagrama adiante:



Trabalhar com um servidor MySQL. Primeiro porque ele é o sistema de administração de banco de dados mais utilizado no mundo - não que ele seja melhor que os outros. Segundo por ele ser open-source, gratuito e altamente profissional. O script para a criação das tabelas no banco segue adiante. Não se esqueça de criar o banco primeiro: **create database mydb**. Não se assuste se não souber a linguagem SQL. Este é o nosso próximo assunto.

15.3.2 Sript SQL

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_
KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';

DROP SCHEMA IF EXISTS `mydb` ;
```

```
CREATE SCHEMA IF NOT EXISTS `mydb` DEFAULT CHARACTER SET
latin1 COLLATE latin1_swedish_ci ;
SHOW WARNINGS;
USE `mydb` ;

-- -----------------------------------------------------
-- Table `mydb`.`cargos`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`cargos` ;

SHOW WARNINGS;
CREATE TABLE IF NOT EXISTS `mydb`.`cargos` (
`id_cargos` INT NOT NULL AUTO_INCREMENT ,
`cargo` VARCHAR(100) NOT NULL ,
`salario` FLOAT NOT NULL ,
PRIMARY KEY (`id_cargos`) )
ENGINE = InnoDB;

SHOW WARNINGS;
CREATE UNIQUE INDEX `cargo_UNIQUE` ON `mydb`.`cargos`
(`cargo` ASC) ;

SHOW WARNINGS;

-- -----------------------------------------------------
-- Table `mydb`.`Setores`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`Setores` ;

SHOW WARNINGS;
CREATE TABLE IF NOT EXISTS `mydb`.`Setores` (
`id_setores` INT NOT NULL AUTO_INCREMENT ,
`nome_setor` VARCHAR(50) NOT NULL ,
`descricao` VARCHAR(255) NULL ,
PRIMARY KEY (`id_setores`) )
ENGINE = InnoDB;

SHOW WARNINGS;
```

```
CREATE UNIQUE INDEX `nome_setor_UNIQUE` ON `mydb`.`Setores`
(`nome_setor` ASC) ;

SHOW WARNINGS;

-- -----------------------------------------------------
-- Table `mydb`.`empregado`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`empregado` ;

SHOW WARNINGS;
CREATE TABLE IF NOT EXISTS `mydb`.`empregado` (
`id_empregado` INT NULL AUTO_INCREMENT ,
`nome_empregado` VARCHAR(200) NOT NULL ,
`data_nascimento` DATE NOT NULL ,
`endereco` VARCHAR(255) NULL ,
`id_cargo` INT NOT NULL ,
`d_setor` INT NOT NULL ,
`CPF` BIGINT NOT NULL ,
PRIMARY KEY (`id_empregado`) )
ENGINE = InnoDB;

SHOW WARNINGS;
CREATE UNIQUE INDEX `empregadocol_UNIQUE` ON
`mydb`.`empregado` (`CPF` ASC) ;

SHOW WARNINGS;

-- -----------------------------------------------------
-- Table `mydb`.`habilidades`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`habilidades` ;

SHOW WARNINGS;
CREATE TABLE IF NOT EXISTS `mydb`.`habilidades` (
`id_habilidade` INT NOT NULL AUTO_INCREMENT ,
`nome_habilidade` VARCHAR(100) NOT NULL ,
`descricao_habilidade` VARCHAR(255) NOT NULL ,
```

```
PRIMARY KEY (`id_habilidade`) )
ENGINE = InnoDB;

SHOW WARNINGS;
CREATE UNIQUE INDEX `nome_habilidade_UNIQUE` ON
`mydb`.`habilidades` (`nome_habilidade` ASC) ;

SHOW WARNINGS;
CREATE UNIQUE INDEX `descricao_habilidade_UNIQUE` ON
`mydb`.`habilidades` (`descricao_habilidade` ASC) ;

SHOW WARNINGS;

-- -----------------------------------------------------
-- Table `mydb`.`idiomas`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`idiomas` ;

SHOW WARNINGS;
CREATE TABLE IF NOT EXISTS `mydb`.`idiomas` (
`id_idioma` INT NOT NULL AUTO_INCREMENT ,
`idioma` VARCHAR(50) NOT NULL ,
`idiomascol` VARCHAR(255) NULL ,
PRIMARY KEY (`id_idioma`) )
ENGINE = InnoDB;

SHOW WARNINGS;
CREATE UNIQUE INDEX `idioma_UNIQUE` ON `mydb`.`idiomas`
(`idioma` ASC) ;

SHOW WARNINGS;

-- -----------------------------------------------------
-- Table `mydb`.`empregado_has_habilidades`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`empregado_has_habilidades` ;

SHOW WARNINGS;
```

```
CREATE TABLE IF NOT EXISTS `mydb`.`empregado_has_habili-
dades` (
`empregado_id_empregado` INT NOT NULL ,
`habilidades_id_habilidade` INT NOT NULL ,
PRIMARY KEY (`empregado_id_empregado`, `habilidades_id_ha-
bilidade`) )
ENGINE = InnoDB;

SHOW WARNINGS;

-- -----------------------------------------------------
-- Table `mydb`.`empregado_has_idiomas`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `mydb`.`empregado_has_idiomas` ;

SHOW WARNINGS;
CREATE TABLE IF NOT EXISTS `mydb`.`empregado_has_idiomas` (
`empregado_id_empregado` INT NOT NULL ,
`idiomas_id_idioma` INT NOT NULL ,
PRIMARY KEY (`empregado_id_empregado`, `idiomas_id_idioma`)
)
ENGINE = InnoDB;

SHOW WARNINGS;

SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;

-- ------------------
-- Populando o banco:
-- ------------------

INSERT INTO `mydb`.`Setores` (`nome_setor`, `descricao`)
VALUES ('Presidência', 'Gabinete do presidente');

INSERT INTO `mydb`.`Setores` (`nome_setor`, `descricao`)
VALUES ('Tecnologia da Informação', 'Setor responsável pela
tecnologia e desenvolvimento de produtos');
```

```
INSERT INTO `mydb`.`Setores` (`nome_setor`, `descricao`)
VALUES ('Recursos Humanos', 'Administração de Funcioná-
rios');

INSERT INTO `mydb`.`Setores` (`nome_setor`, `descricao`)
VALUES ('Serviços Gerais', 'Limpeza, motoristas e almoxari-
fado');

INSERT INTO `mydb`.`cargos` (`cargo`, `salario`) VALUES
('Presidente', '55537.4');

INSERT INTO `mydb`.`cargos` (`cargo`, `salario`) VALUES
('Analista de Suporte', '3000.00');

INSERT INTO `mydb`.`cargos` (`cargo`, `salario`) VALUES
('Analista Programador', '4500.00');

INSERT INTO `mydb`.`cargos` (`cargo`, `salario`) VALUES
('Contador', '3600.00');

INSERT INTO `mydb`.`cargos` (`cargo`, `salario`) VALUES
('Auxiliar de Escritório', '1500.00');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Java', 'Habilidades em progra-
mação Java');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('C', 'Habilidades em programa-
ção C');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Perl', 'Conhecimentos de de-
senvolvimento em Perl');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Linux', 'Conhecimentos em sis-
tema operacional Linux');
```

```
INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Gerência de Pessoal', 'Habili-
dade em liderar pessoas');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Matemática Financeira', 'Co-
nhecimentos em Matemática financeira');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Direito', 'Para advogados');

INSERT INTO `mydb`.`habilidades` (`nome_habilidade`, `des-
cricao_habilidade`) VALUES ('Contabilidade', 'Habilidades e
conhecimentos em contabilidade empresarial');

INSERT INTO `mydb`.`idiomas` (`idioma`, `idiomascol`) VA-
LUES ('Inglês', 'Fluência na língua inglesa');

INSERT INTO `mydb`.`idiomas` (`idioma`, `idiomascol`) VA-
LUES ('Espanhol', 'Fluência em espanhol');

INSERT INTO `mydb`.`idiomas` (`idioma`, `idiomascol`) VA-
LUES ('Frances', 'Fluência em francês');

INSERT INTO `mydb`.`empregado` (`nome_empregado`, `data_
nascimento`, `endereco`, `id_cargo`, `id_setor`, `CPF`)
VALUES ('The Big Boss', '1973-11-01', 'Rua dos bobos, 0',
'1', '1', '3300998877');
INSERT INTO `mydb`.`empregado` (`nome_empregado`, `data_
nascimento`, `endereco`, `id_cargo`, `id_setor`, `CPF`) VA-
LUES ('Happy Worker', '1979-03-11', 'Rua das Camélias, 45',
'2', '2', '76542308740');

INSERT INTO `mydb`.`empregado` (`nome_empregado`, `data_
nascimento`, `endereco`, `id_cargo`, `id_setor`, `CPF`) VA-
LUES ('Thiago Glauco', '1980-02-13', 'Av. Gen. Marcos Sil-
va, 3578', '3', '2', '678954639279');
```

15.4 SQL e o Perl

O SQL é uma linguagem definida por um padrão internacional que deve ser observado pelos fabricantes de banco de dados. O padrão mais recente é o ***SQL:2008***. O SQL é uma linguagem de programação completa, e os seus detalhes estão fora do escopo deste livro. Mas iremos, juntamente com o andamento da programação em Perl, verificar os comandos essenciais do SQL.

Para trabalhar com o SQL e o Perl, primeiramente devemos instalar o módulo DBI.

```
$ cpan -i DBI
```

O DBI foi desenvolvido em 1997 num projeto da NASA. Ele gera uma interface rápida e amigável para programação com o banco por meio de drivers. Após instalado, devemos instalar o driver do ***RDBMS*** que queremos utilizar. Para verificar se o driver do seu banco está instalado, utilize este pequeno comando:

```
# perl -MDBI -E '
say "Meus drivers são: ", join " ", DBI->available_drivers;
'
Meus drivers são: DBM ExampleP File Gofer Proxy Sponge mys-
ql
```

Caso você precise utilizar um outro banco, basta procurar no CPAN o driver para seu banco com a String de pesquisa ***DBD::NomeDoRDBMS*.** Por exemplo, para o Oracle instale o ***DBD::Oracle***. Para o MS-SQL você pode utilizar o ***DBD::ADO*.**

Para se conectar ao banco usamos a seguinte estrutura:

```
  1 #exemplo15.1
  2 use 5.12.0;
  3 use DBI;
  4
  5 #A dsn é uma variável que contém a string para se co-
nectar ao banco.
  6 #No caso do Perl, ela tem a seguinte estrutura:
  7 # DBI:driver_name:database=banco;host=nome__ou_ip_do_
host;port=porta_tcp
```

```
  8 #Se host for omitido será utilizado o computador local
como host e se a
  9 # porta for omitida será utilizada a porta padrão do
banco.
 10
 11 my $dsn = "DBI:mysql:database=mydb;host=localhost";
 12
 13 #Após definida a dsn, podemos efetuar a conexão no ban-
co.
 14 #O método DBI->errstr retorna alguma mensagem de erro
do banco.
 15 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass")
 16 or die("Não foi possível conectar ao banco:", DBI-
->errstr);
 17
 18 say $dbh;
```

Para testar a mensagem de erro, tente modificar o usuário da conexão por um usuário que não existe no banco. Após uma conexão realizada com sucesso, você pode realizar as queries via programação Perl. Para criar uma query, basta criar uma string qualquer que contenha código ***SQL:2008***. Adicione o código a seguir ao exemplo15.1:

```
 19
 20 my $select = "SELECT * FROM cargos ORDER BY cargo";
```

A string ***$select*** é uma string qualquer que pode ser criada dinamicamente ou colocar um valor léxico diretamente no método ***prepare***. Para ver o método em ação, adicione as linhas a seguir ao exemplo anterior:

```
 21 my $sth = $dbh->prepare($select);
 22 $sth->execute;
```

O método ***prepare*** tem um funcionamento especial, pois ele deixa a query **pré--armazenada na memória do banco**. Guarde este método na manga, pois ele previne ataques de **SQL-Injection** em suas aplicações web e otimiza aplicações

que repetem queries semelhantes. O comando execute efetua a execução da query armazenada. Então podemos utilizar os dados retornados. Para isso temos diversos métodos, como ***fetch***, ***fetchrow_hashref*** ou ***fetchrow_arrayref***. Vamos ver cada um deles em detalhes, mas o método ***fetchrow_hashref*** retorna cada linha do banco afetada pela query no formato:

```
$fetchrow_hashref->{ column_name }
```

Coloque o seguinte código ao final do exemplo15.1:

```
 23
 24 while(my $data = $sth->fetchrow_hashref) {
 25  printf "Um %s ganha R\$ %.2f\n",$data->{cargo}, $data-
->{salario};
 26 }
```

Se juntarmos o código visto neste capítulo em um programa teríamos a saída:

```
DBI::db=HASH(0x8953ab0)
Um Analista de Suporte ganha R$ 3000.00
Um Analista Programador ganha R$ 4500.00
Um Auxiliar de Escritório ganha R$ 1500.00
Um Contador ganha R$ 3600.00
Um Presidente ganha R$ 55537.40
```

Esta saída corresponde à tabela alimentada com:

```
+-----------+-----------------------+---------+
| id_cargos | cargo | salario |
+-----------+-----------------------+---------+
| 2 | Analista de Suporte | 3000 |
| 3 | Analista Programador | 4500 |
| 5 | Auxiliar de Escritório | 1500 |
| 4 | Contador | 3600 |
| 1 | Presidente | 55537.4 |
```

NOTA: você terá estes valores na tabela do MySQL se você utilizou o script descrito no subcapítulo 15.3.2.

Para ver a diferença entre o método ***fetchrow_hashref*** e o método ***fetch*** vamos modificar o loop final do exemplo15.1:

```
24 while(my $data = $sth->fetch) {
25  say $data->[0],"\t", $data->[1], "\t", $data->[2];
26 }
```

Assim você terá a saída:

```
DBI::db=HASH(0x83c7a60)
2 Analista de Suporte 3000
3 Analista Programador 4500
5 Auxiliar de Escritório 1500
4 Contador 3600
1 Presidente 55537.4
```

O ***fetch*** retorna um array onde cada item é uma célula da linha. Já na forma ***fetchrow_hashref*** temos uma forma mais legível, afinal é possível ler o nome da coluna e não apenas um índice numérico.

15.5 Lidando com os erros de forma segura

O maior medo dos desenvolvedores que utilizam algum servidor SQL são os ataques de **SQL injection**. E as mensagens de erro exibidas em aplicações web mal escritas são uma grande fonte de informação quanto à estrutura das tabelas e aos tipos de informações que elas armazenam. Para trabalharmos com os erros de SQL, o módulo DBI oferece as propriedades de controle: **PrintError** e **RaiseError**.

PrintError, quando ativado, faz com que os erros sejam enviados para STDOUT via ***warn***. ***RaiseError*** faz com que o programa execute a função ***die*** e envie a mensagem de erro para STDOUT. Estas mensagens de erro são legais enquanto estamos desenvolvendo, mas não devem ser vistas pelos usuários. Um exemplo de mensagem de erro que permite esta visão seria:

```
DBD::mysql::st execute failed: You have an error in your
SQL syntax; check the manual that corresponds to your MyS-
QL server version for the right syntax to use near 'cargos
WHERE cargo='humm' ORDER BY cargo' at line 1 at /home/thia-
go/workspace/TICursos/DBI/erro.pl line 18, <STDIN> line 1.
```

Veja que esta mensagem informa qual o seu **RDBMS**, no caso mysql, dando a primeira informação de que um **hacker** precisa: qual o servidor que você usa. A partir daí ele pode explorar falhas conhecidas. Uma segunda informação valiosa **é parte da query SQL** enviada. Isto ajuda o hacker a **deduzir qual a estrutura da sua tabela**. No Perl DBI com MySQL temos a vantagem de que apenas parte da query é exposta, porém alguns RDBMS enviam a query inteira para a saída padrão. Uma mensagem de erro destas é um presente para qualquer hacker. Uma forma de evitar que um hacker visualize estas mensagens é configurar o **PrintError** para 0 e criar um **arquivo de logs** para as saídas de **STDERR** no seu programa em produção. Veja o exemplo:

```
  1 #exemplo15.2
  2 use 5.12.0;
  3 use DBI;
  4
  5 my $dsn = "DBI:mysql:database=mydb;host=localhost";
  6
  7 #Fechamos STDERR e abrimos novamente, direcionado a um
arquivo de log.
  8 #Todo erro gerado por warn ou die será enviado para
este arquivo.
  9 close(STDERR); open(STDERR, ">>", logfile");
 10
 11 #Definimos os atributos de erros do DBI para não enviar
erros diretamente
 12 # para a tela, nem terminar a execução em caso de erros.
```

```
 13 my %attr = ( PrintError => 0, RaiseError => 0 );
 14 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass",
\%attr)
 15 or err_handle();
 16
 17 my $select = "SELECT * FROM cargos ORDER BY cargo";
 18
 19 my $sth = $dbh->prepare($select);
 20 $sth->execute;
 21
 22 #Após executar os comandos SQL verificamos se ocorreu
algum erro.
 23 if (DBI->err()) {
 24  err_handle()
 25 }
 26 while(my $data = $sth->fetch) {
 27  say $data->[0],"\t", $data->[1], "\t", $data->[2];
 28 }
 29 #A função err_handle envia para nosso arquivo de log
todas as informações
 30 # pertinentes, como PID, hora do erro, número do erro e
mensagem de erro.
 31 sub err_handle{
 32 die(scalar localtime,
 33  "\tPID: ",
 34  $$,
 35  " $0",
 36  " Ocorreu um erro de operação com o banco: ",
 37  DBI->err(), " ", DBI->errstr);
 38 }
```

Veja alguns exemplos de mensagens de erro gerado forçadamente:

```
Tue Sep 28 18:42:25 2010 PID: 5268 /home/thiago/workspace/
TICursos/DBI/Connect2.pl Ocorreu um erro de operação com o
banco: 1146 Table 'mydb.cargo' doesn't exist at /home/thia-
go/workspace/TICursos/DBI/Connect2.pl line 30.
Tue Sep 28 18:43:04 2010 PID: 5320 /home/thiago/workspace/
TICursos/DBI/Connect2.pl Ocorreu um erro de operação com o
```

```
banco: 1045 Access denied for user 'root'@'localhost' (us-
ing password: YES) at /home/thiago/workspace/TICursos/DBI/
Connect2.pl line 30.
Tue Sep 28 18:50:08 2010 PID: 6298 /home/thiago/workspace/
TICursos/DBI/Connect2.pl Ocorreu um erro de operação com o
banco: 1146 Table 'mydb.cargo' doesn't exist at /home/thia-
go/workspace/TICursos/DBI/Connect2.pl line 30.
Tue Sep 28 18:50:39 2010 PID: 6430 /home/thiago/workspace/
TICursos/DBI/Connect2.pl Ocorreu um erro de operação com o
banco: 1146 Table 'mydb.cargo' doesn't exist at /home/thia-
go/workspace/TICursos/DBI/Connect2.pl line 30.
Tue Sep 28 18:51:19 2010 PID: 6597 /home/thiago/workspace/
TICursos/DBI/Connect2.pl Ocorreu um erro de operação com o
banco: 1054 Unknown column 'cargos' in 'order clause' at /
home/thiago/workspace/TICursos/DBI/Connect2.pl line 30.
Fri Oct 1 21:23:58 2010 PID: 4208 /home/thiago/workspace/
TICursos/DBI/erro.pl Ocorreu um erro de operação com o
banco: 1226 User 'mydb_user' has exceeded the 'max_connec-
tions_per_hour' resource (current value: 10) at /home/thia-
go/workspace/TICursos/DBI/erro.pl line 11.
```

Assim você exerce controle sobre o que acontece com sua aplicação e ainda mantém o controle sobre os erros que acontecem para manutenção futura da aplicação. Outra preocupação importante diz respeito ao controle que devemos ter das entradas de usuários. Vamos supor uma aplicação que permita ao usuário verificar o seu salário...

```
  1 #exemplo15.3
  2 use 5.12.0;
  3 use DBI;
  4
  5 my $dsn = "DBI:mysql:database=mydb;host=localhost";
  6 my %attr = ( PrintError => 0, RaiseError => 0 );
  7 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass",
\%attr)
  8 or err_handle();
  9
```

```
 10 print "Verificar salário: ";
 11 chomp (my $nome = <STDIN>);
 12
 13 my $sql = <<SQL;
 14 select empregado.nome_empregado,cargos.salario
 15 from empregado,cargos where empregado.id_cargo=cargos.
id_cargos
 16 and empregado.nome_empregado='$nome'
 17 SQL
 18 my $sth = $dbh->prepare($sql);
 19 $sth->execute;
 20
 21 if (DBI->err()) {
 22  err_handle()
 23 }
 24
 25 while(my $data = $sth->fetch) {
 26  say $data->[0],"\t", $data->[1];
 27 }
 28 sub err_handle{
 29   die(scalar localtime,
 30     "\tPID: ",
 31     $$,
 32     " $0",
 33     " Ocorreu um erro de operação com o banco: ",
 34     DBI->err(), " ", DBI->errstr);
 35 }
```

saída:

```
Verificar salário: Thiago
Thiago 3000
```

A finalidade desta aplicação é um tanto óbvia. O usuário informa o nome do funcionário e verifica o salário. Mas nada impede que usuário insira uma string maliciosa que confunda nossa aplicação. Veja o que acontece se o usuário inserir código SQL em vez de um nome:

```
Verificar salário: ' or 'a'='a
Thiago 55537.4
Happy Worker 55537.4
The Big Boss 55537.4
Thiago 3000
Happy Worker 3000
The Big Boss 3000
Thiago 4500
Happy Worker 4500
The Big Boss 4500
Thiago 3600
Happy Worker 3600
The Big Boss 3600
Thiago 1500
Happy Worker 1500
The Big Boss 1500
```

Esta é uma demonstração simples de como um usuário pode tentar enganar o programa e fazê-lo mostrar informações indevidas. No caso, ele conseguiu a lista com o nome de todos os funcionários. Com essa informação ele pode interrogar o banco sobre cada funcionário. Se ele conseguir informações sobre a estrutura de tabelas do seu banco, poderá realizar comandos mais complexos que este. Uma falha como esta, que permite que uma entrada de usuário seja utilizada como código SQL, é conhecida como **SQL injection**.

15.6 Os principais comandos do SQL em programas Perl

Para trabalharmos com SQL é fundamental que você conheça um mínimo da linguagem. Veja alguns comandos de METADATA essenciais:

CREATE TABLE: este comando tem a finalidade de adicionar tabelas ao seu banco. A comando a seguir cria uma tabela "produtos" com as colunas: id_do_produto, descrição e valor.

```
CREATE TABLE produtos(
 id_do_produto INT NOT NULL, nome VARCHAR(50), descrição
VARCHAR(255), valor REAL NOT NULL, )
```

ALTER TABLE: modifica colunas da tabela.

```
ALTER TABLE produtos ADD fornecedor VARCHAR(50);
ALTER TABLE vendedor ADD telefone_recados VARCHAR(15);
ALTER TABLE produtos MODIFY fornecedor VARCHAR(100);
```

DROP TABLE: remove uma tabela.

