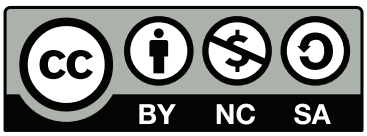
Programação 1

ISBN: a definir

Sobre os autores: André Grégio, Fabiano Silva, Luis Carlos Albini e Marcos Castilho são professores do Departamento de Informática da Universidade Federal do Paraná.



Versão 0.0.1

AVISO: Este texto ainda está em construção.

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Mint* e *Ubuntu*), compilador de texto LaTeX 2$_\varepsilon$, gerenciador de referências *BibTeX*, ...

Versão compilada em 17 de setembro de 2019.

Sumário

Capítulo 1

Introdução

1.1 Introdução ao Curso

Esta disciplina complementa a de Algoritmos e Estruturas de Dados II, ambas ministradas no segundo período do Curso de Bacharelado em Ciência da Computação da UFPR.

Enquanto que os aspectos algoritmicos de mais alto nível são abordados em Algoritmos II, tais como métodos de ordenação, busca, algoritmos recursivos e backtraking, a disciplina de Programação I procura entrar em detalhes intrínsecos da arte de programação.

Neste sentido, usamos a linguagem C como pano de fundo para cobrir aspectos também altamente relevantes na parte algoritmica, tais como alocação dinâmica e tipos abstratos de dados, além de uma breve introdução ao *Shell BASH*, porque acreditamos que programar em C usando *Shell* é mais produtivo para alunos de Ciência da Computação, além do fato de que *Shell* é uma importante ferramenta para Computação.

Apesar de adotarmos a linguagem C, faz parte desta disciplina cobrir as boas práticas de programação, necessárias para uma excelente formação de programadores, para qualquer linguagem de programação.

Este texto foi pensado para conter a nossa maneira de abordar os temas exigidos na emenga da disciplina CI1001 – Programação I, ministrada para o curso de Ciência da Computação da UFPR. Existe farto material na literatura sobre os aspectos aqui abordados, mas pensamos que ter um texto próprio contendo referências para os textos clássicos pode facilitar o estudante em uma primeira leitura.

São excelentes livros consagrados:

- *Shell*: [Newham und Rosenblatt, 2005];
- Linguagem C: [Kernighan, 1990], [Banahan u. a., 1991];
- Algoritmos e Estruturas de Dados: [Leiserson u. a., 2002], [Sedgewick, 1983], [Ziviani, 2007], [Wirth, 1976] e [Tenenbaum u. a., 1995].

1.2 Objetivos

Capacitar o estudante a compreender o modelo de programação dos computadores atuais e adesenvolver programas usando técnicas elementares de algoritmos

e estruturas de dados sobre este modelo. Capacitar o aluno a desenvolver soluções simples e eficazes para problemasdiversos que podem ser resolvidos com as técnicas elementares, sempre considerando a noção de eficiência dos códigos desenvolvidos.

São objetivos específicos:

- Apresentar as características principais de sistemas Unix: comandos, processos, shell. Apresentar
- Comandos básicos do Unix: manipulação de arquivos (ls, chmod, cp, mv, rm, tar), editores de texto (vi, emacs), documentação on-line (man, apropos). Apresentar aspectos adicionais de UNIX: controle de processos (kill, jobs, ps), ambiente shell (status de execução de comandos, variáveis de ambiente, redirecionamento de E/S), comandos paraanálise de conteúdo de arquivos (grep, diff, cut, paste). Combinando comandos simples (pipes).
- Introduzir uma nova linguagem de programação, abordando todos os conceitos vistos emdisciplinas anteriores: estrutura geral de um programa em C, variáveis, estruturas de controle,funções, vetores, strings. O processo de compilação de programas em C, incluindo o uso simplificadoda ferramenta 'make', importante para a geração de programa de médio e grande porte que sãovistos nos semestres seguintes. É fundamental o estudante compreender os bons princípios de programação em C, tais como uso do .h e .c, dentre outros.
- Apresentar os tipos Registro (struct) e Enumeração, com suas aplicações.
- Compreender conceito de ponteiros e sua utilidade de manipulação de variáveis através de seuendereço em memória. Será visto também o processo de criar variáveis dinamicamente durante aexecução de um programa, através do mecanismo de alocação de memória. A relação destes conceitos com vetores, strings, matrizes e registros também é vista.
- Compreender o conceito de Tipos abstratos de Dados e Estruturas de dados fundamentais.
- Estudar a implementação de estruturas de dados na memória principal, com o uso de alocação estática e dinâmica de memória.
- Estudar em profundidade listas lineares, suas especializações e aplicações: listas ordenadas, listas encadeadas, filas e pilhas.
- Implementar as estruturas de dados vistas anteriormente (listas, pilhas e filas) em linguagem C.
- Compreender os métodos básicos de testes de condicionais e repetições, bem como ferramentas de depuração.