```
DROP TABLE products;
```

Exemplo com o Perl:

```
  1 #exemplo15.4
  2 use 5.12.0;
  3 use DBI;
  4
  5 my $dsn = "DBI:mysql:database=mydb;host=localhost";
  6 close(STDERR); open(STDERR, ">> logfile");
  7
  8 my %attr = ( PrintError => 0, RaiseError => 0 );
  9 my $dbh = DBI->connect($dsn, "mydb_adm", "admpass",
\%attr)
 10 or err_handle();
 11 my $sql = <<SQL;
 12 CREATE TABLE produtos(
 13 id_do_produto INT NOT NULL,
 14 nome VARCHAR(50),
 15 descricao VARCHAR(255),
 16 valor REAL)
 17 SQL
 18
 19 #$return recebe um valor true se do for executado cor-
retamente ou
 20 #um valor false se ocorrer um erro na execução.
```

```
21 my $return = $dbh->do($sql);
22 err_handle() if not $return;
23 say "A tabela foi criada com sucesso";
24
25 sub err_handle{
26 die(scalar localtime,
27 "\tPID: ",
28 $$,
29 " $0",
30 " Ocorreu um erro de operação com o banco: ",
31 DBI->err(), " ", DBI->errstr);
32 }
```

Comandos para obter dados de tabelas: para obter dados de uma tabela, a função SQL mais comum ***SELECT***. Vimos diversos exemplos com a função SELECT. Eu recomendo muito que você se aprofunde em SQL:2008 e nas características do banco que você usa em seu trabalho. Caso você seja consultor, busque mercados mais excêntricos e tente aprender, além do **MySQL,** mais um banco, como o Oracle ou PostgreSQL. Eu digo isto aqui porque a função ***SELECT*** é muito importante e rica, com diversas opções. Mas nossa limitação será em obter dados com o ***SELECT*** e as cláusulas ***WHERE, ORDER BY*** e ***COUNT***.

COUNT permite que saibamos com antecedência quantas linhas serão afetadas pela nossa query.

```
 1 #exemplo15.5
 2 use 5.12.0
 3 use DBI;
 4
 5 my $dsn = "DBI:mysql:database=mydb;host=localhost";
 6 close(STDERR); open(STDERR, ">> logfile");
 7
 8 my %attr = ( PrintError => 0, RaiseError => 0 );
 9 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass",
\%attr)
10  or err_handle();
11
12 #retorna o número de linhas da tabela cargos
```

```
 13 my $sql = "SELECT COUNT(*) FROM cargos";
 14 my $sth = $dbh->prepare($sql);
 15 say $sth->execute; #Retorna True se o SQL executou cor-
retamente
 16 say "@{$sth->fetch}"; #informa os número de linhas
 17
 18
 19 sub err_handle{
 20   die(scalar localtime,
 21   "\tPID: $$ $0 Ocorreu um erro de operação com o ban-
co: ",
 22   DBI->err(), " ", DBI->errstr);
 23 }

saída:

1
5
```

O método ***execute*** do objeto ***$sth*** retorna um valor verdadeiro se a execução foi ok ou um valor booleano falso se ocorreu algum problema. Este é um método auxiliar para manusear erros, principalmente em queries criadas de forma dinâmica. O método ***fetch*** retorna uma referência a um array com os valores retornados pelo banco. Neste caso temos apenas um valor.

Veja agora uma query com ***ORDER BY*** e com ***WHERE.*** A cláusula WHERE é como um condicional. Podemos traduzir a cláusula do exemplo15.6 como:

"SELECIONE os empregados e seus cargos DAS TABELAS empregados e cargos ONDE o id_cargo da tabela empregado seja igual ao id_cargo da tabela cargos".

```
  1 #exemplo15.6
  2 use 5.12.0;
  3 use DBI;
  4 my $dsn = "DBI:mysql:database=mydb;host=localhost";
  5 close(STDERR); open(STDERR, ">> logfile");
  6
```

```
  7 my %attr = ( PrintError => 0, RaiseError => 0 );
  8 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass",
\%attr)
  9  or err_handle();
 10 my $sql = <<SQL;
 11 SELECT empregado.nome_empregado, cargos.cargo
 12 FROM empregado,cargos
 13 WHERE empregado.id_cargo = cargos.id_cargos
 14 ORDER BY cargos.cargo
 15 SQL
 16
 17 my $sth = $dbh->prepare($sql);
 18 $sth->execute;
 19
 20 if (DBI->err()) {
 21 err_handle()
 22 }
 23 while(my $data = $sth->fetch) {
 24  printf"Funcionario: %s\tcargo: %s\n", $data-
->[0],$data->[1];
 25 }
 26
 27 sub err_handle{
 28 die(scalar localtime,
 29 "\tPID: ",
 30 $$,
 31 " $0",
 32 " Ocorreu um erro de operação com o banco: ",
 33 DBI->err(), " ", DBI->errstr);
 34 }
```

Uma vantagem de utilizar a cláusula ***ORDER BY*** do SQL é não ter que organizar os dados retornados em um loop em seu programa, significando ganho de desempenho e redução de código.

Mais um ponto interessante a ser ressaltado é a possibilidade de usar "**place holders".** Place holder é um metacaractere adicionado ao comando que será enviado ao SQL, deixando o comando armazenado em cache para execução futura. Isso resulta em um ganho de performance caso o programa tenha a intenção de

executar a mesma query diversas vezes. Porém, é uma perda de performance se a query for executada apenas uma vez. Veja o trecho:

```
10 my $sql = <<SQL;
11 SELECT empregado.nome_empregado, cargos.cargo
12 FROM empregado,cargos
13 WHERE empregado.id_cargo = cargos.id_cargos
14 ORDER BY cargos.cargo
15 SQL
16
17 my $sth = $dbh->prepare($sql);
18 $sth->execute;
```

Apesar de aumentar um pouco o código, **a utilização de place holder protege o seu código dos ataques de SQL injection.** Como? Qualquer string enviada para a query via ***bind_param*** é interpretada como uma string comum e não como um comando. Veja o código completo e o exemplo de execução:

```
 1 #exemplo15.7
 2 use 5.12.0;
 3 use DBI;
 4
 5 my $dsn = "DBI:mysql:database=mydb;host=localhost";
 6 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass");
 7 print "Verificar salário: ";
 8 chomp (my $nome = <STDIN>);
 9
10 my $sql = <<SQL;
11 select empregado.nome_empregado,cargos.salario
12 from empregado,cargos where empregado.id_cargo=cargos.
id_cargos
13 and empregado.nome_empregado= ?
14 SQL
15 my $sth = $dbh->prepare($sql);
16 $sth->bind_param(1, $nome);
17 $sth->execute;
18
19 while(my $data = $sth->fetch) {
20 say $data->[0],"\t", $data->[1];
21 }
```

Execute o programa e note que, mesmo se enviarmos as mesmas entradas que geram SQL injection, o usuário não consegue confundir o programa.

15.7 Inserindo dados nas tabelas

Para inserir dados nas tabelas basta criarmos uma query SQL com o comando INSERT do SQL:2008:

INSERT INTO nome_da_tabela VALUES(valor_coluna1, valor_coluna2,...);

ou

INSERT INTO nome_da_tabela SET (nome_coluna1='valor1,nome_coluna2='valor2');

No Perl, com o DBI, podemos criar a query como uma string comum e inserir os dados com o método ***prepare*** e ***execute*** ou, simplesmente, com o método ***do***:

```
1 #exemplo15.8
  2 use 5.12.0;
  3 use DBI;
  4
  5 my $dsn = "DBI:mysql:database=mydb;host=localhost";
  6
  7 close(STDERR); open(STDERR, ">> logfile");
  8
  9 my %attr = ( PrintError => 0, RaiseError => 0 );
 10 my $dbh = DBI->connect($dsn, "mydb_user", "mydbpass",
\%attr)
 11 or err_handle();
 12
 13 print "Informe a habilidade a ser inserida: ";
 14 chomp(my $habilidade = <STDIN>);
 15 print "Descreva a habilidade (max 255 caracteres): ";
 16 chomp(my $desc = <STDIN>);
 17 check_for_sec($habilidade, $desc);
 18
 19 #retorna o número de linhas da tabela cargos
 20 my $sql = "INSERT INTO habilidades SET".
 21 " nome_habilidade='$habilidade',descricao_
habilidade='$desc'";
```

```
 22 my $sth = $dbh->prepare($sql);
 23 $sth->execute or err_handle(); #Retorna True se o SQL
executou corretamente
 24 $dbi->close;
 25
 26 sub err_handle{
 27   die(scalar localtime,
 28     "\tPID: ",
 29     $$,
 30     " $0",
 31     " Ocorreu um erro de operação com o banco: ",
 32     DBI->err(), " ", DBI->errstr);
 33 }
 34
 35 sub check_for_sec{
 36 #check_for_sec verifica se algum caractere das strings
que serão enviadas
 37 #para a query não contém caracteres que podem confundir
o programa.
 38   foreach my $string (@_){
 39  my @char_seq = split(//, $string);
 40  foreach my $char (@char_seq){
 41        say $char;
 42        die "Entrada insegura" if $char !~ /\w|\s/;
 43  }
 44   }
 45 }
```

Todo o trecho:

```
22 my $sth = $dbh->prepare($sql);
23 $sth->execute or err_handle();
```

Pode ser substituído por:

```
22 $dbh->do($sql) or err_handle();
```

15.8 As diferenças entre os métodos do( ) e prepare( )

Até o momento usamos o método ***do*** para realizar queries que criam tabelas ou inserem dados no banco e o método ***prepare*** para executar apenas queries de SELECT no banco. O método ***do( )*** é mais eficaz para realizar queries que não se repetirão na execução do mesmo processo. Por exemplo, se tiver que executar vários INSERTS na mesma tabela, o método ***prepare*** é o mais eficaz. O método ***prepare*** também é importante para evitarmos ataques de SQL injection, tornando-se uma ótima escolha em casos onde a query é criada dinamicamente e recebe entradas do usuário. O método ***do*** é muito eficiente se a query não é repetitiva durante a execução do seu programa. Ou seja, é mais eficiente em queries que criam ou modificam tabelas, pois elas não se repetem por diversas vezes.

15.9 O que ler agora?

Programação com banco de dados é extremamente importante para sistemas web e aplicações robustas em geral. Quanto mais você conhecer sobre o padrão SQL, álgebra relacional e as peculiaridade do sistema de gestão de banco de dados que você utiliza, mais rápidos e seguros serão os seus programas. As minhas recomendações de leitura são:

- SQL Bible, de Alex Kriegel e Boris M. Trukhnov, Wiley Publishing.
- Programming the Perl DBI, de Tim Bunce, editora O'Reilly.
- A documentação do sistema de gestão de bancos de dados relacionais que você adotar.

Exercícios

1. Faça um projeto de um banco de dados simples para cadastrar clientes de um consultório médico.

2. Monte um programa com uma versão em linha de comando e outra com ***Curses::UI*** para pesquisar, adicionar clientes e atualizar os dados dos clientes.

3. Monte um aplicativo de agenda simples. Você pode usar o módulo ***Date::Simple*** para ajudar com as datas.

4. Baseando-se na classe Person, do capítulo 13, crie um banco para cadastrar os contribuintes de uma cidade, com os documentos, a renda anual e o endereço.

Capítulo 16. Suas Aplicações na Web com CGI.pm

O Perl foi uma das primeiras linguagens para desenvolvimento de páginas dinâmicas. O próprio PHP iniciou baseado nas capacidades web do Perl. No início da web dinâmica, os sites eram feitos em CGI (ver RFC 3875). O CGI é um programa externo ao web server que gera uma saída em formato HTML e a entrega de volta ao servidor web. Por ser uma tecnologia antiga, o CGI tem algumas limitações. Ainda é muito usado, principalmente com as implementações de **FastCGI** e do Apache **Mod_perl**, que permitem que scripts CGI sejam executados superando-se as dificuldades impostas pela tecnologia antiga.

Para facilitar o desenvolvimento de DHTML com o Perl, foi desenvolvido o módulo ***CGI.pm***, que é mantido e atualizado até hoje, de forma que ele acompanha as evoluções da internet e das necessidades do programadores.

A primeira coisa a saber sobre o CGI.pm é a sua implementação orientada a objeto das funções do HTML. Assim, podemos criar templates web de forma mais elegante e organizada que o HTML. Veja o exemplo16.1:

```
1 #exemplo16.1
2 use 5.12.0;
3 use CGI;
4
5 my $cgi = CGI->new;
6 say $cgi->header();
7 say $cgi->start_html('Olá, mundo!!');
8 say $cgi->h1('Bem vindo à Programação Web com Perl');
9 say $cgi->end_html;
```

saída:

```
Content-Type: text/html; charset=ISO-8859-1
```

```
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.
dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
<head>
<title>Olá mundo!!</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body>

<h1>Bem vindo à Programação Web com Perl</h1>

</body>
</html>
```

Você pode perceber que o CGI.pm permite a criação mais organizada de uma página web simples. Mas não apenas isso, pois o CGI.pm mapeia todas as tags do HTML 3 e 4 em métodos, permitindo criar documentos HTML de forma dinâmica ou usá-los apenas para gerar o HTML que será usado no site.

Um exemplo mais complexo:

```
  1 #exemplo16.2
  2 use 5.12.0;
  3 use CGI;
  4
  5 my $cgi = CGI->new();
  6 say $cgi->header();
  7 say $cgi->start_html(-title=>'TICursos.net',
  8                      -author=>'thiago@ticursos.net',
  9                      -base=>'true',
 10                      -meta=>{'keywords'=>'Perl, script,
web, cgi',
 11                                'copyright'=>'copyright
2010 TICursos'},
 12                   #-style=>{'src'=>'/styles/style1.
css'}, #adiciona uma CSS
 13                      -BGCOLOR=>'blue');
 14
```

```
 15 say $cgi->h3('Hello world of Perl'), $cgi->hr;
 16 #table cria a tabela, tr cria uma linha de tabela e td
gera uma célula.
 17 say $cgi->table(
 18  $cgi->Tr({-align => 'CENTER'},
 19  [
 20        $cgi->td([
 21              #carrega uma imagem - substitua aqui pela
imagem que quiser
 22              $cgi->img({src=>'/logo.jpg'}),
 23              $cgi->h3('Bem Vindo à TICursos'),
 24        ]),
 25        $cgi->td([
 26              $cgi->blockquote("O curso de Perl Onli-
ne"),
 27              $cgi->blockquote("Só aqui na TICursos"),
 28        ]),
 29  ]
 30 ));
 31
 32 say $cgi->end_html;
```

saída HTML:

```
Content-Type: text/html; charset=ISO-8859-1


<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.
dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US"
xml:lang="en-US">
<head>
<title>TICursos.net</title>
<link rev="made" href="mailto:thiago%40ticursos.net" />
<base href="http://localhost" />
<meta name="keywords" content="Perl, script, web, cgi" />
<meta name="copyright" content="copyright 2010 TICursos" />
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
</head>
<body bgcolor="blue">
```

```
<h3>Hello world of Perl</h3><hr />
<table><tr align="CENTER"><td><img src="/logo.jpg" /></td>
<td><h3>Bem Vindo à TICursos</h3></td></tr> <tr align="CEN
TER"><td><blockquote>O curso de Perl Online</blockquote></
td> <td><blockquote>Só aqui na TICursos</blockquote></td></
tr></table>

</body>
</html>
```

Mas eu acho que simplesmente ver saídas HTML deve ser um pouco constrangedor. Nós queremos páginas web. Então vamos colocar esse script Perl na pasta **cgi-bin** do nosso Apache. Para instalar o servidor Apache, você pode utilizar a versão da sua distribuição Linux ou baixar a versão para instalar em www.apache.org - tanto Unix quanto Windows. No meu sistema **Fedora 13**, a pasta fica em **/var/www/cgi-bin**. Isso varia conforme a distribuição do Linux e as opções configuradas no Apache. Nesta pasta vou colocar meu script com o nome **first_page.pl** e dar permissão de execução com o comando **chmod +x first_page.pl.** Ao executar no browser teremos a página:

Neste exemplo o uso do Perl não é uma vantagem real sobre o uso do HTML, além do simples uso de uma programação mais organizada, coisa que pode ser compensada por um editor HTML com templates e interface gráfica. A grande vantagem do Perl é a possibilidade da página ser dinâmica. Imagine uma página que

gere uma mensagem randômica para cada acesso. Vamos pegar 10 mensagens, colocar em um banco de dados e acessar via CGI.

Primeiro vamos criar um banco, o mydb, que iremos usar nos nossos exemplos web. Inicialmente ele terá uma tabela com duas colunas para gerar a mensagem de fortuna:

```
--
-- Banco de Dados: `mydb`
--

-- --------------------------------------------------------

--
-- Estrutura da tabela `mensagens`
--

CREATE TABLE IF NOT EXISTS `mensagens` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `mensagem` varchar(500) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_
ci AUTO_INCREMENT=11 ;

--
-- Extraindo dados da tabela `mensagens`
--

INSERT INTO `mensagens` (`id`, `mensagem`) VALUES
(1, 'Bem-aventurado o homem que não anda segundo o conselho
dos ímpios, nem se detém no caminho dos pecadores, nem se
assenta na roda dos escarnecedores.'),
(2, 'A suspeita sempre persegue a consciência culpada; o
ladrão vê em cada sombra um policial'),
(3, 'Choramos ao nascer porque chegamos a este imenso cená-
rio de dementes.'),
(4, 'Os ricos fazem tudo pelos pobres, menos descer de suas
costas.'),
```

```
(5, 'Todos pensam em mudar o mundo, mas ninguém pensa em
mudar a si mesmo.'),
(6, 'O diabo pode citar as Escrituras quando isso lhe con-
vém.'),
(7, 'A nossa felicidade será naturalmente proporcional em
relação à felicidade que fizermos para os outros.'),
(8, 'Se você não falha em pelo menos 90% das vezes, seus
objetivos não foram ambiciosos o suficiente.'),
(9, 'O orgulho é a fonte de todas as fraquezas, porque é a
fonte de todos os vícios.'),
(10, 'Ter fé é assinar uma folha em branco e deixar que
Deus nela escreva o que quiser.');
```

Uma vez que o banco está criado, poderemos criar o CGI que acessa o banco de dados:

```
  1 #!/usr/local/bin/perl -T
  2 #A primeira coisa é evocar o Perl com o she-bang, mesmo
no Windows
  3 #
  4 #exemplo16.3
  5 use strict;
  6 use warnings;
  7 use DBI;
  8 use CGI;
  9 use CGI::Carp('fatalsToBrowser'); #desativar esta linha
quando em produção.
 10
 11 #Evita ataques de DoS (Denial of Service)
 12 $CGI::DISABLE_UPLOADS = 1;
 13 $CGI::POST_MAX        = 102_400; # 100 KB
 14
 15 #Conecta-se com o banco
 16 my $dsn = "DBI:mysql:database=mydb;host=localhost";
 17 my %attr = ( PrintError => 1, RaiseError => 1 );
 18 my $dbh = DBI->connect($dsn, "thiag054_webuser", "goA-
head", \%attr)
```

```
 19  or err_handle();
 20
 21 #Gera randomicamente um número, que será usado para
identificar a mensagem.
 22 my $number = int(rand(10)) + 1;
 23
 24 #Gera a string que será enviada ao banco.
 25 my $select = "SELECT mensagem FROM mensagens WHERE
id=$number";
 26
 27 #Prepara e executa a query
 28 my $sth = $dbh->prepare($select);
 29 $sth->execute;
 30
 31 my $mensagem;
 32
 33 #Obtém a mensagem:
 34 while(my $ref = $sth->fetch){
 35     $mensagem = $ref->[0];
 36 }
 37
 38 #Gera um CGI conforme já vimos.
 39 my $cgi = CGI->new();
 40 print $cgi->header(-type => 'Content-Type',
 41 -charset => 'iso-8859-1',
 42 );
 43 print $cgi->start_html(-title=>'TICursos.net',
 44                      -author=>'thiago@ticursos.net',
 45                      -base=>'true',
 46                      -meta=>{'keywords'=>'Perl, script,
web, cgi',
 47                                 'copyright'=>'copyright
2010 TICursos'},
 48                   #-style=>{'src'=>'/styles/style1.
css'}, #adiciona uma CSS
 49                      -BGCOLOR=>'cyan');
 50
 51 print $cgi->h3({-align=>'center'},'Mensagem do
```

```
S&#225;bio:'), $cgi->hr;
 52
 53 #A mensagem é dinamicamente adicionada à página:
 54 print $cgi->p({-align => 'left'}, $mensagem);
 55
 56 print $cgi->end_html;
 57
 58 sub err_handle{
 59  #coloque aqui sua função para lidar com erros.
 60 }
```

Você pode ver a página funcionando em: http://www.ticursos.net/cgi-bin/mensagem.pl.

Assim como a mensagem, podemos criar páginas que carreguem imagens ou que montem qualquer estrutura de página de forma dinâmica.

16.1 Criando formulários com Perl e CGI.pm

Após a explicação inicial sobre CGI, é interessante criarmos uma aplicação com formulários. Então, inicialmente criamos um formulário e definimos a sua ação:

```
$cgi->start_form(-method=>'post', -action=>$cgi->url);
```

Esta linha inicia um formulário. A ação do formulário informa para qual aplicação/script os dados do formulário devem ser enviados. Um formulário pode conter, basicamente:

- Campos de Texto, ou ***textfield***, que armazenam ou exibem uma string.
- Área de Texto, ou ***textarea***, que armazena e/ou exibe uma string longa, com centenas de caracteres.
- Campo de senhas, ou ***password_field***, armazena uma string mas exibe somente o caractere asterisco *.
- ***Menus Popups***, ***Scroll Lists***, ***ComboBoxes***, ***Radio Button***. Como as interfaces gráficas padrão, a interface de um formulário pode conter todos estes elementos.

Vamos fazer um formulário simples para verificarmos como funciona o envio e a recepção de mensagens no CGI:

```
  1 #!/usr/local/bin/perl -T
  2 #exemplo16.4
  3 use strict;
  4 use warnings;
  5 use CGI;
  6 use CGI::Carp ('fatalsToBrowser');
  7
  8 BEGIN{
  9 #Configurações de segurança
 10     $CGI::DISABLE_UPLOADS = 1;
 11     $CGI::POST_MAX        = 102_400; # 100 KB
 12 }
 13
 14 my $cgi = CGI->new;
 15
 16 #inicia a página html
 17 print $cgi->header();
 18 print $cgi->start_html(-title => "Mail me with Perl");
 19
 20 if (not $cgi->param){
 21 #Se não foram enviados parâmetros, cria a página de
formulário.
 22     print $cgi->h3("Enviar mensagem!"),
 23           $cgi->start_form(-action => $cgi->url),
 24           $cgi->p("Nome: ",$cgi->textfield(-
name=>'nome', -size=>50)),
 25           $cgi->p("Digite sua Mensagem: ", $cgi->br,
 26             $cgi->textarea(-name=>'mensagem', -rows=>5,
-columns=>50)),
 27           $cgi->submit(-name=>'submit');
 28 }
 29 else{
 30 #Se foram enviados parâmetros, eles devem ser processa-
dos.
 31     print $cgi->p("O nome informado eh: ".$cgi-
```

```
->param("nome").$cgi->br.
 32         "A mensagem eh: " . $cgi->param("mensagem"));
 33    print $cgi->p($_ . " => " . $cgi->param($_)) foreach
$cgi->param( );
 34 }
 35
 36 print $cgi->end_form;
 37 print $cgi->end_html;
```

Você pode ver esta página em funcionamento em: http://www.ticursos.net/cgi-bin/formulario.pl. Os elementos de um formulário são parâmetros de uma query HTTP. Então, os ***textfield*** e ***textarea*** são parâmetros quando iniciados com dados.

Em ***$cgi->start_form*** definimos a ação a ser executada quando o botão ***submit*** for pressionado. No caso ***-action*** informa a URL para a qual os valores dos parâmetros devem ser enviados. O nosso CGI envia para ele mesmo ***$cgi->start_form(-action => $cgi->url)***. O método **param** do objeto CGI retorna verdadeiro se a query do HTTP tiver algum parâmetro, ou falso se não tiver nenhum. Em nosso programa, não havendo parâmetros, o formulário é criado. O método **param**, quando tem algum nome de parâmetro como argumento, irá retornar o valor armazenado pelo parâmetro. Assim podemos verificar se é a primeira vez que o cliente executa o nosso formulário e se este deve processar alguma informação com

```
$cgi->param("param_name").
```

Analisando em detalhes nosso formulário, criamos um campo de texto, uma área de texto e um botão ***submit***. Agora vamos adicionar funcionalidade neste formulário, para envio de e-mails. Você pode ver a página em funcionamento em http://ticursos.net/cgi-bin/mail.pl**.**

```
  1 #!/usr/local/bin/perl -T
  2 #exemplo16.5
  3 #strict e warnings evitam problemas por parte do pro-
gramador.
  4 use strict;
  5 use warnings;
  6
  7 #caminho da biblioteca de módulos Perl do sistema
  8 use lib("/home/thiago/perl/usr/lib/perl5/");
```

```
 9 use CGI;
10 use CGI::Carp ('fatalsToBrowser');
11
12 #Data::Validate::Email permite verificar se o e-mail é
válido ou não.
13 use Data::Validate::Email qw/is_email_rfc822/;
14
15 BEGIN{
16 #Statements inseridos em BEGIN são executados na compi-
lação do programa.
17 #As configurações DISABLE_UPLOADS e POST_MAX evitam
ataques de DoS.
18     $CGI::DISABLE_UPLOADS = 1;
19     $CGI::POST_MAX        = 102_400; # 100 KB
20
21 #As configurações a seguir evitam exploits
22     $ENV{'PATH'} = '/usr/sbin';
23     delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
24 }
25
26 my $cgi = CGI->new;
27 print $cgi->header(-type => 'Content-Type',
28              -charset => 'iso-8859-1',
29              );
30 print $cgi->start_html(-title=>'TICursos.net',
31                       -author=>'thiago@ticursos.net',
32                       -base=>'true',
33                       -meta=>{'keywords'=>'Perl,
script, web, cgi',
34                               'copyright'=>'copyright
2010 TICursos'},
35                   #-style=>{'src'=>'/styles/style1.
css'}, #adiciona uma CSS
36                       -BGCOLOR=>'orange');
37
38 if (not $cgi->param){
39 #Cria formulário
40     print $cgi->h3("Enviar mensagem!"),
```

```
 41            $cgi->start_form(-action => $cgi->url),
 42            $cgi->p("Nome: ",$cgi->textfield(-
name=>'nome', -size=>50)),
 43            $cgi->p("Email: ",$cgi->textfield(-
name=>'email', -size=>50)),
 44            $cgi->p("Digite sua Mensagem: ",
 45               $cgi->br,
 46               $cgi->textarea(-name=>'mensagem',
-rows=>5, -columns=>50)
 47               ),
 48             $cgi->submit(-name=>'enviar');
 49 }
 50 else{
 51 #Se o formulário já está preenchido os dados são pro-
cessados aqui.
 52
 53  #Checa se o e-mail é válido
 54  my $email = is_email_rfc822($cgi->param("email"));
 55  if(not $email){
 56        print $cgi->h3("Email invalido." .
 57                 $cgi->param("email") .
 58                 " não é um email"
 59                 );
 60        exit(0);
 61        }
 62
 63  #Checa se nome não é uma tentativa de exploit
 64        if(not validate($cgi->param("nome"))){
 65              print $cgi->p("Nome Inválido");
 66              exit 0;
 67        }
 68
 69  #Checa se mensagem não é uma tentativa de exploit
 70  if(not validate($cgi->param("mensagem"))){
 71              print $cgi->p("Mensagem com formato sus-
peito. " .
 72                      "Não utilize: ; : $ % * & ( ) &lt
&gt"
```

```
 73                        );
 74                exit 0;
 75        }
 76
 77     if(send_message($cgi->param("nome"), $email, $cgi-
->param("mensagem"))){
 78        print $cgi->h2("Mensagem enviada com sucesso");
 79     }
 80 }
 81
 82
 83 print $cgi->end_form;
 84 print $cgi->end_html;
 85
 86 sub send_message {
 87 #Envia a mensagem via sendmail
 88  my ($from_name, $from_email, $message) = @_;
 89
 90  #Criamos um pipe de escrita para o programa sendmail
 91  open my $mail, "|-", "sendmail -t -f '$from_email' -i
"
 92        or die "Could not open sendmail: $!";
 93  print $mail <<END_OF_MESSAGE;
 94 To:thiago\@ticursos.net
 95 Subject: Web Site Feedback
 96
 97 Feedback from a user:
 98
 99 $message
100 END_OF_MESSAGE
101  close $mail or die "Error closing sendmail: $!";
102  return 1;
103 }
104
105 sub validate {
106 #Esta sub verifica se não existem caracteres suspeitos
nas mensagens, gerando proteção contra exploits.
107
```

```
108  my $texto = shift;
109  if ($texto =~ /[:;\$\%\*\&\(\)<>]/) {
110        return 0;
111  }
112  else {
113              return 1;
114       }
115 }
```

16.2 Desenvolvimento web com FCGI

O FCGI tem como vantagem manter uma programação estilo CGI mas com a mesma eficiência que o PHP ou o ASP. Uma aplicação ***FastCGI*** mantém o aplicativo compilado na memória RAM do servidor e pode também manter conexões com banco de dados e variáveis na memória, dependendo de como o código foi implementado. Essencialmente, uma aplicação FastCGI com Perl terá a seguinte estrutura:

```
1  #!/usr/bin/perl
2 #exemplo16.6
3     use CGI::Fast;
4     &inicializacao();
5     #Loop de requisição:
6     while ($q = new CGI::Fast) {
7         &processa_requisicao($q);
8     }
```

Assim, o código a seguir gera a saída que pode ser vista em http://www.ticursos.net/fcgi-bin/fcgi2.fpl.

```
1 #!/usr/local/bin/perl -T
2 #exemplo16.7
3 use CGI::Fast qw(:standard);
4     $COUNTER = 0;
5     while (my $cgi = new CGI::Fast) {
6         print $cgi->header;
```

```
 7          print $cgi->start_html("Fast CGI Rocks");
 8          print
 9              $cgi->h1("Fast CGI Rocks"),
10              "Invocation number ",b($COUNTER++),
11              " PID ",$cgi->b($$),".",
12              $cgi->hr;
13          print $cgi->end_html;
14      }
```

Ao executar a página você vai perceber que a variável ***$COUNTER*** torna-se persistente na memória e que o Apache mantém o mesmo processo na memória por um longo tempo, permitindo que diversas requisições sejam processadas pelo mesmo processo.

16.3 O que ler agora?

O desenvolvimento web profissional inclui habilidades de programação além de design. Se você deseja programar para web de forma profissional, eu recomendo que você leia:

1. Beginning Perl Web Development: From Novice to Professional, de Steve Suehring, editora Apress;
2. CGI Programming ont the World Wide Web, de Shishir Gundavaram, editora O'Reilly;
3. My SQL and Perl for the Web, de Paul DuBois, editora New Riders Publishing;
4. Perl Template Toolkit, de Darren Chamberlain e outros, editora O'Reilly.

Exercícios

1. Monte uma versão web dos exercícios do capítulo 15.

2. Pesquise no CPAN e na internet sobre: Catalyst Framework, ***CGI::Application***, Mojolicious Framework.

3. Crie um formulário web para o exercício 2 do capítulo 14. Faça com que o programa envie a imagem gerada via e-mail para o usuário.

Capítulo 17. Trabalhando com Threads

Quando estamos aprendendo tarefas novas é natural que façamos as coisas em uma espécie de passo a passo para cada atividade envolvida na tarefa. Porém, depois de certa expertise, queremos otimizar a execução da tarefa. Por exemplo, quando estamos aprendendo a cozinhar fazemos as coisas uma de cada vez. Com alguma experiência começamos a fazer coisas em paralelo, ou seja, enquanto a água ferve vamos untar a forma e enquanto o bolo assa, podemos começar a fazer a cobertura. Assim, quando o bolo estiver pronto a cobertura também estará quase que ao mesmo tempo pronta e nós iremos simplesmente pegar o resultado dos dois trabalhos e juntá-los colocando o creme de cobertura sobre o bolo. Se tivermos que esperar o bolo ficar pronto para depois iniciarmos o preparo da cobertura, o tempo de preparo será cerca de 20 ou 30 minutos maior...

Quando solicitamos a execução de um programa externo por um pipe ou pela função system para somente depois prosseguir com o programa é o mesmo que fazer o bolo para depois fazer a cobertura. O mesmo ocorre com nossos programas de exemplos que executam pings. O tempo de envio e recebimento de dados pela rede é muito superior ao tempo de processamento de um processador, e mesmo que de um IO de disco. Para melhorar isso a programação moderna e o Perl nos oferecem técnicas como multiprocessamento e multithreading. Agora veremos programas que utilizam apenas um processo e diversas threads.

17.1 Mas antes vem a pergunta: o que é uma thread???

Bem, vamos deixar as coisas simples no começo. E continuar cozinhando. Num almoço comum tipicamente brasileiro temos o arroz, o feijão, alguma carne e uma salada. Supondo que o tempo de preparo de cada elemento é de 15 minutos, num trabalho sequencial o nosso almoço demoraria 15 x 4 = 60 minutos... muito

tempo para quem está com fome. Mas qualquer fogão moderno tem quatro bocas, permitindo o cozimento em paralelo de cada elemento. Levando isso para um programa muito simples:

```
 1 #exemplo17.1
 2 use 5.12.0;
 3 use Benchmark;
 4
 5 sub cook_rice{
 6  sleep 5;
 7 }
 8 sub cook_beens{
 9  sleep 5;
10 }
11 sub cook_meat{
12  sleep 5;
13 }
14 sub make_salad {
15  sleep 5;
16 }
17
18 my $start_time = Benchmark->new;
19
20 make_salad;
21 cook_rice;
22 cook_beens;
23 cook_meat;
24
25 my $end_time = Benchmark->new();
26
27 say "Food is done in: ", timestr(timediff($end_time,
$start_time));

$ time perl exemplo17.1.pl
Food is done in: 20 wallclock secs ( 0.00 usr +  0.00 sys =
0.00 CPU)

real 0m20.030s
user 0m0.023s
sys  0m0.005s
```

NOTA: O comando ***time*** dos sistemas Linux mostra o tempo de execução do programa e dos tempos de sistema e usuário. Sem o comando ***time*** a saída será:
C:>perl exemplo17.1.pl
Food is done in: 20 wallclock secs (0.00 usr + 0.00 sys = 0.00 CPU)
Repare que o algoritmo tem tempo de execução de vinte segundos, o restante diz respeito ao compilador, IO de discos e ao consumo de recursos pelo próprio sistema operacional. Está última saída foi executada em um sistema Windows.

Agora vamos cozinhar com threads...

```
  1 #exemplo17.2
  2 use 5.12.0;
  3 use threads;
  4 use Benchmark;
  5
  6 #criamos as funções para preparar o almoço
  7 sub cook_rice{
  8  sleep 5;
  9 }
 10
 11 sub cook_beans{
 12  sleep 5;
 13 }
 14
 15 sub cook_meat{
 16  sleep 5;
 17 }
 18
 19 sub make_salad {
 20  sleep 5;
 21 }
 22
 23 my $start_time = Benchmark->new;
 24
 25 # O módulo threads facilita o nosso trabalho com threa-
ds,
```

```
 26 # permitindo um trabalho OOP e prático;
 27 # iniciamos uma thread com o método threads->new,
 28 # que tem como argumento a referência a uma função. A
função passada para
 29 # thread->new contém  a porção de código que será exe-
cutada
 30 # como processo paralelo
 31
 32 my $t1 = threads->new(\&cook_rice);
 33 my $t2 = threads->new(\&cook_beans);
 34 my $t3 = threads->new(\&cook_meat);
 35 my $t4 = threads->new(\&make_salad);
 36
 37 # O método join dos objetos threads fazem com que o
programa
 38 # principal aguarde a conclusão da thread para finali-
zar
 39 # ou utilizar o valor retornado.
 40
 41 my $time_rice = $t2->join();
 42 say "Rice done in $time_rice";
 43 my $time_beans = $t1->join();
 44 say "Beans done in $time_beans";
 45 my $time_meat = $t3->join();
 46 say "Meat done in $time_meat";
 47 my $time_salad = $t4->join();
 48 say "Salad done in $time_salad";
 49
 50 my $end_time = Benchmark->new();
 51 say "Food is done in: ", timestr(timediff($end_time,
$start_time));
```

saída:

```
C:>perl exemplo17.2.pl
Rice done in 5
Beans done in 5
Meat done in 5
Salad done in 5
Food is done in:  5 wallclock secs ( 0.05 usr +  0.00 sys =
0.05 CPU)
```

Caso esteja utilizando o Linux pode testar este programa com o comando ***time***.

Essencialmente os programas fazem as mesmas coisas, mas a versão multithread realiza todo o trabalho em um quarto do tempo da versão com apenas um processo.

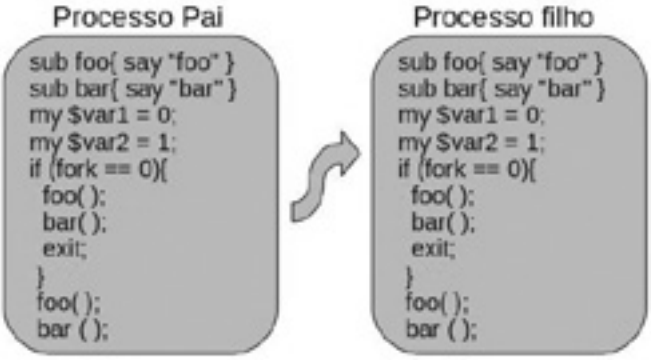
Nem sempre o trabalho realizado por um programa multithread é mais rápido que o executado por um programa sem threads. Imagine: para cada thread é iniciado um interpretador Perl, o que consome tempo, e carregamos mais um objeto para cada thread em nossa memória. Essas coisas consomem tempo e recursos do sistema.

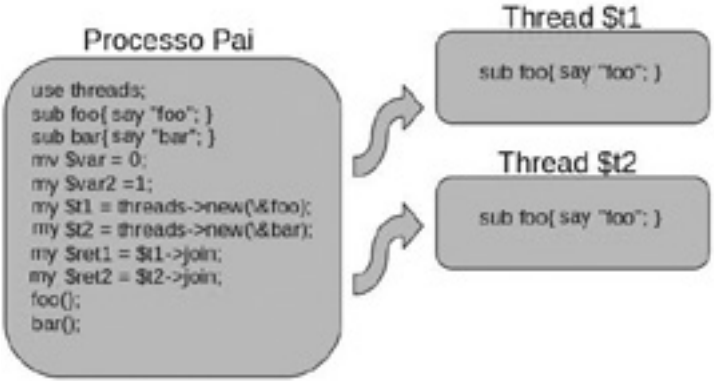
Este é um programa simples, que inicialmente define as funções (elas também podem estar no final do programa, é indiferente) e então cria uma thread para cada função com ***thread->new***. Os métodos ***thread->join*** ou ***thread->detach*** servem para fazer com que a thread tenha seu código executado antes do programa terminar, forçando que esperemos sua conclusão.

17.1.1 De forma mais técnica, uma thread...

Uma thread é uma forma de dividir o processamento, criando uma outra entidade de processamento com uma atividade específica ligada ao seu processo. A thread inicialmente foi o jeito encontrado pela Microsoft para criar uma forma de multiprocessamento, uma vez que os sistemas Windows não suportam bem a criação de diversos processos filhos como no Unix. No Unix a forma óbvia de multiprocessamento ou processamento paralelo era simplesmente a criação de processos filhos. Só que um processo filho é uma cópia completa de seu processo pai, já uma thread recebe apenas os dados e o código com os quais ela deve lidar. Tal é a dificuldade do Windows em lidar com processos filhos que a função ***fork*** do Perl cria, na verdade, uma nova thread e copia os dados do processo.

Dessa forma, uma ótima definição de thread é: "um processo filho sem gordura". No nosso exemplo de cozinha, podemos dizer que um fork dá a cada processo filho uma cozinha independente, e à thread apenas um utensílio e uma boca do fogão. A ilustração pode colocar as coisas de forma melhor:





17.2 Como trabalhar com threads

O nosso exemplo de preparar o jantar é uma das formas de trabalhar com threads: uma tarefa maior (fazer o jantar) é dividida em diversas tarefas pequenas e cada tarefa é realizada por uma thread, e o resultado é retornado para a thread principal. Esse modelo é o ***BOSS/WORKER (CHEFE/EMPREGADO)***, onde o trabalho da thread Boss é apenas gerenciar as threads trabalhadoras, que executam o processamento real dos dados.

Outra forma de trabalhar é com o modelo ***Pipeline***, onde o valor de retorno de cada thread é enviado para a thread seguinte. O terceiro modelo é ***Work Crew***. Aqui, diversas threads executam a mesma porção de código, porém com dados diferentes.

17.2.1 A segurança

Antes da versão 5.8.0 do Perl era inseguro trabalhar com threads, uma vez que os dados eram compartilhados entre todas as threads. Para tornar a programação com threads mais segura e fácil, o Perl adotou um modelo de trabalho denominado **ithread** ou "**interpreted threads**". Neste modelo, o Perl cria uma instância de interpretador para cada thread, tornando os dados individualizados, compartilhando apenas os dados de retorno. Para compartilhar outros dados temos que especificar claramente ao Perl que queremos fazer isso e temos que providenciar a criação dos semáforos. Os semáforos irão regrar o acesso a dados compartilhados.

17.2.2 Criando uma thread

Como em diversas situações, o Perl nos oferece um módulo que simplifica a nossa vida, no caso a diretiva ***use threads*** permite que criemos threads e trabalhemos com o formato OOP.

```
use threads

my $thr = threads->new(\&funcao);

sub funcao{
     algo útil aqui
}
```

O método ***new*** cria uma thread que executará o código da função que recebe como argumento. Se você precisar passar argumentos para a função, basta usar um dos formatos a seguir:

```
threads->new(\&funcao, "param1", $param2);
threads->new(\&funcao, @params_list);
```

Uma vez que as threads são funções executadas em paralelo, elas podem retornar ou não valor, e para recebermos o valor enviado pela função usamos o método ***join,*** que retorna o valor da função executada:

```
my $return_value = $thread->join( );
```

Um detalhe importante a ressaltar sobre o valor de retorno da thread em contexto de lista é: threads que retornam valores de lista devem ser criadas em contexto de lista, ou seja:

```
my ($thread) = threads->new(\&funcao);
my @ret = $thread->join;
```

A utilização de ***join*** também é importante, pois ela cuida da finalização e limpeza dos recursos utilizados pelas threads. Porém, caso não queiramos ou não precisemos esperar a execução de todas as threads para encerrar um processo, podemos usar o método ***detach*** como a seguir:

```
 1 #exemplo17.3
 2 use 5.12.0;
 3 use threads;
 4
 5 sub sub1{ sleep 2; say "Eu dormi 2 segundos."}
 6 sub sub2{ sleep 2; say "Eu também dormi 2 segundos."}
 7
 8 my $t1 = threads->new(\&sub1);
 9 my $t2 = threads->new(\&sub2);
10
11 $t1->detach();
12 $t2->detach();
13
14 while($t1->is_running() && $t1->is_running()){
15  sleep 1;
16 }

saída:

Eu dormi 2 segundos.
Eu também dormi 2 segundos.
```

O método ***is_running*** de módulo ***threads*** retorna true enquanto a thread está sendo executada. Neste exemplo nós precisamos garantir que apenas uma thread tenha terminado o serviço para podermos sair. Então usamos ***detach***

para dizer ao Perl que não queremos o valor de retorno e que ele nem precisa esperar que a thread termine para sair. Estamos dizendo ao Perl que vamos cuidar disso tudo.

Além disso, temos a propriedade ***tid***, que é o identificador da thread. Assim como o ***PID*** identifica processos, o ***TID*** identifica uma thread.

```
 1 #exemplo17.4
 2 use 5.12.0;
 3 use threads;
 4
 5 my $t1 = threads->new(\&hard_work);
 6 my $t2 = threads->new(\&hard_work);
 7 $t1->detach();
 8 $t2->detach();
 9
10 while($t1->is_running() || $t2->is_running()){
11  next
12 }
13
14 sub hard_work{
15  my $thread = threads->self;
16  my $x = 0;
17  while($x<10){
18        say "x is $x on TID: ", $thread->tid;
19        $x++;
20        sleep 1;
21  }
22 }

saída:

x is 0 on TID: 1
x is 0 on TID: 2
x is 1 on TID: 1
x is 1 on TID: 2
x is 2 on TID: 1
x is 2 on TID: 2
```

```
x is 3 on TID: 1
x is 3 on TID: 2
x is 4 on TID: 2
x is 4 on TID: 1
x is 5 on TID: 1
x is 5 on TID: 2
x is 6 on TID: 2
x is 6 on TID: 1
x is 7 on TID: 1
x is 7 on TID: 2
x is 8 on TID: 1
x is 8 on TID: 2
x is 9 on TID: 1
x is 9 on TID: 2
```

Como o TID corre de acordo com a ordem de criação da thread, uma função pode definir execução diferente em relação ao TID:

```
  1 #exemplo17.5
  2 use 5.12.0;
  3 use threads;
  4
  5 my $t1 = threads->new(\&hard_work);
  6 my $t2 = threads->new(\&hard_work);
  7 my $t3 = threads->new(\&hard_work);
  8
  9 $t1->detach();
 10 $t2->detach();
 11 $t3->detach();
 12
 13 while($t1->is_running() || $t2->is_running() || $t3-
->is_running){
 14  next
 15 }
 16
 17 sub hard_work{
 18  my $thread = threads->self;
```

```
19  my $x = 0;
20  if($thread->tid == 1){
21        while($x<10){
22              say "x is $x on TID: ", $thread->tid;
23              $x++;
24              sleep 1;
25        }
26  }
27  elsif($thread->tid == 2){
28        while($x < 40){
29              say "x is $x on TID: ", $thread->tid();
30              $x = ($x + 1) * 4;
31              sleep 1;
32        }
33  }
34  else {
35        say "Thread ", $thread->tid(), " não prevista";
36        return;
37  }
38 }
```

saída:

```
x is 0 on TID: 2
x is 0 on TID: 1
Thread 3 não prevista
x is 4 on TID: 2
x is 1 on TID: 1
x is 20 on TID: 2
x is 2 on TID: 1
x is 3 on TID: 1
x is 4 on TID: 1
x is 5 on TID: 1
x is 6 on TID: 1
x is 7 on TID: 1
x is 8 on TID: 1
x is 9 on TID: 1
```

Perceba, neste exemplo, que encerramos a execução de uma ***thread*** com a função ***return***, enquanto que encerramos a execução de um processo filho com ***exit***. A função ***return*** encerra uma função. Se usarmos ***exit*** dentro de uma ***thread***, vamos encerrar a execução de todo o programa.

Temos aqui um exemplo de threads prático. Uma vez que aplicações que utilizam a rede são mais lentas por natureza, podemos dividir diversas solicitações a serviços da rede em ***threads*** para cada solicitação:

```
  1 #exemplo17.6
  2 use 5.12.0;
  3 # Verifica se o comando perl foi compilado com suporte
a threads
  4 BEGIN {
  5     use Config;
  6     die "No interpreter threads!\n" unless
$Config{useithreads};
  7 }
  8
  9 use threads;
 10 use Net::Ping::External qw/ping/;
 11
 12 sub hard_work {
 13  my $host = shift;
 14     my $self = threads->self;
 15     print "Thread ", $self->tid, " started \n";
 16     my $is_up = ping(host => $host);
 17     print "Thread ", $self->tid, " ending \n";
 18     return [$is_up, $host];
 19 }
 20
 21 my @threads;
 22 foreach (qw/www.brasport.com.br www.ticursos.net www.
freebsd.org www.aol.com www.perl.org/){
 23  push @threads, threads->new(\&hard_work, $_);
 24 }
 25 while (my $thread = shift @threads) {
 26     my $monitor = $thread->join;
```

```
27     if ($monitor->[0]){
28        say "Host ", $monitor->[1], " is up";
29     }
30     else{
31        say "Host ", $monitor->[1], " is down";
32     }
33 }
```

saída:

```
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 2 ending
Thread 1 ending
Host www.brasport.com.br is up
Host www.ticursos.net is up
Thread 3 ending
Host www.freebsd.org is up
Thread 5 ending
Thread 4 ending
Host www.aol.com is down
Host www.perl.org is up
```

17.3 Compartilhando dados entre threads

Como dissemos, as threads tornaram-se muito mais seguras a partir da versão 5.8, impedindo o compartilhamento de dados de forma indiscriminada. Porém, em alguns algoritmos pode ser interessante compartilhar dados. Para isto contamos com o suporte de ***threads::shared***. Mas temos que atentar para a segurança, uma vez que compartilhar dados ou permitir que diversas threads possam modificar uma determinada informação pode trazer problemas de ***race conditions***.