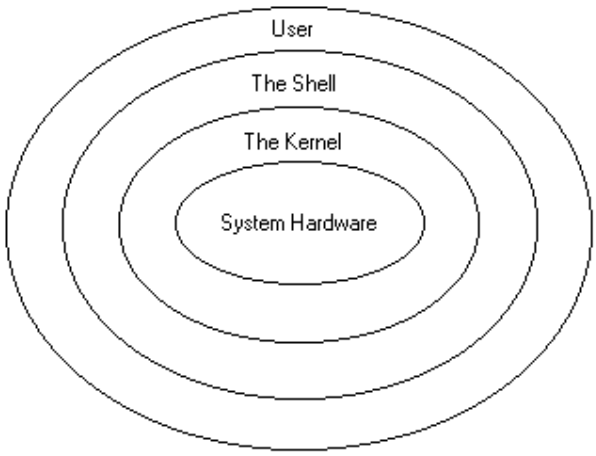
Capítulo 2

Shell

2.1 Introdução

O que é *Shell*?

A *Shell* é, basicamente, a interface entre o usuário e o sistema operacional de um computador.



Como aprender *Shell*?

- Lendo muito;
- Fazendo exercícios;
- Lendo mais;
- Fazendo mais exercícios. . .

É difícil aprender *Shell*?

- O básico é simples;
- Mas um *shell* é repleto de detalhes, para entender todos é preciso prática e interesse;

- O uso da *shell* exige clareza e simplicidade, pequenos programas em *shell* podem parecer grandes enigmas;
- O interesse pode ter a ver com necessidade.

Exemplo de uso da *Shell*

- Suponha que eu queira converter milhares de arquivos de um formato para outro, estes arquivos estão distribuídos por uma hierarquia de diretórios.
- Realizar esta tarefa manualmente é exaustivo, trabalhoso, tedioso e sujeito a erros.
- Em alguns poucos minutos podemos fazer um script que, com poucas linhas de código, realiza esta tarefa para nós automaticamente.
- Escrever este script é prazeiroso, gratificante e o resultado é correto e rápido.

Outro exemplo de uso da *Shell*

- Um administrador de sistemas que tem 2000 usuários precisa verificar se os dados das contas estão consistentes, se todos possuem HOME, NOBACKUP e HTML. Caso falte algum deve ser criado e disponibilizado para o usuário. Além disso, é necessário verificar se a quota do usuário está no padrão e se não estiver, colocar.
- Para cada usuário resulta em uma combinação de 4 ou 5 comandos que pode deixar o administrador do sistema louco e o resultado não será correto.
- Com um script de cerca de 20 linhas que o administrador levou uma hora para fazer a tarefa é feita com perfeição!
- A menos que o programador do script faça uma besteira, daí seria uma catástrofe. . . .

Por quê é importante aprender *Shell*?

- Um bom administrador de sistemas e um bom programador devem conhecer muito bem uma *shell*.
- Serve para automatizar tarefas do quotidiano, melhorando sua eficiência como programador e usuário de uma máquina;
- Serve também para facilitar a criação de protótipos de soluções, que depois podem dar origem à soluções em linguagens de mais alto nível.

A tarefa da *shell*

- A tarefa da *shell* é traduzir as linhas de comando do usuário em instruções do sistema operacional.

Exemplo de uma tarefa do *shell*

- Considere a seguinte linha de comando, que significa: mostre o conteúdo do meu diretório *HOME* e coloque o resultado no arquivo */tmp/saida_ls.txt*.

```
ls -l ~ > /tmp/saida_ls.txt
```

Exemplo de uma tarefa da *shell*

- A *shell* separa a linha em palavras: *ls*, -l, ~, >, /tmp/saida_ls.txt.
- Em seguida determina o propósito de cada palavra: *ls* é um comando, -l é uma opção, ~ é um argumento e > /tmp/saida_ls.txt são instruções de Entrada/Saída.
- Define a Entrada/Saída de acordo com > /tmp/saida_ls.txt.
- Finalmente, encontra o comando *ls* e o executa com a opção -l (formato longo) e com o argumento ~.

Exemplo de uma tarefa da *shell*

- A real execução do comando feita pelo sistema operacional é escondida do usuário;
- Na verdade, cada etapa descrita contém subetapas que também dependem do sistema operacional do sistema particular que roda a *shell*.

Console da IBM anos 1970

Sem mouse!

- Os consoles de antigamente não tinham mouse nem interface gráfica
- A *shell* também servia para facilitar tarefas, recuperar comandos, etc.

- Mas também já servia como uma linguagem de programação poderosa
- O ganho de produtividade sem uso do mouse, para programadores, é imenso!