```
1 #exemplo17.7
2 use 5.12.0;
3 use threads;
```

```
  4 use threads::shared;
  5
  6 #shared é um atributo da variável, identificado em
threads::shared.
  7 #podemos também utilizar a função sharede, como:
  8 #my $var1 = 1; shared($var1);
  9
 10 my $var1 : shared = 1;
 11 my $var2 = 2;
 12 my @threads;
 13 for(1..3){
 14 #Criamos uma thread para cada loop e colocamos o valor
de retorno
 15 #no array @threads ($var2 += 5).
 16  push @threads, threads->new(sub{$var1 *= 2; $var2 += 5
})->join();
 17 }
 18 say "\$var1 mudou de 1 para $var1";
 19 say "as mudanças de \$var2 não se propagam: \$var2 =
$var2";

saída:

$var1 mudou de 1 para 8
as mudanças de $var2 não se propagam: $var2 = 2
```

Porém, acessar variáveis por diversas threads pode ser perigoso, pois quando mais de uma thread executa atividade de escrita, pode gerar uma condição de competição ou **race condition**. Veja este exemplo:

```
  1 #exemplo17.8
  2 use 5.12.0;
  3 use threads;
  4 use threads::shared;
  5 my $a :shared = 2;
  6 my $b :shared = 0;
  7 my $c :shared = 0;
```

```
  8 my $thr1 = threads->create(sub { $b = $a; $a = $b + 1;
});
  9 my $thr2 = threads->create(sub { $c = $a; $a = $c + 1;
});
 10 $thr1->join();
 11 $thr2->join();
 12 say "$a, $b, $c";

saída:

4, 2, 3
```

NOTA: Esta saída varia conforme a execução...

Bem, neste caso a saída depende do sistema e da condição na qual executamos o programa. Ao final, ***$a*** pode ser 4 ou 3 e ***$b*** ou ***$c*** podem ser 2 ou 3... em um programa real, uma condição de competição mal programada pode ter um resultado trágico, significando perda de milhões de dólares.

Para solucionar estes problemas, tínhamos inicialmente a função lock. Bem, ainda temos, mas é uma forma que despende esforços hoje desnecessários. Para mais detalhes sobre lock, consulte ***perldoc perlthrtut.***

Hoje temos técnicas como ***Filas e Semáforos.***

17.4 A queue

Uma queue ou fila é uma técnica que processa em threads uma sequência de dados. O módulo ***Thread::Queue*** permite esta facilidade.

```
  1 #exemplo17.9
  2 use 5.12.0;
  3 use threads;
  4 use Thread::Queue;
  5 use Net::Ping::External qw/ping/;
  6
```

```
 7 #Criamos uma lista com os dados que queremos processar.
 8 my @to_ping = qw/www.brasport.com.br www.ticursos.net
 9        www.perl.org 127.0.0.1 127.0.0.2/;
10
11 #criamos o objeto de filas.
12 my $queue = Thread::Queue->new();
13
14 #Cria uma fila de valores para serem processados pela
thread
15 $queue->enqueue(@to_ping);
16
17 #Define uma thread e a função.
18 #Podemos criar mais de uma thread aqui, para acelerar o
processamento
19 #e a queue vai garantir que cada elemento será proces-
sado apenas uma vez.
20
21 my ($t) = threads->new(\&sub_thr);
22 $t->join();
23
24 sub sub_thr{
25  for (0..$#to_ping){
26        my ($host) = $queue->dequeue_nb();
27        my @return = hard_work($host);
28        say $return[1], " is up!!!" if $return[0];
29        say $return[1], " is down!!!" if !$return[0];
30  }
31 }
32
33 sub hard_work{
34     my $host = shift;
35      my $self = threads->self;
36      print "Thread ", $self->tid, " started \n";
37      my $is_up = ping(host => $host, timeout => 2,
count => 1);
38      print "Thread ", $self->tid, " ending \n";
39      return ($is_up, $host);
40
41 }
```

```
saída:

Thread 1 started
Thread 1 ending
www.brasport.com.br is up!!!
Thread 1 started
Thread 1 ending
www.ticursos.net is up!!!
Thread 1 started
Thread 1 ending
www.perl.org is up!!!
Thread 1 started
Thread 1 ending
127.0.0.1 is up!!!
Thread 1 started
Thread 1 ending
127.0.0.2 is up!!!
```

O objeto ***Thread::Queue*** tem as seguinte interface:

Thread::Queue->new => cria uma nova queue

Thread::Queue->enqueue => adiciona elementos à queue

Thread::Queue->dequeue => remove elemento da queue

Agora iremos estudar os semáforos.

17.5 Os semáforos

Os semáforos funcionam como sinais que permitem ou bloqueiam o acesso de uma thread a uma variável compartilhada. A classe ***Thread::Semaphore*** define três principais métodos, a saber:

Threads::Semaphore->new => cria um novo objeto do tipo semáforo.

Threads::Semaphore->down => farol vermelho. A sua thread está usando código que não pode ser corrompido pelo acesso de outras variáveis.

Threads::Semaphore->up => farol verde. Sua thread liberou o código para uso.

```
 1 #exemplo17.10
 2 use threads;
 3 use threads::shared;
 4 use Thread::Semaphore;
 5
 6 #Cria um objeto semáforo
 7 my $semaphore = Thread::Semaphore->new();
 8 my $GlobalVariable :shared = 0;
 9
10 #Cria as threads para a mesma função
11     $thr1 = threads->create(\&sample_sub, 1);
12     $thr2 = threads->create(\&sample_sub, 2);
13     $thr3 = threads->create(\&sample_sub, 3);
14
15 #Declara a função
16     sub sample_sub {
17         my $SubNumber = shift(@_);
18         my $TryCount = 10;
19         my $LocalCopy;
20         while ($TryCount--) {
21             $semaphore->down();
22  #semáforos down: Farol vermelho, impede que outra
thread acesse o código
23             $LocalCopy = $GlobalVariable;
24             say("Faltam $TryCount tentativas para a
função $SubNumber." ,
25                "\$GlobalVariable é $GlobalVariable");
26             $LocalCopy++;
27             $GlobalVariable = $LocalCopy;
28             $semaphore->up();
29  #libera a execução para outra thread;
30             sleep 1;
31         }
32     }
33
34 #Executa as Threads
35     $thr1->join();
36     $thr2->join();
```

```
 37     $thr3->join();

saída:

Faltam 9 tentativas para a função 2. $GlobalVariable é 0
Faltam 9 tentativas para a função 1. $GlobalVariable é 1
Faltam 9 tentativas para a função 3. $GlobalVariable é 2
Faltam 8 tentativas para a função 2. $GlobalVariable é 3
Faltam 8 tentativas para a função 1. $GlobalVariable é 4
Faltam 8 tentativas para a função 3. $GlobalVariable é 5
Faltam 7 tentativas para a função 2. $GlobalVariable é 6
Faltam 7 tentativas para a função 1. $GlobalVariable é 7
Faltam 7 tentativas para a função 3. $GlobalVariable é 8
Faltam 6 tentativas para a função 2. $GlobalVariable é 9
Faltam 6 tentativas para a função 1. $GlobalVariable é 10
Faltam 6 tentativas para a função 3. $GlobalVariable é 11
Faltam 5 tentativas para a função 2. $GlobalVariable é 12
Faltam 5 tentativas para a função 1. $GlobalVariable é 13
Faltam 5 tentativas para a função 3. $GlobalVariable é 14
Faltam 4 tentativas para a função 2. $GlobalVariable é 15
Faltam 4 tentativas para a função 1. $GlobalVariable é 16
Faltam 4 tentativas para a função 3. $GlobalVariable é 17
Faltam 3 tentativas para a função 2. $GlobalVariable é 18
Faltam 3 tentativas para a função 1. $GlobalVariable é 19
Faltam 3 tentativas para a função 3. $GlobalVariable é 20
Faltam 2 tentativas para a função 2. $GlobalVariable é 21
Faltam 2 tentativas para a função 1. $GlobalVariable é 22
Faltam 2 tentativas para a função 3. $GlobalVariable é 23
Faltam 1 tentativas para a função 2. $GlobalVariable é 24
Faltam 1 tentativas para a função 1. $GlobalVariable é 25
Faltam 1 tentativas para a função 3. $GlobalVariable é 26
Faltam 0 tentativas para a função 2. $GlobalVariable é 27
Faltam 0 tentativas para a função 1. $GlobalVariable é 28
Faltam 0 tentativas para a função 3. $GlobalVariable é 29
```

Veja nesta saída que todas as threads puderam alterar a variável $GlobalVariable. Mas está alteração ocorreu de forma organizada, ou seja, apenas uma thread por vez alterou a variável, evitando situações em que uma thread altera o valor de uma variável enquanto a outra ainda faz uso dela.

17.6 O que ler agora?

Se você achou este assunto interessante e deseja desenvolver habilidades em programação com threads, aprofunde seus conhecimentos lendo:

1. O tutorial ***perldoc perlthrtut***;
2. "Uma introdução à programação com threads", disponível em ftp://ftp.dec.com/pub/DEC/SRC/research-reports/SRC-035.pdf;
3. "Where Wizards Fear To Tread", disponível em http://www.perl.com/pub/a/2002/06/11/threads.html;
4. Pro Perl, de Peter Wainwright, editora Apress, capítulo 21.

Exercícios

1. Altere o exemplo 17.6 para que o programa mostre o tempo total de execução. Utilize o módulo ***Benchmark***. Depois crie uma versão sem threads do mesmo programa e compare os tempos de execução.

2. Execute o exemplo 17.9 utilizando o módulo ***Benchmark*** para verificar os tempos de execução e compare com os tempos do exercício anterior.

3. Altere o exemplo 17.9 criando diversos objetos do tipo ***Thread::Queue*** e cada objeto com uma lista de hosts própria para verificar. Verifique os tempos de execução e tire suas próprias conclusões.

Capítulo 18. Programação Orientada a Objetos com Moose

Já vimos como trabalhar com objetos com o **Perl 5**. Porém, com o tempo, as linguagens OOP evoluíram e muitos programadores gostariam de ter no Perl algo semelhante com a forma de trabalho encontrada em outras linguagens. Por exemplo, algumas linguagens permitem que você diga: este método só funciona se os argumentos forem strings de texto. Ou apenas: faça com que este método seja privado. Eu quero esta variável estática (variável de classe). Você pode fazer tudo isso com o **Perl 5.12**, porém com um custo adicional de programação.

Assim, cansados deste custo adicional de programação, programadores espertos e dedicados começaram a estudar as melhores características de linguagens orientadas a objetos como Java, SmalTalk e Ruby e desenvolveram o **Moose**, que é uma extensão do sistema de Orientação a Objetos do Perl 5, unindo o que estes programadores inteligentes consideraram ótimas características destas linguagens no **Moose**. Para utilizar o **Moose** você primeiro precisa instalar o módulo no seu sistema. Você já sabe instalar módulos. O **Moose** está disponível via CPAN ou via pacotes nas principais distribuições Linux.

Para mostrar o ganho que o programador tem ao usar o **Moose** vamos recriar nosso exemplo Person, visto no capítulo de programação OOP:

```
1 #exemplo18.1
2 package Person;
3 use Moose;
4 1;
```

E o programa para usar o pacote Person:

```
  1 #exemplo18.2
  2 use 5.12.0;
  3 use lib(".");
  4 use Person;
  5
  6 my $pessoa = Person->new();
  7 say $pessoa;

saída:

Person=HASH(0x96860a0)
```

Viu só que legal? Nem precisamos criar o construtor. O próprio **Moose** cria um construtor para nós, que será sempre o método **new( )**. E não é só isso. Lembra aquelas expressões regulares complexas para verificar os argumentos enviados para os métodos? O **Moose** pode fazer isso para nós - pelo menos boa parte da verificação. Nós definimos os atributos e o tipo e o **Moose** verifica para nós:

```
  1 package Person;
  2 #exemplo18.3
  3 use Moose;
  4
  5 #name pode ser lido e modificado. Seu argumento é uma
string - Str
  6 has "name", is=>"rw", isa => "Str";
  7
  8 #Age é um Inteiro - Int. Valor padrão é 0
  9 has "age" => (  is=>"rw",
 10        isa => "Int",
 11        default => 0);
 12
 13 #É uma string. Valor padrão é single
 14 has "marital_state" =>( is => "rw",
 15              isa => "Str",
 16              default => 'single');
 17 1;
```

Aqui definimos os atributos com a função **has** que recebe um atributo - propriedade do objeto - se o atributo pode ser alterado (**rw)** ou apenas visto (**ro)**. Assim **is=>"rw"** diz: faça deste atributo de leitura e escrita. Também definimos se usaremos uma string (**Str)** ou um número, **Int**.

Veja um pequeno script que usa esse bloco de código em nossa nova versão e Person:

```
 1 #exemplo18.4
 2 use 5.12.0;
 3 use lib(".");
 4 use Person;
 5
 6 my $pessoa = Person->new(name => "João da Silva");
 7 say $pessoa->name(), " tem ", $pessoa->age(30) ,
 8     " anos de idade e é ", $pessoa->marital_state();

 saída:

João da Silva tem 30 anos de idade e é single
```

É notório que o Moose já gerou um verificador grosseiro para os atributos do meu objeto, de acordo com a definição da classe **isa => Str** ou **isa => Int.** A verificação do **Moose** é bem ampla e abrange os seguintes tipos:

```
Any
Item
     Bool
     Maybe['a]
     Undef
     Defined
           Value
                 Str
                     Num
                        Int
                 ClassName
                 RoleName
           Ref
```

```
            ScalarRef['a]
            ArrayRef['a]
            HashRef['a]
            CodeRef['a]
            RegexpRef
            GlobRef
                  FileHandle
            Object
```

Assim, o Moose pode verificar se o atributo recebe um escalar, uma referência, um filehandle etc. No diagrama anterior, **['a]** é qualquer tipo. Além de verificar se o atributo recebe uma referência a um array, o Moose pode verificar o conteúdo deste array:

```
ArrayRef[Str] ou ArrayRef[Int] ou ainda ArrayRef[Object].
```

Também informamos o valor **default** de um atributo de forma simples e clara. Aliás, o **Moose** deixa as coisas tão claras que é como se ele próprio fosse documentando a classe com sua sintaxe. A chave **default** informa o valor padrão a ser usado, caso o usuário não passe valor algum ao construtor. Mesmo o **Moose** realizando verificação de tipo, nossos métodos devem verificar se os argumentos têm valores dentro de parâmetros possíveis. Temos que evitar situações onde alguém possa informar uma idade de 3000 anos, por exemplo. Ou limitar os tipos de Str do marital para "married", "single" e "divorced". Assim, ninguém terá um estado civil estranho, como dvced ou mrried.

Para tanto, o Moose oferece mais uma chave para seus atributos, como **trigger**, que é uma referência a uma função. Esta função será evocada toda vez que o atributo é modificado. No exemplo18.5, se o programa que utiliza a classe Person evocar um acessor, a função em ***trigger*** é executada:

```
1 package Person;
2 #exemplo18.5
3 use Moose;
4
5 has "name" => ( is=>"rw",
6  isa => "Str",
```

```
  7  trigger =>sub{
  8  confess ("Nome com formato incorreto")
  9        if($_[1] =~/[!\@#\$\%"\&\*\(\)+=\{\}\]\
[,<>:;,\^]/);
 10  });#Nomes não contêm @ # ou outras coisas estranhas
 11
 12 has "age" => (is=>"rw",
 13  isa => "Int",
 14  default => 0,
 15  trigger =>sub{
 16        #O Moose traz confess do módulo Carp
 17        confess ("Idade Informada Indevida")
 18              if(not $_[1] >= 0 and $_[1] <= 150);
 19  });#Além de personagens bíblicos, ninguém até hoje
passou de 120 anos
 20
 21 has "marital_state" =>( is => "rw",
 22              isa => "Str",
 23        default => 'single',
 24        trigger => sub{
 25              confess ("Estado civil incorreto")
 26              if($_[1] !~ /^(married)|(single)|(divorc
ed)$/);
 27        });
 28 1
```

Este código equivale à maior parte da classe Person, estudada no capítulo 13. O código aqui é mais simples, mais legível e mais conciso. Veja um exemplo de código simples que utiliza nossa classe:

```
  1 use 5.12.0;
  2 #exemplo18.6
  3 use lib(".");
  4 use Person;
  5
  6 my $pessoa = Person->new();
  7 $pessoa->name("João Pessoa");
```

```
  8 say $pessoa->name( ), " tem ", $pessoa->age(30) , "
anos de idade ",
  9  "e é ", $pessoa->marital_state();
 10
 11 say $pessoa->name . " ficou mais velho: " . $pessoa-
->age($pessoa->age() + 1);
 12 say $pessoa->name . " casou e seu estado civil agora
é:" .
 13  $pessoa->marital_state("married");
 14 say $pessoa->name . " divorciou " . $pessoa->marital_
state("divrcado");
```

Foi introduzido um erro de digitação intencionalmente em

```
$pessoa->marital_state("divrcado")
```

Assim podemos ver a saída de erros gerada caso o argumento não esteja com a verificação correta:

```
João Pessoa tem 30 anos de idade e é single
João Pessoa ficou mais velho: 31
João Pessoa casou e seu estado civil agora é:married
Estado civil incorreto at Person.pm line 23
     Person::__ANON__('Person=HASH(0x883fee0)', 'divrcado',
'married') called at accessor marital_state defined at Per-
son.pm line 35
     Person::marital_state('Person=HASH(0x883fee0)', 'divr-
cado') called at /home/thiago/workspace/Moose/moose_intro.pl
line 14
```

A saída de erro informa qual método de Person recebeu o argumento incorreto, quais argumentos foram enviados para o método e a linha no script de origem.

Além disso, podemos criar nossos métodos para classe, como **get_older**. Métodos adicionais são criados facilmente, de forma semelhante ao que fizemos antes:

```
28
29 sub get_older{
30  my $self = @_;
31  $self->age($self->age + 1);
32 }
33
34 1
```

18.1 Relacionamento de herança e relacionamento "tem um"

Vamos ver como o **Moose** permite que a herança seja algo mais simples do que ela já é. E também vou apresentar um novo tipo de relacionamento entre objetos, o "**tem um".**

O relacionamento **tem um** difere do relacionamento de herança, também conhecido como **"é um"**. No capítulo 13, criamos as Classes **WonderWorldPerson** que é uma extensão de **Person**. Aqui vamos criar **a classe Ponto**, **a classe Ponto3D e a classe Reta.** Estas classes têm a seguinte relação: um objeto da classe Ponto3D será um ponto, pois herdará as características de Ponto e adicionará outras. A classe Reta será uma classe diferente, mas terá em seus atributos dois objetos ponto. Assim, um objeto da classe Reta terá objetos da classe Ponto em seus atributos, mas não será um objeto Ponto.

```
 1 #exemplo18.7
 2 package Ponto;
 3 use Moose;
 4
 5 has 'x' => (   isa => 'Int',
 6          is => 'rw',
 7          required => 1,
 8        ); #Cria coordenada x
 9
10 has 'y' => (   isa => 'Int',
11          is => 'rw',
```

```
12          required => 1,
13        );#Cria coordenada y
14
15 sub clear{
16  my $self = $_[0];
17  $self->x(0);
18  $self->y(0);
19 }
20 1;
```

```
 1 #exemplo18.8
 2 package Ponto3D;
 3 use Moose;
 4
 5 #Herda todas as características e funções de Ponto
 6 extends 'Ponto';
 7 has 'z' => (    isa => 'Int',
 8        is => 'rw',
 9        required => 1,
10        );#Cria coordenada z
11
12 #after informa que o método clear de Ponto3D deve ser
executado
13 #após (after) o método clear da classe pai, Ponto. Per-
ceba que
14 #o método clear de Ponto reseta as coordeandas x e y e
de Ponto3D reseta z
15
16 after 'clear' => sub{
17  my $self = $_[0];
18  $self->z(0);
19 };
20
21 #também podemos usar before, que executaria o método
clear
22 #do objeto filho (Ponto3D) e depois o método clear do
objeto Pai.
```

```
 23 #Também poderíamos simplesmente sobrescrever o método
da classe pai
 24 #inteiramente criando um método novo com override
 25 1;
```

```
  1 #exemplo18.9
  2 package Reta;
  3
  4 use Moose;
  5 #O construtor de Reta recebe um array de objetos do
tipo 'Ponto'.
  6 #Afinal, 2 pontos definem uma reta.
  7
  8 has pontos => ( isa => 'ArrayRef[Ponto]',
  9                   is => 'rw',
 10                   required => 1,
 11                   auto_deref => 1,
 12                );
 13 #auto_derefe transforma um ArrayRef em um Array dentro
do objeto.
 14
 15 #definimos alguns métodos para Reta
 16 sub distancia_y{
 17  my $self = shift;
 18  return abs($self->pontos->[1]->y - $self->pontos->[0]-
>y);
 19 }
 20
 21 sub distancia_x{
 22  my $self = shift;
 23  return abs($self->pontos->[1]->x - $self->pontos->[0]-
->x);
 24 }
 25
 26 sub tamanho{
 27  my $self = shift;
 28  return sqrt($self->distancia_x ** 2 + $self-
>distancia_y ** 2);
 29 }
 30 1;
```

```
  1 #exemplo18.10
  2 #Agora vamos usar nossos objetos:
  3 use 5.12.1;
  4 use lib ".";
  5 use Ponto;
  6 use Ponto3D;
  7 use Reta;
  8
  9 my $ponto = Ponto->new(x => -8, y => 15);
 10 printf "Coordenadas: x:%d y:%d\n", $ponto->x, $ponto-
->y;
 11 $ponto->clear( );
 12 printf "Novas coordenadas: x:%d y:%d\n", $ponto->x,
$ponto->y;
 13 printf "Um objeto feito com Moose é um HashRef: %s\n",
$ponto;
 14 $ponto = undef; #Objeto destruído
 15
 16 my $ponto3d = Ponto3D->new(x => 3, y => 5, z => 2);
 17 printf "Coordenadas de \$ponto3d: x:%d, y:%d, z:%d\
n",$ponto3d->x,
 18        $ponto3d->y, $ponto3d->z;
 19 $ponto3d->clear( );
 20 printf "Coordenadas de \$ponto3d após clear: x:%d,
y:%d, z:%d\n",
 21        $ponto3d->x, $ponto3d->y, $ponto3d->z;
 22 say "\$ponto3d é um objeto do tipo Ponto" if $ponto3d-
->isa("Ponto");
 23
 24 my $ponto1 = Ponto->new(x => 5, y => 5);
 25 my $ponto2 = Ponto->new(x => 10, y => 10);
 26
 27 my $reta = Reta->new( pontos => [$ponto1, $ponto2] );
 28
 29 say "A reta contém o objeto: ", $_ for $reta->pontos;
 30 say "Mas a reta não é um Ponto" if not $reta-
->isa("Ponto");
 31
```

```
 32 #Por meio do atributo que contém o objeto Ponto podemos
usar
 33 #todos os métodos do objeto Ponto e acessar as suas
propriedades
 34 say $reta->pontos->[0]->x, " ", $reta->pontos->[0]->y;
 35
 36 #Usamos alguns atributos exclusivos de Reta
 37 say $reta->distancia_y( );
 38 say $reta->distancia_x( );
 39 say $reta->tamanho( );
```

Veja a saída deste pequeno programa:

```
Coordenadas: x:-8 y:15
Novas coordenadas: x:0 y:0
Um objeto feito com Moose é um HashRef:
Ponto=HASH(0x8e60800)
Coordenadas de $ponto3d: x:3, y:5, z:2
Coordenadas de $ponto3d após clear: x:0, y:0, z:0
$ponto3d é um objeto do tipo Ponto
A reta contém o objeto: Ponto=HASH(0x9365d78)
A reta contém o objeto: Ponto=HASH(0x9365eb8)
Mas a reta não é um Ponto
5 5
5
5
7.07106781186548
```

Neste último programa temos uma Classe Ponto, que é a classe Pai de Ponto2D e é usada como atributo da classe Reta. Assim, a Classe Reta se relaciona como "**tem um"** com a classe Ponto. Além destes, a classe Reta definiu métodos como **tamanho,** que calcula o tamanho da reta de acordo com as coordenadas dos pontos, e **distancia_x**, que calcula a distância entre os pontos que definem a reta em relação ao eixo **x**. O mesmo serve para **distancia_y**.

18.2 O que ler agora?

Não existem livros de referência para o Moose; assim, o melhor material para referência e estudos, caso você precise de mais poder em suas aplicações (e conforme elas se tornarem maiores você vai precisar), está nas páginas de manual e no site http://www.iinteractive.com/moose/.

O **Moose** tem sido bem aceito pela comunidade e promete ser um módulo que será usado e mantido por muito tempo.

Exercícios

1. Crie versões dos exemplos do capítulo 13 utilizando o Moose.
2. Refaça os exercícios do capítulo 13 com o Moose.

Capítulo 19. Gerenciando Elementos de Rede com o Perl

Quando dizemos administração de redes muitas coisas vêm a mente, como administração de roteadores, switches, servidores, serviços, pontos de acesso sem fio, enlaces de rádio, modems. E cada item tem sua peculiaridade. As interfaces de um roteador Cisco são diferentes das interfaces de um roteador Enterasys. As características de manutenção de um servidor web com Apache são diferentes das características de servidor web com IIS. Diversas empresas tentam desenvolver produtos que sejam abrangentes o bastante para cobrir todas estas diferenças. Apesar de todos os esforços, apenas a criação de protocolos comuns como SNMP e Syslog permitem soluções abrangentes, porém elas perdem a particularidade e por vezes seu cliente pode ter uma situação tão inusitada que somente a união da criatividade com a técnica resolvem o problema.

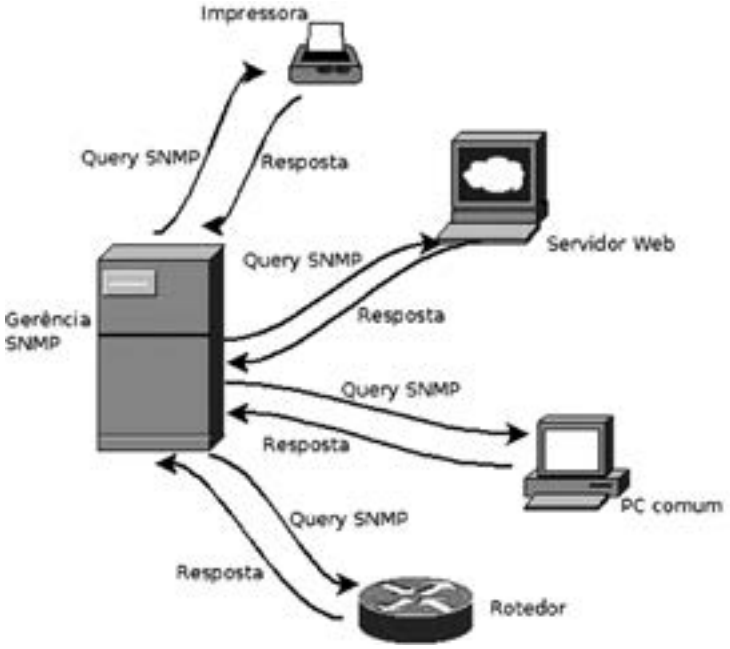
Recentemente eu me deparei com a necessidade de monitorar e configurar mais de 100 modems GSM que estavam conectados a um sistema de telesupervisão. O sistema de telesupervisão gerenciava a integridade de dutos de transporte de derivados de petróleo. A minha missão foi criar um sistema que verifique a disponibilidade e configuração correta dos modems de tempos em tempos. Infelizmente não posso apresentar a solução em seus detalhes, mas ela foi feita cem por cento em Perl, com os módulos ***Net::Ping*** e ***Net::Telnet***.

Apesar das disparidades, são desafios comuns a todo administrador de redes:

- Fazer backups regularmente, seja de arquivos de configuração, seja de dados do cliente;
- Garantir a disponibilidade dos elementos e serviços da rede;
- Garantir a segurança e a conformidade das configurações conforme os padrões da empresa e normas internacionais, como ISO 27002 e SOX (mais sobre SOX em http://pt.wikipedia.org/wiki/Lei_Sarbanes-Oxley).

19.1 Entendendo o SNMP

Para administrar uma rede com o SNMP é necessário um Sistema de Gerência, um Elemento Gerenciável e uma MIB.



O Sistema de Gerência é um software capaz de gerar requisições SNMP via protocolo UDP porta 161. Para obter a resposta do elemento, a requisição deve estar em conformidade com as especificações da MIB do Elemento.

O protocolo SNMP também especifica o uso de traps na porta UDP 162, mas estas não serão discutidas aqui. Para criar um sistema de gerência que receba traps, seria necessário configurar um servidor que escute a porta 162.

Mas isto não gera a polêmica em torno do SNMP, e sim a apresentação da MIB II e a grande quantidade de informação que é possível de ser obtida. A maioria dos administradores utiliza apenas de recursos para verificar se o elemento está up/down e o estado das interfaces de rede dos elementos. Mas podemos obter uma gama imensa de dados sobre dispositivos de rede, sistemas operacionais e softwares instalados nos computadores com o SNMP.

19.1.1 O que você deve saber sobre MIBs

O código a seguir não é um script Perl, mas a representação do objeto 1.3.6.1.2.1.1.3 – ou **iso.org.dod.internet.mgmt.mib-2.system.sysUpTime** – de um arquivo MIB II. Este objeto permite que obtenhamos o tempo em que um sistema está operacional, ou seja, o SysUpTime de um elemento.

```
1 sysUpTime OBJECT-TYPE
2 SYNTAX Time Ticks
3 MAX- ACCESS Read Only
4 STATUS Current
5 DESCRIPTION
6  The time (in hundredths of a second) since the
7  network management portion of the system was last
8  re-initialized.
9  ::= { system 3 }
10
```

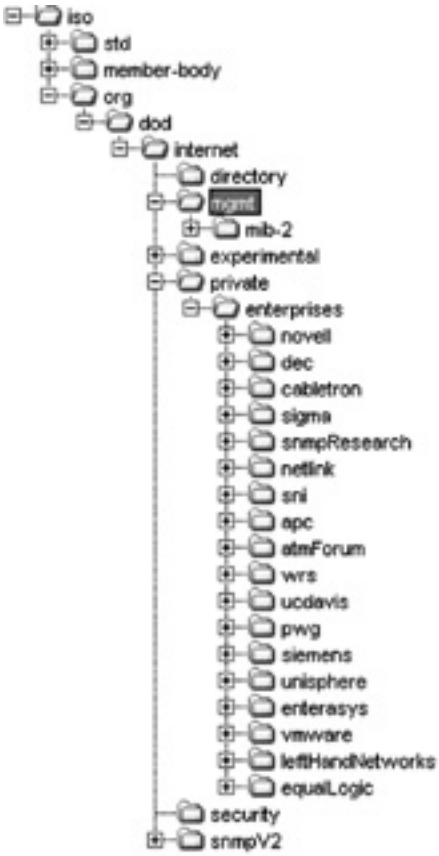
As MIBs são escritas em ASN.1 (Abstract Syntax Notation 1). O ASN.1 é mantido pela ISO e utilizado para as mais diversas finalidades, desde descrever Controle Aéreo até a sua placa de rede. O ASN.1 é uma especificação muito extensa e queremos aqui passar apenas o necessário para melhorar sua habilidade de administrador de uma rede.

Observe que o documento é estruturado. A seção DESCRIPTION, na linha 5, é a mais interessante para os administradores, pois ela diz o que este objeto faz. Neste caso, como já dissemos, ela retorna o tempo, em centésimos de segundos, desde a última inicialização do sistema. Na terceira linha temos o MAX-ACCESS. Este define a ação que o administrador pode tomar com o objeto. Pode ser "read-only", "read-write", "read-create", "not-accessible" e "accessible-for-notify". A segunda linha apresenta a SYNTAX, que nada mais é que o tipo de dados que recebemos de um objeto ou devemos usar para configurá-lo. A lista de tipos é extensa, mas podemos destacar como principais os descritos na tabela:

Tipo de Dado	Descrição
OBJECT IDENTIFIER	Notação formada por uma sequência de números decimais separados por ponto que representa o objeto dentro da árvore de objetos. No nosso exemplo o objetct-type é 1.3.6.1.2.1.1.3, traduzido para sysUpTime.
INTEGER	Valor numérico inteiro que pode ser utilizado para representar estados (0 = "down", 1= "up", 3 = "dormant") ou valores quantitativos como número de bytes transmitidos.

OCTET STRING	String de bytes. Pode ser utilizada para representar, por exemplo, caracteres ASCII.
IpAddress e NetworkAddress	Representação de endereços IPv4. As versões 1 e 2 do SNMP não suportam o IPv6.
Counter e Gauge	São contadores. A diferença é que o Gauge aceita valores crescentes e decrescentes, enquanto o Counter aceita apenas valores crescentes.
TimeTicks	Utilizado para medir tempo.
"Sequence" e "Sequence of"	Sequence é formado por uma sequência dos tipos de dados definidos pela ASN.1. "Sequence of" representa uma sequência de objetos.

O OBJECT IDENTIFIER ou OID é uma representação única de um elemento dentro da árvore de objetos gerenciáveis do elemento. A figura a seguir representa uma árvore de objetos de um arquivo MIB:



Se observarmos o início de um arquivo MIB, vamos ter as seguintes informações:

```
SNMPv2-SMI DEFINITIONS ::= BEGIN

-- the path to the root

org OBJECT IDENTIFIER ::= { iso 3 } -- "iso" = 1
dod OBJECT IDENTIFIER ::= { org 6 }
internet OBJECT IDENTIFIER ::= { dod 1 }

directory OBJECT IDENTIFIER ::= { internet 1 }

mgmt OBJECT IDENTIFIER ::= { internet 2 }
mib-2 OBJECT IDENTIFIER ::= { mgmt 1 }
transmission OBJECT IDENTIFIER ::= { mib-2 10 }
experimental OBJECT IDENTIFIER ::= { internet 3 }

private OBJECT IDENTIFIER ::= { internet 4 }
enterprises OBJECT IDENTIFIER ::= { private 1 }
security OBJECT IDENTIFIER ::= { internet 5 }
snmpV2 OBJECT IDENTIFIER ::= { internet 6 }
```

Observando o arquivo e a imagem gerada por um software gráfico para leitura de MIBs, vemos que "ISO" é a raiz da árvore. Seu OID é 1. ORG é definido como {iso 3}. Então ele é o terceiro elemento da árvore, dentro de ISO. Seu OID é 1.3. Assim vão se formando os OIDs dentro da árvore. A mib-2 é definida como {mgmt 1}. Então, mib-2 é 1.3.6.1.2.1.

Para nós, administradores de rede, os objetos mais interessantes estão dentro de **mib-2** e **private**. MIB-2 contém objetos que nos permitem gerenciar características que são comuns a diversos equipamentos e servidores de rede. Os objetos dentro de "PRIVATE" permitem gerenciar características específicas definidas por fabricantes. Na figura a seguir vemos os objetos dentro de ISO.org.dod.mgmt.mib-2.ospf, que nos permitem gerenciar as características do protocolo de roteamento OSPF em roteadores de qualquer fabricante que tenha implementado a gerência via SNMP.

19.1.2 E como eu faço para minha rede responder ao protocolo SNMP?

Os elementos que você deseja gerenciar com SNMP devem ter um daemon ou serviço que responda às queries enviadas pelo software de gerência. Elementos de rede como switches, impressoras e roteadores precisam ter o SNMP implementado no firmware e habilitado via configuração. Elementos de fabricantes diferentes possuem formas diferenciadas para habilitar o agente SNMP, portanto não irei abordar isto aqui. Veja as opções de configuração SNMP em um roteador Cisco:

```
RTFrodo#conf t
Enter configuration commands, one per line.  End with
CNTL/Z.
RTFrodo(config)#snmp-server ?
```

```
  chassis-id        String to uniquely identify this chas-
sis
  community         Enable SNMP; set community string and
access privs
  contact           Text for mib object sysContact
  context           Create/Delete a context apart from de-
fault
  drop              Silently drop SNMP packets
  enable            Enable SNMP Traps
  engineID          Configure a local or remote SNMPv3 en-
gineID
  file-transfer     File transfer related commands
  group             Define a User Security Model group
  host              Specify hosts to receive SNMP notifica-
tions
  ifindex           Enable ifindex persistence
  inform            Configure SNMP Informs options
  location          Text for mib object sysLocation
  manager           Modify SNMP manager parameters
  packetsize        Largest SNMP packet size
  queue-length      Message queue length for each TRAP host
  source-interface  Assign an source interface
  system-shutdown   Enable use of the SNMP reload command
  tftp-server-list  Limit TFTP servers used via SNMP
  trap              SNMP trap options
  trap-source       Assign an interface for the source
address of all traps
  trap-timeout      Set timeout for TRAP message retrans-
missions
  user              Define a user who can access the SNMP
engine
  view              Define an SNMP MIB view

RTFrodo(config)#snmp-server
```

E um switch Enterasys:

```
M A T R I X   N 7   P L A T I N U M
Command Line Interface
```

```
Enterasys Networks, Inc.
50 Minuteman Rd.
Andover, MA 01810-1008 USA

Phone:  +1 978 684 1000
E-mail: support@enterasys.com
WWW:    http://www.enterasys.com

(c) Copyright Enterasys Networks, Inc. 2009

SWFrodo(rw)->set snmp ?
  access                  SNMP VACM access configuration
  community               SNMP v1/v2c community name config-
uration
  group                   SNMP VACM security to group con-
figuration
  notify                  SNMP notify configuration
  notifyfilter            SNMP notify filter configuration
  notifyprofile           SNMP notify profile configuration
  targetaddr              SNMP target address configuration
  targetparams            SNMP target parameters configura-
tion
  timefilter              SNMP timefilter configuration
  user                    SNMP USM user configuration
  view                    SNMP VACM view tree configuration
```

A primeira escolha a ser feita para iniciar a gerência de uma rede com SNMP é o que será gerenciado. Quais elementos da sua rede – roteadores, switches, servidores, no-breaks ou máquinas de usuários – e quais objetos da MIB serão monitorados ou configurados – estado da interface, temperatura, softwares instalados, tensão na fonte? Esta escolha depende do tipo de informação, disponibilidade e equipamentos sob sua responsabilidade.

A segunda escolha será como gerenciar os elementos. Irei usar "traps" ou enviar comandos "get"? Ou ambos? Se quisermos informações de mudanças na rede que necessitamos conhecer no momento do evento, é interessante utilizarmos um "trap" ou um "notification" (ver adiante). Se eu desejo verificar que as configurações estão de acordo com um padrão, eu posso usar um "get".

Se eu desejo poder configurar meus elementos via SNMP, posso usar o "set". Dentro desta escolha temos mais uma pergunta: existe um software no mercado

que contemple minha necessidade ou terei de criar esta solução? Tenho verba para adquiri-lo ou será melhor desenvolver uma solução caseira e economizar alguns dólares, tão raros nos anos de crise? Existe o sistema gratuito Net-SNMP, em http://net-snmp.sourceforge.net. É uma opção gratuita para gerenciar SNMP que oferece bibliotecas para desenvolvimento em C, Perl e Python.

A terceira escolha diz respeito à versão do SNMP que iremos adotar: SNMPv1, SNMPv2 ou SNMPv3. O SNMPv3 é o único que tem autenticação de usuário criptografada. O SNMPv1 e SNMPv2 enviam a "community" de autenticação em texto puro pela rede, o que é considerado inseguro, pois pode ser interceptada por um equipamento promíscuo na rede. Community, em SNMP, é uma senha de autenticação. O SNMPv2 possui mais opções de comando em relação ao SNMPv1. A tabela a seguir apresenta os comandos que podem ser executados por cada versão do SNMP:

Comando	**Versão**	**O que faz?**
get	todas	Retorna o valor de um objeto no elemento de rede.
getnext	todas	Permite obter os OIDs dentro de um nível da arquitetura. É utilizado juntamente com o getresponse.
getresponse	todas	Realiza um get no OID definido pelo getnext.
trap	todas	O trap é enviado do elemento da rede para o sistema de gerência a fim de informar mudanças em algum objeto monitorado.
set	todas	Permite editar valores em propriedades gerenciadas via SNMP.
getbulk	SNMPv2 e SNMPv3	Permite receber todos os valores de um objeto que representa uma tabela. Um exemplo deste tipo de objeto é iso. org.dod. internet.mgmt.mib-2.interfaces.ifTable.
inform	SNMPv2 e SNMPv3	Funcionamento idêntico ao trap.
notification	SNMPv2 e SNMPv3	Funcionamento idêntico ao trap, porém gera uma resposta do software da gerência para o elemento a fim de confirmar o recebimento da notificação.
report	SNMPv2 e SNMPv3	Não implementada.

Apesar da melhor segurança oferecida pelo SNMPv3, muitas implementações SNMP em uso hoje utilizam a versão 2 do protocolo.

19.2 Criando uma solução SNMP personalizada com o Perl

Você pode decidir criar um gerenciador SNMP para atender a uma necessidade em particular. No meu caso, softwares gerenciadores de fabricantes específicos preferem usar os objetos dentro de iso.org.dod. internet.private e, como já dissemos, este espaço é reservado para características particulares dos fabricantes. Logo, um gerenciador da Cisco não funciona perfeitamente com elementos da Siemens e vice-versa. Softwares que ofereçam gerência para elementos de diversos fabricantes podem ser muito caros – pelo menos para meus clientes – e ainda necessitariam da criação de alguns módulos para atender a todas as necessidades de uma grande empresa.

Eu decidi utilizar o Perl nestas tarefas pelos seguintes motivos:

- É realmente uma linguagem multiplataforma. Inclusive os módulos para criar programas de rede funcionam em Linux, Solaris ou Windows;
- Fácil de usar e atingiu seu desafio de manter as coisas simples simples e as coisas difíceis, possíveis;
- O CPAN – repositório do Perl em www.cpan.org – torna as coisas difíceis simples. São milhares de módulos gratuitos e com o código aberto;
- É uma linguagem gratuita e open source.

O Perl já vem instalado em máquinas GNU/Linux padrão. Porém, para criar uma solução SNMP, precisamos instalar o ***Net::SNMP***. Você o encontra no repositório CPAN em http://search.cpan.org/~dtown/Net-SNMP-v6.0.0/lib/Net/SNMP.pm. No CPAN existem diversos módulos para gerenciamento via SNMP. Eu gosto particularmente deste porque é um módulo sem dependências externas. Não temos que instalar nada na máquina que irá executar o programa. Se ela for capaz de rodar o Perl, ela poderá usar o Net::SNMP e a nossa aplicação.

Depois de tanto blá-blá-blá estamos ansiosos para começar a codificar. Uma vez que você baixou e instalou o módulo Net::SNMP, você está pronto para começar.

Como programadores organizados e por gostarmos de evitar retrabalho, vamos começar com um módulo genérico que obtém as informações de interface do elemento.

```
1 package SNMPRouter;
2 use strict;
3 use warnings;
4 use Net::SNMP;
5 use Carp qw/croak/;
```

Agora que definimos um nome para nosso pacote e adicionamos as diretivas comuns, vamos codificar o construtor do objeto:

```
6 sub new{
7
8
9 my ($class, $ip, $community) = @_;
10
11 my $self = {
12 IP=>$ip, # Armazena o IP/nome do elemento
13 Session => undef, # Armazena o Objeto SNMP gerado por
connect
14 Community => $community, # Armazena a community de segu-
rança
15 SysName => undef, #
16 SysDescription => undef, #
17 NumberOfInterfaces => undef, # Armazena o número de In-
terfaces
18 InterfaceIndexNames => {}, # Relaciona o número da in-
terface com o nome
19 InterfaceIPAddress => {}, # Relaciona o número da inter-
face com o IP
20 InterfaceAlias => {}, # Relaciona o número da interface
com o Alias
21 InterfaceAdminStatus => {}, #
22 InterfaceOperStatus => {}, #
23
24 };
25
26 bless $self, $class
27 }
```

O nosso objeto será basicamente um hash que irá armazenar as informações estratégicas das interfaces de um elemento de rede. A chave ***Session*** do hashref ***$self*** irá armazenar um objeto ***Net::SNMP***.

```
28 sub connect{
29 my $self = shift;
30 $self->{Session} = Net::SNMP->session(Hostname => $self-
->{IP},
31 Community => $self->{Community},
32 ) or croak "Unable to create SNMP Session\n";
33 }
```

No método ***connect***, criamos um objeto SNMP e criamos a sessão com o elemento definido em IP, que foi passado como argumento ao construtor juntamente com a community. Com isto podemos solicitar ou escrever as primeiras informações do elemento via SNMP:

```
34 sub query_system_info{
35 my $self = shift;
36 my $mib2System = '1.3.6.1.2.1.1';
37
38 my %mibSystemTable = (
39 Mib2SysName => $mib2System . '.5',
40 Mib2SysDescription => $mib2System . '.1',
41 );
42
43 my @queryList = values %mibSystemTable;
44
45 my $result = $self->{Session}->get_next_
request(Varbindlist => \@queryList);
46 croak "Erro ao tentar receber informações da MIB
System\n" unless
47 defined $result;
48
49 foreach my $key(keys %$result){
50 $self->{SysName} = $result->{$key} if
```

```
51 $key =~ /^$mibSystemTable{Mib2SysName}/;
52 $self->{SysDescription} = $result->{$key} if
53 $key =~ /^$mibSystemTable{Mib2SysDescription}/;
54 }
55 }
```

Observe que o método query_info_system obtém as informações dos objetos **iso.org.dod.internet.mgmt.mib-2.system.name** e **system.description**. Pode parecer um código um pouco longo e pouco prático para obter apenas duas informações, mas ele é bastante flexível. Se você quiser obter outros dados dentro do objeto **system**, como **uptime** ou **services**, basta adicionar mais chaves no hash ***%mibSystemTable*** e expressões regulares dentro do loop foreach para associar com a propriedade do objeto.

Agora vamos montar a tabela de interfaces. Como elementos de rede podem ter diferentes números de interfaces de diferentes tipos, nosso método primará pela flexibilidade do programa. Usamos as informações dos objetos **mib-2.interfaces.ifNumber,** que fornecem o número de interfaces de rede do elemento – tanto físicas quanto lógicas – e **mib-2.ifMIB**. Usamos estes dados para montar uma tabela hash que relaciona o índice da interface com o nome dela. Esta tabela ficará armazenada na propriedade ***InterfaceIndexName*** do objeto ***SNMPRouter.***

Os nomes de interface são, por exemplo, em um servidor Linux, eth0 e lo, gi5/3 ou se0/1 num roteador CISCO e fe.1.2 ou ge.1.24 num switch Enterasys. Esta é a beleza do SNMP com o Perl, não importam as diferenças entre os fabricantes, a raça, a cor ou o sexo. Podemos obter as informações e formatá-las ao nosso gosto.

```
56 sub create_int_table{
57 my $self = shift;
58 my $mib2ifNumber = '1.3.6.1.2.1.2.1';
59 my $mib2ifIndex = '1.3.6.1.2.1.2.2.1.1';
60 my $mib2ifXMIB = '1.3.6.1.2.1.31.1.1.1';
61 my $mib2ifName = $mib2ifXMIB . '.1';
62
63 my $ifNumber = $self->{Session}->get_next_request(
64 Varbindlist => [$mib2ifNumber]);
65
66 $ifNumber = $ifNumber->{"$mib2ifNumber.0"};
```

```
67
68 for(my $i = 1;$i <= $ifNumber; $i++){
69 my $ifIndex = $self->{Session}->get_request(
70 Varbindlist => ["$mib2ifIndex.$i"]);
71
72 if($ifIndex){
73 my $ifName = $self->{Session}->get_request(
74 Varbindlist => ["$mib2ifName.$i"]);
75
76 $ifName = $ifName->{"$mib2ifName.$i"};
77 $self->{InterfaceIndexNames}->{$i} = $ifName;
78 }else{$ifNumber++}
79 }
80 }
```

Muito bem. Já conseguimos os nomes das interfaces, o nome e a descrição do elemento. Agora vamos obter os estados e a descrição da interfaces. Afinal, é importante saber quais são as interfaces dos elementos, mas também é muito importante saber se elas estão ativas e respondendo. O hash ***%statusTable*** contém os valores retornados pelos objetos **mib-2.interfaces.ifTable.ifEntry.ifAdminStatus** e **ifOperStatus** relacionados com seu significado. Se você observar um arquivo de definição das MIBSs verá que a sintaxe dela é **INTEGER**. Este valor deve ser convertido para um texto com significado.

```
81 sub query_int_status{
82 my $self = shift;
83
84 croak "Tabela de interfaces não foi criada para este ob-
jeto"
85 unless $self->{InterfaceIndexNames};
86
87 my $mib2IfAdminStatus = '1.3.6.1.2.1.2.2.1.7';
88 my $mib2IfOperStatus = '1.3.6.1.2.1.2.2.1.8';
89 my $mib2ifAlias = '1.3.6.1.2.1.31.1.1.1.18';
90 my %statusTable = (
91 1=>'up',
92 2=>'down',
```

```
93 3=>'testing',
94 4=>'unknow',
95 5=>'dormant',
96 6=>'notPresent',
97 8=>'lowerLayerDown',
98 );
99
100 foreach my $key (keys %{$self->{InterfaceIndexNames}}){
101 my $status = $self->{Session}->get_request(
102 Varbindlist => ["$mib2IfOperStatus.$key",
103 "$mib2IfAdminStatus.$key",
104 "$mib2ifAlias.$key"
105 ]);
106 $self->{InterfaceOperStatus}->{$key} =
107 $statusTable{$status->{"$mib2IfOperStatus.$key"}};
108 $self->{InterfaceAdminStatus}->{$key} =
109 $statusTable{$status->{"$mib2IfAdminStatus.$key"}};
110 $self->{InterfaceAlias}->{$key} = $status-
->{"$mib2ifAlias.$key"};
111 }
112 }
```

Falta apenas o método que obtenha os endereços IPs das interfaces de rede. Este é um pouco mais confuso pois, ao contrário dos outros objetos, **mib-2.ip.ipAddrTable.ipAddrEntry.ipAdEntIfIndex** não relaciona o número da interface com o IP e sim o contrário. O IP é a referência do OID.

```
113 sub query_int_ip{
114 my $self = shift;
115 my $mib2IpAddrTable = '1.3.6.1.2.1.4.20';
116 my $mib2IpAddrxIntIndex = '1.3.6.1.2.1.4.20.1.2';
117 my $hshIpTable = $self->{Session}->get_table(Baseoid=>$
mib2IpAddrTable)
118 or croak "Erro ao tentar acessar a mib IpAddrTable";
119
120 foreach my $key (keys %{$hshIpTable}){
121 if ($key =~ $mib2IpAddrxIntIndex){
```

```
122 $key =~ s/$mib2IpAddrxIntIndex\.//;
123 my $intIndex = $hshIpTable->{$mib2IpAddrxIntIndex . ".”
. $key};
124 $self->{InterfaceIPAddress}->{$intIndex} = $key;
125
126 }
127 }
128 }
```

E, para terminar o módulo, não se esqueça de finalizar com 1.

```
129 1
```

Agora que você sente que teve um trabalhão para criar este módulo, vamos fazer um pequeno script de teste:

```
1  use strict;
2  use warnings;
3  use lib qw/./;
4
5  use SNMPRouter;
6
7  print “Digite o HOSTNAME ou IP do ROTEADOR: “;
8  chomp(my $host = <STDIN>);
9  my $community = ‘public’; #Altere este valor para a sua
community ou
10                           #solicite ao usuário para in-
formar.
11 my $router = SNMPRouter->new($host,$community);
12 $router->connect();
13 $router->query_system_info();
14
15 print “Informações do elemento:\n”,”#”x80,”\n”,
16      $router->{SysDescription},”\n”,’#’x80,”\n\n\n”;
17
```

```
18 print "\tElemento ",$router->{SysName},"\tIP: ",$router-
>{IP}, "\n";
19 print "#"x80,"\n\n";
20
21 $router->create_int_table();
22 $router->query_int_status;
23 $router->query_int_ip();
24
25 foreach (sort{$a<=>$b} keys %{$router-
>{InterfaceIndexNames}}){
26    print  "$router->{InterfaceIndexNames}->{$_}\t",
27           "$router->{InterfaceOperStatus}->{$_}\t",
28           "$router->{InterfaceAdminStatus}->{$_}\t",
29           ($router->{InterfaceIPAddress}->{$_}||"\t"),
30           "\t",($router->{InterfaceAlias}->{$_}||"\t"),
31           "\n";
32 };
33 print '#'x80,"\n\nPressione ENTER para sair\n";
34 <STDIN>
```

Ao executar este script em diferentes elementos de rede, teremos saídas um pouco diferentes. Por exemplo, um roteador Cisco gera uma saída parecida com esta:

```
Digite o HOSTNAME ou IP do ROTEADOR: 10.10.10.1

Informações do elemento:

################################################################
#

Cisco Internetwork Operating System Software

IOS (tm) C2600 Software (C2600-JS-M), Version 12.2(13)T4,
RELEASE SOFTWARE (fc2)

TAC Support: http://www.cisco.com/tac
```

```
Copyright (c) 1986-2003 by cisco Systems, Inc.

Compiled Thu 01-May-03 03:53 by eaarma

##########################################################
#

      Elemento RT01FRODO      IP: 10.10.10.1

##########################################################
#

Fa0/0 up up 10.10.11.254 REDE LOCAL

Se0/0 down down RESERVADA

Fa0/1 up up 10.10.12.254 REDE LOCAL

Se0/1 up up 10.10.13.254 REDE EXTERNA

Nu0 up up

Lo100 up up 10.10.14.100 INTERFACE ADMINISTRATIVA

VoiceOverIpPeer108 up up *** Para ligacoes externas ***

VoiceOverIpPeer100 up up *** Para ligacoes DDD ***

VoiceEncapPeer6085 up up *** RAMAL do JOAO ***

VoiceEncapPeer6086 up up *** RAMAL da ANA ***

VoiceEncapPeer6087 up up *** RAMAL do CARLOS ***

VoiceEncapPeer6084 up up

Vi1 up up
```

```
Foreign Exchange Station 1/0/0 dormant up

Foreign Exchange Station 1/0/1 dormant up

Foreign Exchange Station 1/0/2 dormant up

##########################################################
#

Pressione ENTER para sair
```

Se você tiver familiaridade com roteadores Cisco, perceberá que a saída é semelhante à do comando "show interface description". Apesar de otimizado para obter informações de roteadores, como eu disse, o Perl e o SNMP não são preconceituosos. Vamos agora mostrar o que acontece se eu tentar obter informações de um switch de uma concorrente da Cisco, a Enterasys:

```
Digite o HOSTNAME ou IP do ROTEADOR: 10.10.11.1

Informações do elemento:

##############################################

Enterasys Networks, Inc. A2H124-24 Rev 02.01.07.0003

##############################################

Elemento SW01SkyWalker IP: 10.10.11.1

##############################################

fe.1.1 up up

fe.1.2 down up

fe.1.3 down up

fe.1.4 down up

fe.1.5 down up
```

```
fe.1.6 down up
fe.1.7 down up
fe.1.8 down up
fe.1.9 down up
fe.1.10 down up
fe.1.11 down up
fe.1.12 up up
fe.1.13 up up
fe.1.14 down up
fe.1.15 up up
fe.1.16 down up
fe.1.17 up up
fe.1.18 down up
fe.1.19 up up
fe.1.20 down up
fe.1.21 up up
fe.1.22 up up
fe.1.23 up up
fe.1.24 down up
ge.1.25 up up
ge.1.26 up up
ge.1.27 down up
```

```
ge.1.28 down up

host up up 10.10.11.1

lag.0.1 down up

lag.0.2 down up

lag.0.3 down up

lag.0.4 down up

lag.0.5 down up

lag.0.6 down up

##############################################

Pressione ENTER para sair
```

O protocolo SNMP pode ser utilizado para gerenciar qualquer sistema que o suporte. Servidores Windows e Linux também suportam este protocolo, além de impressoras de rede e diversos outros equipamentos.

19.3 Telnet e SSH

SNMP rocks. Em teoria, todas as tarefas de administração de uma rede podem ser feitas via SNMP. Mas nem sempre este é o caminho mais fácil de se solucionar um problema, ou, pior, nem sempre o equipamento suporta SNMP. Muita coisa pode ser feita com uma automação de **Telnet** ou **SSH**. Os modems GSM dos quais eu falei no início do capítulo aceitam apenas conexão telnet e respondem a requisições de ping.

O CPAN nos fornece os módulos **Net::Telnet** e **Net::SSH** para que possamos automatizar tarefas de configuração, verificação e backup de elementos e servidores de rede.

O **Net::Telnet,** assim como o **Net::SSH,** abstrai todo o código necessário para embutir um cliente telnet no seu aplicativo. Assim, você só precisa se preocupar com quais comandos deve enviar para os elementos e quais as possíveis respostas dos elementos.

A documentação do módulo é bem completa e começa com o seguinte exemplo:

```
use Net::Telnet ( );
$t = new Net::Telnet (Timeout => 10,
                              Prompt => '/bash\$ $/');
$t->open("sparky");
$t->login($username, $passwd);
@lines = $t->cmd("who");
print @lines;
```

Este exemplo é bem simples e ilustra alguns pontos-chave de uma conexão telnet com um servidor Linux. Primeiro, o módulo é evocado. Uma nova sessão, ***$t***, é criada. Ela é um objeto ***Net::Telnet***, com os parâmetros ***Timeout***, que significa quanto tempo o cliente deve esperar por uma resposta antes de considerar que a comunicação não ocorreu ou foi interrompida. O parâmetro ***Prompt*** espera uma regex que contenha as possibilidades de prompt do seu elemento de rede. O método ***open*** inicia a sessão com o **host** informado. E o método ***cmd*** envia comandos para o elemento.

Apesar de extremamente simples, muita coisa poderosa pode ser feita com ***Net::Telnet*** ou ***Net::SSH***. Imagine, por exemplo, fazer o backup de algumas centenas de roteadores e switches Cisco com o seguinte script:

```
  1 #exemplo19.1
  2 use 5.12.0;
  3 use Net::Telnet;
  4
  5 #solicita informações de login
  6 my @access = login( );
  7
  8 #Como vamos efetuar o backup de algumas centenas de
elementos é
  9 # interessante criarmos um loop:Os IPs e alias dos ele-
mentos estão
 10 #armazenados no filehandle <DATA>
 11 while (my $host = <DATA>){
 12  chomp $host;
 13  my @ip = split (/\s+/, $host);
```

```
 14  #Será criada uma sessão para cada host
 15  my $t = Net::Telnet->new(
 16          Host => $ip[0], #IP do host
 17                Input_log => "$ip[1].txt", #arquivo de
log
 18          Errmode => "return", #Prossegue se houver
erro.
 19          Timeout => 60, #60 segundos, a rede tem ele-
mentos lentos.
 20          Prompt => '/# ?/', #Prompt rw da cisco
 21          );
 22
 23  #Se houve falhas para criar o telnet:
 24  if (!$t){
 25      print "Could not telnet $host\n";
 26              next
 27  }
 28
 29  #Envia usuário e senha
 30  $t->login(@access);
 31
 32  #Se ocorrer falha na sessão veremos aqui
 33  #(usuário pode ser inválido, ou podemos perder a cone-
xão com o elemento.
 34  if ($t->errmsg( )){
 35        warn ($t->errmsg( ) . "\n");
 36      $t->close( );
 37      next
 38  }
 39
 40 #Enviamos os comandos para o switch
 41 $t->cmd(String => "copy run tftp://<IP_DO_SERVIDOR_
TFTP>",
 42   Prompt => '/\?/');
 43 #Mudamos o prompt para ? pois o roteador irá perguntar
se quer usar o nome
 44 #de arquivo padrão
 45 $t->cmd(String => "\n", Prompt => '/\?/');
```

```
46 $t->cmd(String => "\n", Prompt => '/# ?/');
47
48 #encerramos a sessão atual
49 $t->close( );
50
51 }
52
53 #sub-rotina para solicitar informações de login ao usu-
ário.
54 sub login{
55     say "Digite o usuário: ";
56     chomp (my $user = <STDIN>);
57     say "Digite a senha:";
58     chomp (my $passwd = <STDIN>);
59     return ($user, $passwd);
60 }
61
62 #lista de elementos.
63 __DATA__
64 <IP>   <HOSTNAME>
```

Depois de efetuar o backup de centenas de elementos devemos ler os arquivos de log para verificar se algum elemento não concluiu o backup. Lembre-se: neste exemplo estamos usando o **tftp,** que funciona por **udp**, ou seja, pode perder pacotes e encerrar o backup sem sucesso.

```
 1 #exemplo19.2
 2 use 5.12.0;
 3 use FileHandle;
 4
 5 opendir my $dir, ".";
 6
 7 while (my $file = readdir $dir){
 8  next if $file !~ /\.txt$/;
 9  my $log_file = FileHandle->new($file, 'r');
```

```
 10  my $bkp_ok = 0;
 11  while($_ = <$log_file>){
 12    $bkp_ok = 1 if $_ =~ /File transfer operation com-
pleted successfully./;
 13  }
 14  say "bkp_file $file not ok" if not $bkp_ok;
 15  $log_file->close;
 16 }
```

Este segundo script é bem simples: ele abre o diretório onde se encontram os logs e busca pela string "File transfer operation completed successfully". Caso o arquivo de logs não tenha esta informação, a transferência falhaou e o script informa em quais elementos o backup falhou.

Eu utilizo muito o **Net::Telnet** para realizar auditorias de configuração em tempo real enviando comandos de verificação de um elemento e utilizando uma regex como máscara para verificar se a saída está dentro de parâmetros esperados.

Caso você realize backups diários, pode utilizar o módulo **Text::Diff** ou **Text::Diff::Table** para gerar relatório de mudanças ocorridas nas configurações dos arquivos.

19.4 Criando um servidor de syslog

Um servidor de syslog é útil para centralizar as informações dos eventos de um ambiente de rede e para análise de acontecimentos passados, auditoria, perícia ou mesmo manutenção preventiva, pois diversos sistemas enviam mensagens de problemas antes de ficarem indisponíveis. Um exemplo simples de servidor de syslog pode ser visto no exemplo:

```
  1   #exemplo19.3
  2   use Net::Syslogd;
  3   my $syslogd = Net::Syslogd->new()
  4     or die "Error creating Syslogd listener: ",
Net::Syslogd->error;
  5
```

```
 6   while (1) {
 7       my $message = $syslogd->get_message();
 8
 9       if (!defined($message)) {
10           printf "$0: %s\n", Net::Syslogd->error;
11           exit 1
12       } elsif ($message == 0) {
13           next
14       }
15
16       if (!defined($message->process_message())) {
17           printf "$0: %s\n", Net::Syslogd->error
18       } else {
19           printf "%s\t%i\t%s\t%s\t%s\t%s\t%s\n",
20                  $message->peeraddr,
21                  $message->peerport,
22                  $message->facility,
23                  $message->severity,
24                  $message->time,
25                  $message->hostname,
26                  $message->message
27       }
28   }
```

Este script é bem simples e pode ser facilmente alterado para enviar os logs para um arquivo de texto ou para um banco de dados. Depois é só criar as rotinas com regexp para verificar os erros desejados. Você pode enviar logs mais graves por e-mail.

19.5 O que ler agora?

Além da documentação dos módulos ***Net::SNMP***, ***Net::Telnet*** e ***Net::SSH*** eu recomendo:

1. Managing Internetworks with SNMP de Mark A. Miller, editora M&ST Books;
2. Automating System Administration with Perl de David N. Blank, editora O'Reilly (este livro tem incríveis 672 páginas sobre administração de sistemas com Perl);
3. Exame de redes com NMAP de Gordon Lyon, editora Ciência Moderna;
4. Penetration Testing and Network Defense de Andrew Whitaker e Daniel Newman, editora Cisco Press (mostra diversas ferramentas para teste de rede, muitas escritas em Perl);
5. Leia a documentação dos módulos ***NMAP::Parser*** e ***Net::Pcap***.

Exercícios

1. Instale um servidor Telnet em seu computador. Depois crie uma classe filha da classe **Net::Telnet** que implemente métodos que executem comandos via telnet em seu computador. Os métodos seriam, por exemplo: ***MeuTelnet->dir( )*** executaria o comando dir no computador Windows. A função AUTOLOAD pode ser muito útil neste exercício.

2. A instalação do suporte ao SNMP no Windows e no Linux é bem simples. Instale o suporte ao SNMP e monte um script para ler as informações de nome do sistema e tempo de disponibilidade.

3. Crie scripts de parser do comando **NMAP**. O **NMAP** pode ser obtido em http://insecure.org.

Capítulo 20. Dicas para o Mundo Windows

No início do livro dissemos como os programas em Perl são multiplataforma devido à sua característica de implementação. De fato, tudo que fizemos até agora funciona em múltiplas plataformas. Porém, a administração de sistemas, mais especificamente servidores e máquinas cliente com sistemas operacionais distintos, torna-se um pouco fora deste perfil. A filosofia de funcionamento do Windows é diferente da filosofia do Linux. Mesmo os sistemas Unix-Like têm diferentes filosofias entre si, tornando muito difícil que um software único alcance todas as variáveis mensuráveis de todos os sistemas. As variáveis comuns, como uso de processador, memória e disco, podem ser obtidas por meio de SNMP e MIBs padrão, definidas pela RFC. Porém, coisas peculiares a cada sistema devem ser tratadas de forma específica. Existem esforços para unificar a administração de servidores, como o OpenWBEM. Mas sistemas com filosofias de construção distintas exigem ferramentas distintas para verificar suas diversas condições.

Podemos citar como peculiaridades dos sistemas Windows o Registro, o ActiveDirectory, o Sistema de Arquivos FAT e NTFS etc. Assim, por mais que queiramos reduzir esforços de programação com linguagens multiplataformas, utilizar filosofias de construção diferenciadas ainda é uma arma de marketing utilizada pelas empresas de software proprietário. Assim, inevitavelmente, um sistema Windows terá variáveis a serem monitoradas diferentes das variáveis de um sistema Linux, ou as posições a serem monitoradas estarão em locais diferentes.

Para atender a estas diferenças de funcionamento e ser utilizado como linguagem para administração de sistemas também no Windows, os desenvolvedores do Perl criaram os módulos no namespace **Win32**.

20.1 O módulo Win32

O módulo Win32 fornece uma interface para o Perl acessar as funções da API dos sistemas Windows. Esta API dá acesso a diversas funções úteis. Você vai reparar que o Perl fornece meios de efetuar algumas destas operações com suas funções próprias ou outros módulos CPAN, como Win32::CopyFile, que pode ser substituída por código Perl, abrindo filehandles e copiando o conteúdo de um para o outro. Porém, a biblioteca do Win32 permite fazê-lo com menos código e maior desempenho. Por outro lado, perde-se a compatibilidade entre plataformas do seu código. Lembre-se: em TIC não existe almoço grátis.

Win32::CopyFile(From, TO, OVERWRITE)

Realiza cópia de um arquivo

Win32::CreateDirectory(DIRECTORY)

Cria um diretório

Win32::CreateFile(FILE)

Cria um arquivo, como o comando **touch** do Linux

Win32::DomainName()

Retorna o nome do domínio onde o sistema se encontra inserido

Win32::FsType()

Retorna o tipo do FileSystem

Win32::GetChipName()

Retorna o tipo do processador

Win32::GetCwd()

Informa o diretório atual, como o comando **cwd** do Linux

Win32::GetCurrentProcessId();

Obtém o ID do processo

Win32::GetCurrentThreadId();

Obtém a ID da thread

Win32::GetOSName()

Obtém o nome do OS (Windows 2003, Windows 2008 etc.)

Win32::GetTickCount()

Obtém o tempo de disponibilidade do sistema em milissegundos

Win32::SetCWD(PATH)

Modifica o diretório atual, como o comando **cd**

Win32::Sleep(miliseconds)

Interrompe a execução do processo/thread pelo tempo informado

Win32::Process(Comand, ARGS, PID)

Executa um comando

Win32::RegisterServer(LIBRARYNAME)

Carrega uma DLL

Win32::UnregisterServer(LIBRARYNAME)

Descarrega a DLL

```
1 #exemplo20.1
2 use strict;
3 use Win32;
4 print "Architecture  : ".Win32::GetArchName()."\n";
5 print "Chip          : ".Win32::GetChipName()."\n";
6 print "Node Name     : ".Win32::NodeName()."\n";
7 print "Login Name    : ".Win32::LoginName()."\n";
8 print "OS Name       : ".Win32::GetOSName()."\n";
9 print "Dominio       : ".Win32::DomainName()."\n";
```

```
 10 my ($str,$maj,$min,$build,$id) = Win32::GetOSVersion();
 11 print "$str $maj $min $build $id\n";

saída:

Architecture  : x86
Chip          : 586
Node Name     : TICursos
Login Name    : x8ge
OS Name       : Win2003
Dominio       : TICursos
Service Pack 2 5 2 3790 2
```

20.2 Win32::EventLog

Os sistemas Linux têm seus logs, na maioria dos casos armazenados em arquivos de texto, bastando um algoritmo de pesquisa e extração para obter a informação desejada. Já na filosofia Windows, os logs são armazenados em um formato binário próprio com a extensão **.evt**. Para obter os valores destes arquivos usamos o módulo ***Win32::EventLog*** em máquinas vivas. Em sistemas de análise forense é comum escrever um programa que leia o formato binário e o traduza. Muitos sistemas de análise forense usam o Perl, mas por alguma razão eles não costumam divulgar seus módulos no CPAN. Vimos no capítulo 9 como criar filehandles para processar binários, caso seja sua necessidade. Também existe o módulo **File::ReadEVT** encontrado em http://www.cpan.org/modules/by-authors/id/H/HC/HCARVEY/, que pode ser utilizado com fins de análise forense.

O módulo ***Win32::EventLog*** é orientado a objetos. Para criar um objeto deste módulo usamos o método ***new*** e informamos como argumento o tipo de **log** (Aplicação, Sistema, Segurança) e o **host** da rede que desejamos ler.

Alguns métodos do módulo não retornam valor, em vez disso, configuram variáveis que estão na função que evoca o método. Por exemplo, **GetNumber($numero_de_eventos)** não retorna um valor, mas atribui a **$numero_de_eventos** o número de eventos de logs existentes.

Para ler os eventos usamos o método **Read**, que configura um **hash_ref**, conforme você pode ver a seguir:

```
  1 #exemplo20.2
  2 use 5.12.0;
  3 use Win32::EventLog;
  4
  5 my $event = Win32::EventLog->new("System","localhost");
  6 my ($older, $new, $event_hash_ref);
  7
  8 $event->GetNumber($new);
  9 $event->GetOldest($older);
 10 say "$older\t$new";
 11 say EVENTLOG_SEEK_READ;
 12
 13 for my $x (1..5){
 14  $event->Read(EVENTLOG_BACKWARDS_READ|EVENTLOG_SEEK_
READ, $new - $x,
 15        $event_hash_ref);
 16  say "$_ => ", $event_hash_ref->{$_} for keys %$event_
hash_ref;
 17 }

saída:

124968     144097
2
TimeGenerated => 1280367168
ClosingRecordNumber => 0
RecordNumber => 144096
User =>
EventID => -1073741491
Computer => S6020AS03
Length => 40
Timewritten => 1280367168
Strings =>
Data => *&%^#$)(
```

Você vai perceber, ao executar o script, que parte das informações no **$event_hash_ref** não estão no formato correto. Para "desempacotar" a informação precisamos de um algoritmo que conheça o formato binário usado pelo EventLog do Windows e o transforme em algo humanamente compreensível. O módulo ***Win32::EventLog*** fornece a função (não orientada a objetos) ***GetMessageText*** para desempacotar estes dados. Esta função deve ser acessada pelo namespace completo: ***Win32::EventLog::GetMessageText***. Veja agora como fica o novo script que mostra esta informação oculta:

```
  1 #exemplo20.3
  2 use Win32::EventLog;
  3 my $event = Win32::EventLog->new("System","localhost");
  4 my ($older, $new, $event_hash_ref);
  5 $event->GetNumber($new);
  6 $event->GetOldest($older);
  7 for my $x (1..5){
  8  $event->Read(EVENTLOG_BACKWARDS_READ|EVENTLOG_SEEK_
READ, $new - $x,                    $event_hash_ref);
  9        Win32::EventLog::GetMessageText($event_hash_
ref);
 10  say $event_hash_ref->{Message};
 11 }

saída:

O serviço Windows Defender entrou no estado executando.

O serviço Proteção de Software entrou no estado executando.

O serviço Google Update Service (gupdate) entrou no estado
executando.

O serviço Serviço de Entrada de Tablet PC entrou no estado
interrompido.
```

O que fazer com estas informações? Bem, você percebeu que o módulo permite conexão a diferentes hosts da rede no construtor ***Win32::EventLog->new("System","localhost")***. Assim, você pode ler diversos logs dos servidores ou hosts da sua rede, armazenar em um banco de dados como um serviço centralizado de syslog, ou mesmo monitorar os eventos do seu ambiente.

20.3 Manuseando o registro do Windows

O *Microsoft Computer Dictionary*, quinta edição, define o registro como: um banco de dados hierárquico central no Microsoft Windows 98, no Windows CE, no Windows NT e no Windows 2000 usado para armazenar as informações necessárias à configuração do sistema para um ou mais usuários, aplicativos e dispositivos de hardware.

Este banco de dados pode ser visualizado com o aplicativo regedit do Windows, em iniciar > executar > regedit.

As informações do registro dizem como o Windows deve configurar a sessão de usuário, quais arquivos um aplicativo pode abrir, a chave utilizada para instalar aplicativos, o domínio e as diretivas de segurança do equipamento etc.

Os arquivos do registro se localizam na pasta %SystemRoot%\System32\Config e estão distribuídos conforme a tabela:

Ramificação do Registro	**Arquivos de suporte**
HKEY_LOCAL_MACHINE\SAM	Sam, Sam.log, Sam.sav
HKEY_LOCAL_MACHINE\Security	Security, Security.log, Security.sav
HKEY_LOCAL_MACHINE\Software	Software, Software.log, Software.sav
HKEY_LOCAL_MACHINE\System	System, System.alt, System.log, System.sav
HKEY_CURRENT_CONFIG	System, System.alt, System.log, System.sav, Ntuser.dat, Ntuser.dat.log
HKEY_USERS\DEFAULT	Default, Default.log, Default.sav

Os arquivos de suporte para HKEY_CURRENT_USER estão na pasta **%SystemRoot%\Profiles\Username.**

Cada uma das chaves é utilizada para diferentes finalidades. Você pode consultar a finalidade e o tipo de dado de cada chave em http://support.microsoft.com/kb/256986.

Como você pode perceber, utilizar o registro deve ser feito com cuidado, pois pode prejudicar toda a instalação do Windows. Por outro lado, o registro é muito utilizado por desenvolvedores de hardware, sistemas de segurança e para fins de análise forense, uma vez que registra todos os softwares instalados. Pode ainda ser utilizado para comprovar utilização ilegal de sistema ou mesmo para rastrear atividades de usuários. Muitos programas, mesmo sendo removidos, mantêm as chaves no registro; assim, um registro de que o software foi utilizado naquele sistema permanece. Outro ponto importante a notar, apesar das chaves principais permanecerem as mesmas, é que os valores e tipos de dados de uma informação específica podem mudar entre as versões do Windows, tornando o trabalho com o registro ainda mais especializado.

Caso queira conhecer o registro do Windows em detalhes, você pode ler Microsoft® Windows® Registry Guide, Second Edition, de Jerry Honeycutt (como você pode ver, o registro é tão complexo que resultou em um livro de mais de 600 páginas).

Mas vamos ver um pouco sobre como podemos manipular o registro com o Perl.

20.4 Lendo os fabricantes de software instalados

Os fabricantes de softwares para Windows criam uma chave em **HKEY_LOCAL_MACHINE\Software**. Cada fabricante cria uma chave própria, assim podemos ler os fabricantes com um script bem simples:

```
1 #exemplo20.4
2 use 5.12.0;
3 use Win32::TieRegistry(Delimiter=>"/");
4
5 my $fabricantes = $Registry->{"LMachine/Software"};
6 say for keys %$fabricantes;
```

```
saída:

Apache Software Foundation/
Avg/
Dev-C++/
Google/
Microsoft/
Mozilla/
mozilla.org/
MozillaPlugins/
OpenOffice.org/
Oracle/
Windows/
WinPcap/
```

Agora vamos ler o que existe dentro de uma dessas chaves – por exemplo, a chave referente ao OpenOffice.

```
  1 #exemplo20.5
  2 use 5.12.0;
  3 use strict;
  4 use Win32::TieRegistry(Delimiter=>"/");
  5
  6 my $OpenOffice = $Registry->{"LMachine/Software/OpenO-
ffice.org"};
  7 say for keys %$OpenOffice;

saída:

BrOffice.org/
Layers/
UNO/
/QuickStarterInstalled
```

Perceba que /QuickStarterInstalled é um registro e não uma chave, que possui um valor que pode ser lido. Como eu sei? Simples: o caractere / aparece no início de um registro ou no fim de uma chave. Agora adicione este código ao final do arquivo:

```
  9 #QuickStarterInstalled é um registro que informa se o
QuickStarter do
 10 #OpenOffice foi instalado.
 11 say $OpenOffice->{"/QuickStarterInstalled"};
```

Outra chave que podemos olhar dentro da árvore do OpenOfficce é a associação de extensão de arquivos e do aplicativo:

```
 13 #Le a associação de extensão com aplicativo
 14 my $files = $OpenOffice->{"BrOffice.org/3.2/Capabili-
ties/FileAssociations"};
 15 say "$_ => ", $files->{$_} for sort keys %$files;
```

Que no meu sistema gera a saída:

```
/.123 => opendocument.CalcDocument.1
/.csv => opendocument.CalcDocument.1
/.dbf => opendocument.CalcDocument.1
/.dif => opendocument.CalcDocument.1
/.doc => opendocument.WriterDocument.1
/.docm => opendocument.WriterDocument.1
/.docx => opendocument.WriterDocument.1
/.dot => opendocument.WriterDocument.1
/.dotm => opendocument.WriterDocument.1
/.dotx => opendocument.WriterDocument.1
/.htm => opendocument.WriterWebDocument.1
/.html => opendocument.WriterWebDocument.1
/.hwp => opendocument.WriterDocument.1
/.odb => opendocument.DatabaseDocument.1
/.odf => opendocument.MathDocument.1
/.odg => opendocument.DrawDocument.1
… saída omitida ...
```

Você também pode utilizar o **Win32::TieRegistry** para escrever no registro do Windows, embora isto deva ser feito após estudo minucioso do registro e dos impactos que podem ser gerados.

20.5 DBD::WMI

Para usar o ***DBD::WMI*** de forma produtiva você precisa ser administrador do sistema. Se for obter informações em máquinas remotas, vai precisar ser administrador do domínio. Para obter informações completas sobre o WMI acesse: http://msdn.microsoft.com/en-us/library/aa394572%28v=VS.85%29.aspx

O DBD::WMI cria uma interface SQL entre o seu script e o repositório de informações WMI do Windows. Este repositório é o Windows Management Instrumentation, que é a resposta da Microsoft para o WBEM.

O script a seguir pega as informações de processador da máquina via ***DBD::WMI***:

```
  1 #exemplo20.5
  2 use 5.12.0;
  3 use DBI;
  4
  5 #Crio uma conexão com o WMI local
  6 my $db_handle = DBI->connect('dbi:WMI:');
  7
  8 #Win32_Processor é a classe definida pela Microsoft
para armazenar
  9 #informações do processador no WMI
 10 my $sql = "Select * from Win32_Processor"; #Crio a
query SQL
 11
 12 my $sth = $db_handle->prepare($sql); #Prepara a execu-
ção da query
 13 my $afected_rows = $sth->execute(); #Executa a query
 14
 15 while (my @row = $sth->fetchrow) { #Pega as informações
 16  say "Caption: ", $row[0]->{Caption};
 17  say "Nome: ", $row[0]->{Name};
 18  say "L2: ", $row[0]->{L2CacheSize};
 19  say "L3: ", $row[0]->{L3CacheSIze};
 20  say "Description: ", $row[0]->{Description};
 21 }
```

saída:

```
Caption: x64 Family 6 Model 23 Stepping 6
Nome: Intel(R) Core(TM)2 Duo CPU E7200 @ 2.53GHz
L2: 3072
L3: 0
Description: x64 Family 6 Model 23 Stepping 6
```

Então a informação pode ser processada e armazenada em um arquivo ou banco de dados. Para obter informações de uma máquina remota, basta modificar o início do script:

```
 my $remoto = 'dawn'; my $dbh = DBI-
>connect('dbi:WMI:'.$remoto);
```

Esse é o princípio de um sistema de inventário dos servidores.

20.6 O que ler agora?

Para administrar sistemas Windows você vai precisar de mais conhecimentos em Windows do que em Perl. Mas aí vão minhas recomendações:

1. Perl Scripting for Windows Security, de Harlan Carvey e Dave Kleiman, editora Syngress;
2. Windows Forensics and Incident Recovery, de Harlan Varvey, editora Addison-Wesley Professional;
3. Programming PERL in the .NET Environment, de Yevgeny Menaker e outros, editora Addison-Wesley Professional;
4. Microsoft Windows Registry Guide, de Jerry Honeycutt, editora Microsoft Press;
5. Microsoft Windows Scripting with WMI: Self-Paced Learning Guide de Ed Wilson, editora Microsoft Press (Este livro usa programação VBS, que pode ser facilmente convertida em scripts Perl se você conhecer um pouco de Visual Basic).

Exercícios

1. Se você tiver uma máquina Windows, estude a documentação sobre WMI em http://msdn.microsoft.com/en-us/library/aa394572%28v=VS.85%29.aspx e crie um script que obtenha as mesmas informações do exercício 2 do capítulo 19. Use o módulo Benchmark para comparar o tempo de execução dos dois scripts.

2. Em http://en.wikipedia.org/wiki/OLE_Automation você encontra um artigo sobre automação OLE e exemplos em Visual Basic Script, Perl e C++. Leia e tire suas conclusões. Aproveite e crie um script Perl que manuseie uma planilha Excel de sua escolha, se você tiver acesso ao Microsoft Office.

3. Com os módulos ***Win32*** crie sua própria versão dos comandos **dir**, **cd** e **type**.

Capítulo 21. Monitorando Sistemas Linux

A filosofia de funcionamento dos sistemas Unix e Linux é diferente dos sistemas Windows. Os arquivos de configuração e log em sua ampla maioria são arquivos de texto comuns com caracteres especiais como separadores das informações. Assim, não haverá nada que seja muito difícil para um programador Perl na criação de scripts e programas para administrar sistemas Linux.

Como as tarefas administrativas são muito corriqueiras, existem muitos módulos no CPAN que já obtêm determinadas informações processadas e formatadas de forma mais simples ainda.

21.1 Analisando um arquivo de log comum do Linux

Os arquivos de log do Linux geralmente tem o formato como o dos arquivos mostrados adiante:

```
logs de erro do apache:
root# tail /var/log/apache2/error.log
[Sun Jan 23 20:43:50 2011] [notice] Apache/2.2.14 (Ubuntu)
mod_fcgid/2.3.4 mod_perl/2.0.4 Perl/v5.10.1 configured --
resuming normal operations
[Mon Jan 24 07:18:21 2011] [notice] caught SIGTERM, shut-
ting down
[Mon Jan 24 07:20:48 2011] [notice] Apache/2.2.14 (Ubuntu)
mod_fcgid/2.3.4 mod_perl/2.0.4 Perl/v5.10.1 configured --
resuming normal operations
[Mon Jan 24 07:25:41 2011] [notice] caught SIGTERM, shut-
ting down
```

```
[Mon Jan 24 08:38:52 2011] [notice] Apache/2.2.14 (Ubuntu)
mod_fcgid/2.3.4 mod_perl/2.0.4 Perl/v5.10.1 configured --
resuming normal operations
[Tue Jan 25 11:58:42 2011] [notice] Apache/2.2.14 (Ubuntu)
mod_fcgid/2.3.4 mod_perl/2.0.4 Perl/v5.10.1 configured --
resuming normal operations
```

logs do syslog (/var/log/syslog):

```
root # tail /var/log/syslog
Jan 29 18:53:32 thiago-laptop NetworkManager: <info>
(wlan0): supplicant connection state:  group handshake ->
completed
Jan 29 18:59:32 thiago-laptop wpa_supplicant[917]: WPA:
Group rekeying completed with 00:22:57:63:3d:45 [GTK=TKIP]
Jan 29 18:59:32 thiago-laptop NetworkManager: <info>
(wlan0): supplicant connection state:  completed -> group
handshake
Jan 29 18:59:32 thiago-laptop NetworkManager: <info>
(wlan0): supplicant connection state:  group handshake ->
completed
```

logs do kernel:

```
root# tail /var/log/kern.log
Jan 28 22:21:02 thiago-laptop kernel: [ 7851.665826] sd
6:0:0:0: [sdb] 31252024 512-byte logical blocks: (16.0
GB/14.9 GiB)
Jan 28 22:21:02 thiago-laptop kernel: [ 7851.666420] sd
6:0:0:0: [sdb] Write Protect is off
Jan 28 22:21:02 thiago-laptop kernel: [ 7851.666424] sd
6:0:0:0: [sdb] Mode Sense: 16 07 09 51
Jan 28 22:21:02 thiago-laptop kernel: [ 7851.666426] sd
6:0:0:0: [sdb] Assuming drive cache: write through
Jan 28 22:21:02 thiago-laptop kernel: [ 7851.669413] sd
6:0:0:0: [sdb] Assuming drive cache: write through
```

log de erros do mysql:

```
root# tail /var/log/mysql/error.log-old
110128 20:09:50 [Note] Event Scheduler: Purging the queue.
```

```
0 events
110128 20:09:50  InnoDB: Starting shutdown...
110128 20:09:51  InnoDB: Shutdown completed; log sequence
number 0 43655
110128 20:09:51 [Note] /usr/sbin/mysqld: Shutdown complete
110128 20:10:35 [Note] Plugin 'FEDERATED' is disabled.
110128 20:10:37  InnoDB: Started; log sequence number 0
43655
110128 20:10:37 [Note] Event Scheduler: Loaded 0 events
110128 20:10:37 [Note] /usr/sbin/mysqld: ready for connec-
tions.
Version: '5.1.41-3ubuntu12.9'  socket: '/var/run/mysqld/
mysqld.sock'  port: 3306  (Ubuntu)
```

Você pode observar com cuidado os arquivos de log e verificar que alguns seguem o formato de log do syslog do Unix e outros têm um formato próprio. Então, para criar soluções que extraiam ou gravem logs, você precisa estudar o formato utilizado pelo formato desejado ou então verificar no CPAN se já existe algum Parse mastigado para esse log. Por exemplo, ***Apache::ParseLog*** e ***Parse::Syslog*** podem ajudar muito nesses casos.

Mas o mais legal nos sistemas Linux e Unix de código aberto é a possibilidade de ler até mesmo os arquivos com formato binário. Por exemplo: o arquivo wtmp, que armazena os logins e logouts dos usuários em um sistema Linux, é um arquivo binário. Na minha distribuição Ubuntu ele fica em /var/log/wtmp. Você lembra no capítulo 9 que acessamos um arquivo binário de imagem com as funções open e unpack. Vamos fazer o mesmo com o wtmp.

As definições dos dados do wtmp estão em **man wtmp.** O comando man vai lhe informar que os dados do arquivo wtmp provêm de um **struct** feito em **c** (algo parecido com o hash do Perl):

```
struct utmp {
       short   ut_type;              /* Type of record */
Ok
       pid_t   ut_pid;               /* PID of login pro-
cess */Ok
       char    ut_line[UT_LINESIZE]; /* Device name of tty
- "/dev/" */ Ok
       char    ut_id[4];             /* Terminal name suf-
```

```
fix, or inittab(5) ID */ Ok
       char    ut_user[UT_NAMESIZE]; /* Username */ Ok
       char    ut_host[UT_HOSTSIZE]; /* Hostname for remote
login, or kernel version for run-level
                                                   messages */
Ok
       struct  exit_status ut_exit;  /* Exit status of a
process marked as DEAD_PROCESS; not
                                                   used by
Linux init(8) */
               /* The ut_session and ut_tv fields must be
the same size when
                  compiled 32- and 64-bit.  This allows
data files and shared
                  memory to be shared between 32- and 64-
bit applications. */
           #if __WORDSIZE == 64 && defined __WORDSIZE_COM-
PAT32
               int32_t ut_session;           /* Session ID
(getsid(2)), Ok
                                                   used for
windowing */
               struct {
                   int32_t tv_sec;           /* Seconds */
Ok
                   int32_t tv_usec;          /* Microsec-
onds */ Ok
               } ut_tv;                      /* Time entry
was made */ ok
           #else
                long   ut_session;           /* Session ID
*/ ok
                struct timeval ut_tv;        /* Time entry
was made */ ok
           #endif

       int32_t ut_addr_v6[4];        /* Internet address of
remote host; IPv4 address uses just
     ut_addr_v6[0] */
               char __unused[20];            /* Reserved
for future use */
   };
```

Você pode ler o código deste arquivo comentado em http://manpages.ubuntu.com/manpages/hardy/pt/man5/utmp.5.html

A partir desta estrutura podemos definir uma máscara com **s** para os tipos numéricos de 16 bits; **l** para os tipos numéricos com 32 bits e **A** para os tipos caractere (char).

Assim, nossa máscara será: **"l l A32 A4 A32 A256 s s l l l l4 A20"** e podemos ler os dados do arquivo binário:

```
  1 #exemplo21.1
  2 use 5.12.0;
  3 open(my $wtmp, "<", "/var/log/wtmp");
  4
  5 #Criamos a máscara de acordo com o arquivo que define a
estrutura de wtmp
  6 my $mask = 'l l A32 A4 A32 A256 s s l l l l4 A20';
  7
  8 #Definimos o tamanho em bytes da máscara
  9 my $record_size = length( pack($mask) );
 10 say $record_size;
 11
 12 #Criamos uma variável para cada tipo definido na másca-
ra
 13 my ($type, $pid, $line, $id, $user, $host, $termina-
tion, $exit,
 14  $sessao, $time1, $time2, $ip, $descarte);
 15 my $record;
 16
 17 #Lemos o arquivo em lotes do tamanho da máscara (re-
cord_size)
 18 while( read($wtmp, $record, $record_size) ){
 19
 20 #unpack retorna um array onde cada elemento é um item
da máscara
 21  ($type, $pid, $line, $id, $user, $host, $termination,
$exit,
 22        $sessao, $time1, $time2, $ip, $descarte ) =
unpack($mask, $record);
```

```
 23
 24  say "$type:$pid:$line:$id:$user:$host:$termination:$ex
it:$sessao:" .
 25  scalar localtime($time1) . ":$time2:$descarte";
 26 }

saída:

7:2263:pts/0:/0:thiago::0.0:0:0:0:Tue Jan  4 10:41:29
2011:271507:0
7:2338:pts/1:/1:thiago::0.0:0:0:0:Tue Jan  4 10:44:48
2011:750436:0
8:0:pts/1:/1:thiago::0:0:0:Tue Jan  4 10:44:55
2011:173562:0
7:2390:pts/1:/1:thiago::0.0:0:0:0:Tue Jan  4 10:45:10
2011:555844:0
8:0:pts/1:/1:thiago::0:0:0:Tue Jan  4 10:45:20
2011:165550:0
7:2439:pts/1:/1:thiago::0.0:0:0:0:Tue Jan  4 10:46:04
2011:205329:0
8:0:pts/1:/1:thiago::0:0:0:Tue Jan  4 10:46:24
2011:325387:0
7:2489:pts/1:/1:thiago::0.0:0:0:0:Tue Jan  4 10:48:14
2011:716501:0
```

Bom, a moral da história é: trabalhar com parsing de logs no Linux é muito simples se os arquivos forem em formato de texto. Caso eles sejam binários podemos ter um pouco mais de trabalho, porém, como o código do Linux é aberto, podemos encontrar facilmente na internet a documentação necessária para criar um programa em Perl que acesse o arquivo e extraia a informação necessária.

NOTA: Parsing é um termo técnico inglês que podemos definir como dividir uma informação complexa em unidades de dados menores que podem ser extraídos e tenham utilidade e significado para sua aplicação.

21.2 Dados da estatística do sistema

O Linux concentra as informações de seu estado em **/proc.** E para a maior parte das informações relevantes em **/proc** o CPAN oferece módulos que extraem estas informações prontas para o uso. Os módulos são:

Sys::Statistics::Linux::SockStats
Sys::Statistics::Linux::ProcStatus
Sys::Statistics::Linux::Processes
Sys::Statistics::Linux::PgSwStats
Sys::Statistics::Linux::NetStats
Sys::Statistics::Linux::MemStats
Sys::Statistics::Linux::LoadAVG
Sys::Statistics::Linux::FileStats
Sys::Statistics::Linux::DiskUsage
Sys::Statistics::Linux::DiskStats
Sys::Statistics::Linux::CpuStats
Sys::Statistics::Linux::Compilation
Sys::Statistics::Linux::SysInfo

Um pequeno exemplo de como utilizar estes módulos extremamente simples está logo a seguir. Ele acessa as informações de processador por meio do ***Sys::Statistics::Linux::CpuStats***.

```
 1   #exemplo21.2
 2   use 5.12.0;
 3   use Sys::Statistics::Linux::CpuStats;
 4   my $lxs = Sys::Statistics::Linux::CpuStats->new();
 5   # $lxs->get retorna um hash_ref
 6   $lxs->init;
 7   my $stats = $lxs->get;
 8
 9   &hash_stripper($stats, sort keys %$stats );
10
11
12   sub hash_stripper{
13        my $hash_ref = shift;
```

```
 14        my @keys = @_;
 15        foreach my $item ( @keys ) {
 16     if( $hash_ref->{$item} =~ /HASH/ ){
 17       say $item;
 18       &hash_stripper($hash_ref->{$item}, sort keys
%{$hash_ref->{$item}});
 19     }
 20     else{
 21       say "\t$item =>" . $hash_ref->{$item};
 22     }
 23   }
 24 }
```

saída:

```
cpu
     idle =>0.00
     iowait =>0.00
     irq =>0.00
     nice =>0.00
     softirq =>0.00
     steal =>0.00
     system =>0.00
     total =>100.00
     user =>0.00
cpu0
     idle =>0.00
     iowait =>0.00
     irq =>0.00
     nice =>0.00
     softirq =>0.00
     steal =>0.00
     system =>0.00
     total =>100.00
     user =>0.00
cpu1
     idle =>0.00
     iowait =>0.00
```

```
     irq =>0.00
     nice =>0.00
     softirq =>0.00
     steal =>0.00
     system =>0.00
     total =>100.00
     user =>0.00
```

Você pode usar ou modificar a função hash_stripper para extrair os hashes de hashes que os módulos retornam. O meu sistema tem um CPU com dois núcleos, o que faz com que o módulo retorne 3 hashes. Como não podemos saber quantas CPUs as máquinas terão, nós fizemos uma função recursiva, que chama a si própria toda vez que a chave de um hash_ref for outro hash_ref.

```
 16     if( $hash_ref->{$item} =~ /HASH/ ){
 17       say $item;
 18       &hash_stripper($hash_ref->{$item}, sort keys
%{$hash_ref->{$item}});
 19     }
```

A função é bem simples. Ela recebe um hash_ref um array com as chaves do hash e imprime o conteúdo das chaves.

```
&hash_stripper($stats, sort keys %$stats );
```

Assim, o hash_ref inicial pode conter hash_refs em suas chaves indefinidamente que a função irá extraí-los. Este código é bem flexível. Para extrair as informações de outro item, basta modificar o módulo carregado e iniciado:

```
  1 #exemplo21.3
  2 use 5.12.0;
  3
  4     use Sys::Statistics::Linux::DiskUsage;
  5     my $lxs = Sys::Statistics::Linux::DiskUsage->new();
  6     my $stats = $lxs->get;
```

```
  7
  8     &hash_stripper($stats, sort keys %$stats );

saída:

/dev/sda1
     free =>204698096
     mountpoint =>/
     total =>231594420
     usage =>15131928
     usageper =>7
none
     free =>1512708
     mountpoint =>/lib/init/rw
     total =>1512708
     usage =>0
     usageper =>0
```

E, como sempre, você pode fazer o que quiser com a informação de saída.

21.3 O que ler agora?

Trabalhar com Linux e Perl é muito simples porque ambos nasceram sobre a filosofia do Unix. Para automatizar a administração de servidores Linux é necessário o conhecimento da estrutura básica do sistema, além da programação. Exitem muitas fontes que poderão aprimorar seus conhecimentos em Linux, além de desenvolver suas habilidades em programação. Uma fonte que trata de administração de Sistemas, Linux é Automating System Administration with Perl de David N. Blank, editora O'Reilly

Exercícios

1. Se tiver acesso a uma máquina com um sistema Linux, monte um script que obtenha as informações de memória, uso de disco e processador de cinco em cinco minutos e armazene em um banco de dados. Monte um script CGI que permita ao usuário analisar as informações do banco de forma gráfica.

2. Habilite o SNMP no sistema Linux e obtenha as mesmas informações do exercicio anterior via SNMP e compare o desempenho de ambos os sistemas.

Capítulo 22. Android

O Android é um sistema operacional para smartphones baseado no kernel do Linux. Hoje é um projeto patrocinado pelo Google, presente em 9,6% dos smartphones – superando as vendas de Windows Mobile em 2010 (http://www.gartner.com/it/page.jsp?id=1372013). Um ótimo mercado a ser explorado, não?

Para desenvolver projetos para o Android você vai precisar de um SDK com um simulador de dispositivos móveis. Para instalar o simulador do Android e seu SDK você precisa do Java instalado em seu sistema:

```
$ javac -version

javac 1.6.0_20

$ java -version

java version "1.6.0_20"

OpenJDK Runtime Environment (IcedTea6 1.9.7) (6b20-1.9.7-
0ubuntu1~10.04.1)

OpenJDK Server VM (build 19.0-b09, mixed mode)

$
```

Caso você não tenha o Java SDK instalado no seu sistema, você pode baixar no site da Oracle **http://www.oracle.com** ou **http://openjdk.java.net/**.

Uma vez instalado o JDK, você deve baixar o SDK para o Android em **http://developer.android.com**. Siga as instruções de instalação do site conforme sua plataforma.

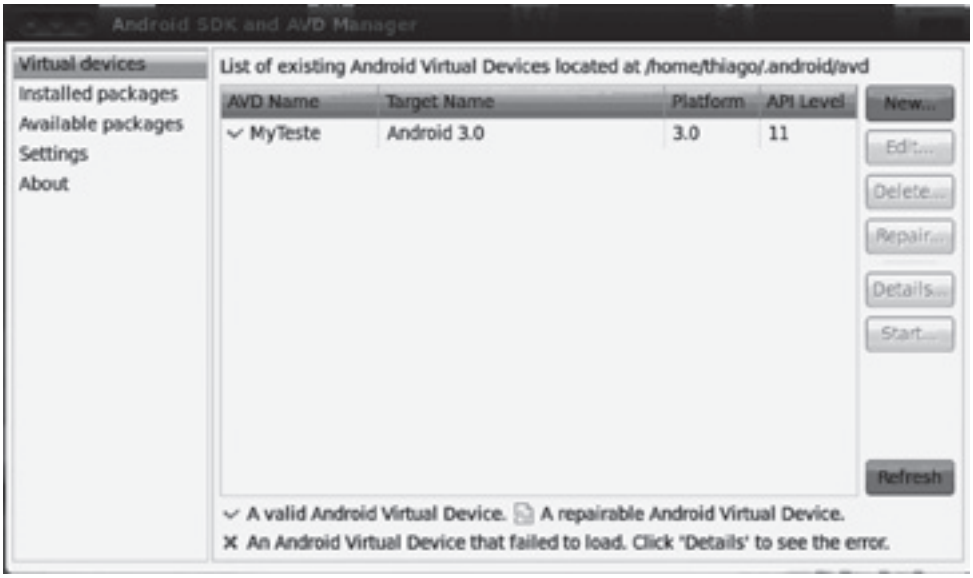
Adicione a pasta **tools** e **plataform-tools** de sua instalação ao PATH do seu sistema. Os comandos a seguir alteram o PATH em um sistema Linux:

```
$export PATH="$PATH:~/android-sdk-linux_86/tools"
$export PATH="$PATH:~/android-sdk-linux_86/platform-tools"
```

Para iniciar o SDK execute o comando ***$ android*** ou ***SDK Manager.exe*** dependendo da sua plataforma.

```
$ android
```

Aparecerá esta tela:



Agora precisamos criar um dispositivo virtual para trabalharmos. Clique em "**available packages**".

Escolha a versão do **SDK Plataform Android** de acordo com a versão do Android que você quer utilizar e clique em **Install Selected.** Siga as instruções. Estou utilizando a versão **SDK Plataform Android 3.0.** Para criar um virtual Device que irá emular o dispositivo móvel clique em **Virtual Devices > New**.

Dê um nome para seu dispositivo, no caso PerlMobile, escolha a resolução de tela e a memória de acordo com o dispositivo do seu projeto e por fim CREATE AVD. Depois de criada, inicie a AVD em Virtual Devices > Start. Fique tranquilo, pois demora um pouco para iniciar... é Java...

22.1 A camada de scripts do Android - SL4A

Agora que temos um simulador de Android funcionando, vamos instalar o SL4A no nosso simulador; assim poderemos criar programas em Perl para dispositivos móveis. A página do SL4A está no http://code.google.com/p/android-scripting/. Primeiro vamos instalar o SL4A e depois o interpretador do Perl.

1 Faça o download dos arquivos **sl4a_r4.apk, textedit-sl4a.apk** e **perl_for_android_r1.apk**.

2 No seu interpretador de comando vá até a pasta onde estão os arquivos dos downloads.

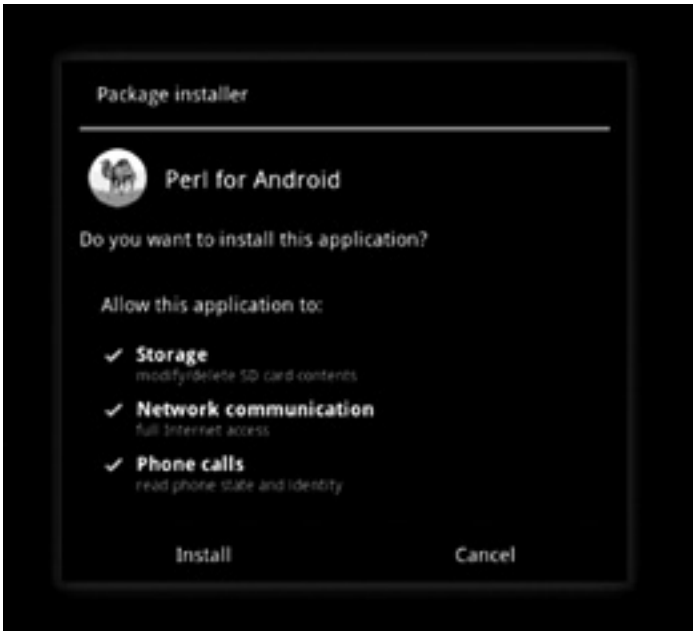
3 Inicie o emulador do Android.

4 Com o emulador executando, digite os comandos na sua CLI:

```
$ adb install sl4a_r4.apk480 KB/s (858471 bytes in 1.744s)
     pkg: /data/local/tmp/sl4a_r4.apk
Success
$adb install textedit-sl4a.apk
606 KB/s (131058 bytes in 0.211s)
        pkg: /data/local/tmp/textedit-sl4a.apk
Success
$ adb install perl_for_android_r1.apk
446 KB/s (33894 bytes in 0.074s)
     pkg: /data/local/tmp/perl_for_android_r1.apk
Success
```

Após esta instalação, os ícones do SL4A e do Perl devem aparecer na tela do seu emulador, em Applications:

Clique no link do Perl para concluir a instalação. Você vai receber a seguinte tela:

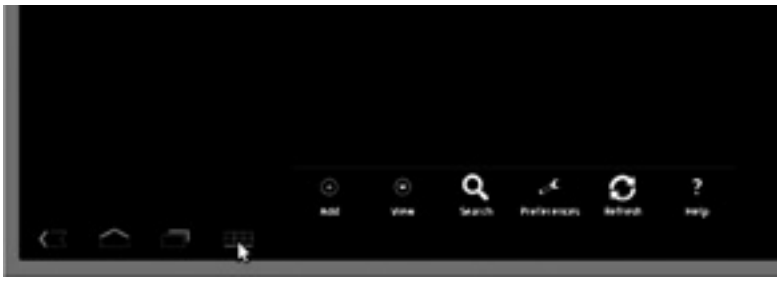


Clique em Install e siga o procedimento. Após concluída a instalação, vá para Applications > SL4A.

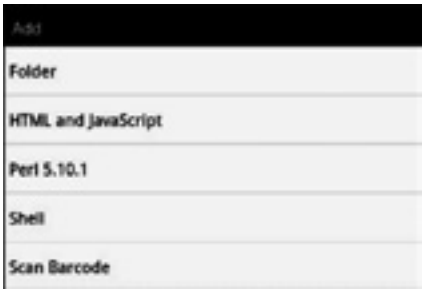
Você verá a tela abaixo. Clique em **Accept**.

Você verá os scripts de exemplo, que podem ser executados a qualquer hora:

Para adicionar o Perl debug, clique em **Add,** no canto inferior esquerdo da tela:



E depois clique em **Perl5.10.1** no menu que irá aparecer:



Agora você terá acesso ao Perl debugger como uma interface de linha de comando:

E poderá trabalhar com ele como uma linha de comando:

Agora que seu ambiente de desenvolvimento está pronto, estude a API do Android em http://code.google.com/p/android-scripting/wiki/ApiReference e divirta-se com a programação para mobile.

Capítulo 23. A Evolução do Perl 5.10 ao 5.14

Toda linguagem de programação amplamente utilizada sofre evolução com o tempo. Assim é com o Java, com o C++ e mesmo com o C. A evolução das versões do Perl tem por objetivo acompanhar as mudanças na forma de trabalhar dos programadores, manter o Perl em conformidade com padrões internacionais, corrigir bugs e falhas de segurança, além de melhoras de desempenho. A versão 5.10 adicionou melhoras consideráveis no Perl. E o mesmo ocorreu com as versões 5.12 e 5.14. Destas melhorias, merecem destaque:

- O operador ***smart match*** foi introduzido na versão 5.10. Ele continha alguns bugs que foram corrigidos na versão 5.12;
- O metacaractere \N para expressões regulares foi introduzido como teste na versão 5.12 e consolidado na versão 5.14;
- A capacidade de realizar substituições não destrutivas em expressões regulares foi introduzida na versão 5.14;
- Suporte ao padrão **Unicode 6**: A versão 5.12 tinha suporte à versão 5.2 do padrão Unicode (a mais atual à época do lançamento). Com o Perl 5.14 foi adicionado o suporte à versão 6 do padrão.

23.1 O operador smart match

O operador smart match é representado no código como uma dupla de til (~~). Ele realiza comparações diversas, conforme o contexto. Ou melhor, dependendo dos operandos da comparação. Deixe-me ser mais claro: o operador de strings ***eq***, por exemplo, compara duas strings; o operador de comparação numérica == verifica

se os dois números têm o mesmo valor; o smart match verifica seus argumentos e decide se fará uma comparação numérica ou uma comparação de strings:

```
1 #exemplo22.1
2 use 5.12.0;
3
4 say "Números diferentes!" if not 10 == 100;
5 say "Erro!!" if not 10 == "10casa"; #???
6 say "Eu sou esperto" if not 10 ~~ "10casa";
```

saída:

```
Números diferentes!
Eu sou esperto
```

O que aconteceu aqui? Neste exemplo (22.1), o Perl verificou que o número 10 não é igual ao número 100. Depois ele verificou que o número 10 é igual à string "10casa" porque o Perl permite que uma string que inicie com dígitos seja tratado como um número, mesmo que seus últimos caracteres não sejam dígitos. Internamente, o operador == entende que 10casa é o número 10, mas o operador smart match percebe a diferença. Porém, o smart match não é um operador comutativo:

```
1 #exemplo22.1
2 use 5.12.0;
3
4 say "Números diferentes!" if not 10 == 100;
5 say "Erro!!" if not 10 == "10casa"; #???
6 say "Eu sou esperto" if not "10casa" ~~ 10;
```

saída:

```
Números diferentes!
```

Não ser comutativo significa que a ordem dos fatores altera o produto, ou seja, a posição dos operandos altera o resultado em uma operação com o smart match. Assim, se o primeiro argumento (lvalue) puder ser interpretado como um

número, o smart match fará uma comparação numérica, e se for um string fará uma verificação de strings. Se o argumento da direita (rvalue) for uma expressão regular, o operador fará uma verificação de expressão regular:

```
 1 #exemplo22.3
 2 use 5.12.0;
 3
 4 my $name = "Dino da Silva Sauro";
 5
 6 say "Encontrei Sa!" if $name =~ /Sa/;
 7 say "Encontrei Silva!" if $name ~~ /Silva/;
 8 say "Dino da Silva Sauro é igual à variável \$name"
 9  if $name ~~ "Dino da Silva Sauro";

saída:

Encontrei Sa!
Encontrei Silva!
Dino da Silva Sauro é igual à variável $name
```

O smart match também pode ser utilizado como loop para verificar se dois arrays são idênticos ou se dois hashes têm as mesmas chaves:

```
 1 #exemplo22.4
 2 use 5.12.0;
 3
 4 my @arr1 = ('a'..'z');
 5 my @arr2 = ('a' .. 'z');
 6 my @arr3 = ('A' .. 'z');
 7
 8 say "\@arr1 é igual ao \@arr2." if @arr1 ~~ @arr2;
 9 say "\@arr3 é igual ao \@arr1." if @arr3 ~~ @arr1;
10 say "\@arr1 é igual ao \@arr3." if @arr1 ~~ @arr3;
11
12 my %hash1 = map{ $_ => " " } @arr1;
13 my %hash2 = map{ $_ => " " } @arr2;
14 my %hash3 = map{ $_ => " " } @arr3;
```

```
 15
 16 say "\%hash1 tem as mesmas chaves que \%hash2" if
%hash1 ~~ %hash2;
 17 say "\%hash1 tem as mesmas chaves que \%hash3" if
%hash1 ~~ %hash3;

saída:

@arr1 é igual ao @arr2.
%hash1 tem as mesmas chaves que %hash2
```

23.2 Substituição não destrutiva

As substituições não destrutivas retornam a string substituída pela expressão regular sem alterar a string original. Veja um exemplo bem simples:

```
  1 #exemplo22.5.pl
  2 use 5.12.0;
  3
  4 my $nome_de_solteira = "Luciana Motão";
  5 my $nome_de_casada = $nome_de_solteira =~ s/Motão/
Sanchez/r;
  6
  7 say "O nome de solteira era $nome_de_solteira";
  8 say "O novo nome é $nome_de_casada";

saída:

O nome de solteira era Luciana Motão
O novo nome é Luciana Sanchez
```

O modificador ***r*** realiza a substituição não destrutiva, mantendo o valor original em ***$nome_de_solteira*** e retornando a nova string que foi armazenada em ***$nome_de_casada***. Repare que a substituição não destrutiva é uma característi-

ca do Perl 5.14, mas foi ativada com o pragma ***use 5.12.0***. Isso porque o Perl 5.14 é cem por cento compatível com o Perl 5.12, e, ao ativar o 5.12, eu também ativo o 5.14 se ele estiver instalado no sistema, já que o 5.14 não irá prejudicar nenhum programa feito com seu antecessor. Logo, você comprou um livro de Perl 5.12 e de quebra aprendeu Perl 5.14.

Despedida

Encerro aqui a nossa jornada pelo universo do Perl5. Espero siceramente que tenha gostado de ler este livro tanto quanto eu gostei de escrever. Não se esqueça de ir além do livro e ler as documentações do Perl e os livros recomendados no decorrer do texto. Lembre-se de que bons programadores devem amar a programação: "profissionais construíram o Titanic, mas um amador, chamado Noé, construiu uma arca que sobreviveu ao dilúvio".

Nos vemos nas comunidades de software aberto Brasil afora. Abraços.

Programando com

Perl

O Perl nasceu no final da década de oitenta pelas mãos de Larry Wall como uma ferramenta para facilitar seus trabalhos com administração de servidores. A linguagem conquistou diversos programadores pelo mundo com a filosofia de manter coisas simples de forma simples e tornar coisas difíceis possíveis. Hoje a linguagem conta com mais de vinte mil módulos que ampliam as funcionalidades e características básicas da linguagem, utilizada atualmente em projetos de internet, administração de sistemas e em ambientes com exigências de alta disponibilidade.

Neste livro a linguagem Perl é abordada de forma didática, quebrando a crença de que Perl é uma linguagem difícil de aprender. Ela apenas possui características diferentes que são muito interessantes e úteis.

Se você é um programador iniciante, este livro é para você.

Se você é um programador experiente em alguma linguagem, mas não conhece Perl, este livro é para você.

Se você quer aprender a utilizar Perl para administração de sistemas ou aplicativos de linhas de comando, este livro é para você.

Se você quer aprender os princípios básicos da programação para internet, este livro irá ajudá-lo.

Thiago Glauco Sanchez atualmente trabalha com administração de sistemas na Petrobras SA, além de ser instrutor de programação e Linux na TICursos.net. Sua primeira formação profissional foi técnico em eletrônica, o que o levou a conhecer programação de computadores e lógica digital logo aos 16 anos de idade. Hoje é tecnólogo com mais de 12 anos de experiência em Tecnologia da Informação e Comunicações, tendo trabalhado para a Secretaria de Agricultura de São Paulo e HP. Participa de palestras sobre software livre e comunidades de Linux e programação e possui algumas certificações na área.

BRASPORT

www.brasport.com.br