Textos dos veteranos

Os textos que os veteranos produziram merecem ser lidos, pois:

- Contém uma breve história do shell e sua evolução
- Ensinam a ganhar produtividade sem uso do mouse
- Descrevem o sistema de arquivos do linux
- Explicam alguns comandos básicos e muito úteis
- Mostram a existência de ambientes gráficos apropriados para programação e também como usar múltiplos consoles
- Explicam sobre diversos elementos da shell em linguagem de fácil leitura

2.2 Elementos básicos

- comando argumentos
  - a primeira palavra é o comando a ser executado
  - o restante são os argumentos, isto é, como e em que o comando vai operar

Exemplos

- `mail joao`
  - o comando `mail` vai enviar um e-mail para o usuário joao
- `lpr -Psecretaria arquivo.pdf`
  - o comando `lpr` vai imprimir o arquivo.pdf na impressora da secretaria

Argumentos

- Argumentos podem ser arquivos ou opções para o comando
- As opções modificam o comportamento do comando
- Exemplos:
  - `ls -a`: mostra também os arquivos escondidos
  - `ls -l`: mostra todas as informações dos arquivos
  - `ls -ltr`: igual ao anterior, mas ordena por data na ordem reversa

Arquivos

- Arquivos regulares
- Arquivos executáveis
- Diretórios

Arquivos regulares

- Qualquer arquivo que possa ser lido, seja por humanos ou por programas
  - arquivos ASCII
  - arquivos PDF
  - arquivos em formato mp3

Arquivos executáveis

- são programas, que podem ser legíveis (ASCII) ou não (binários)
  - bash, lpr, mail, libreoffice, firefox
  - scripts feitos em linguagem shell
  - scripts feitos em linguagem python

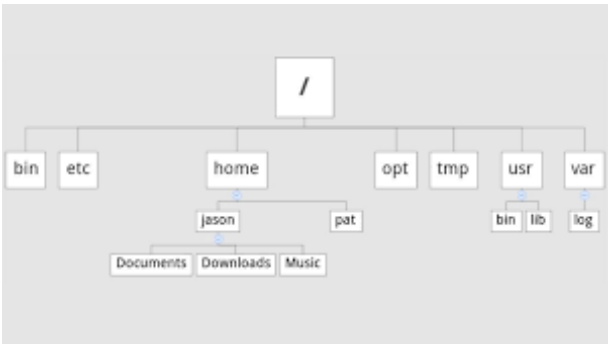
Diretórios

- São arquivos que contém outros arquivos ou outros diretórios (subdiretórios)
  - Seu HOME (~), `~/disciplinas/ci1001`, `~/disciplinas/ci1001/shell`, `~/disciplinas/ci1001/C`
  - `/bin`, `/usr/bin`
  - `/var`, `/var/log`
  - `.`, `..`: são dois diretórios especiais

Hierarquia de diretórios no UNIX

- Existe uma hierarquia de diretórios em forma de árvore, juntamente com um padrão de onde se colocar aquivos nos sistemas UNIX.
  - `/`: raiz
  - `/etc`: arquivos de configuração
  - `/usr`: arquivos da distribuição
  - `/usr/lib`: bibliotecas da distribuição
  - `/usr/bin`: binários da distribuição
  - `/lib`: bibliotecas fundamentais do sistema
  - `/dev`: dispositivos do sistema (discos, consoles, etc)

Desenho de uma árvore de diretórios



Seu diretório pessoal

- Em geral sob `/home`, é o seu diretório de trabalho, onde normalmente você tem permissão de escrever
- o símbolo ~ é usado para abreviar o nome completo do seu diretório
  - `/home/prof/ci1001`
  - `~ci1001`
  - Se o usuário que roda o comando é ci1001: `~`

Navegação entre diretórios

- Comando: `cd`
  - `cd ~`
  - `cd ../bcc/abcd99`
  - `cd ~ci1001/tmp`
  - `cd /etc/ssh`
  - `cd -`

Comandos úteis relativos aos diretórios

- `pushd`: empilha diretório
- `popd+`: desempilha diretório
- `pwd`: imprime nome do diretório atual
  - `PS1`: variável que controla seu *prompt*
  - `PS2`: variável que controla seu *prompt* secundário
  - Em breve aprenderemos a noção de variável, mas é possível alterar seu *prompt* padrão para que fique do seu gosto

Coringas e expansões em nomes de arquivos

- Se existem arquivos em um diretório podemos acessar usando alguns filtros
- Os arquivos existem, mas não sabemos quais nem quantos
  - Coringas
  - Expansão de colchetes

Coringas

- `*`: casa com qualquer sequência de caracteres
- `?`: casa com um único caracter
  - `ls *.c`: expande para todos os arquivos que terminam em `.c`
  - `ls *c`: expande para todos os arquivos que terminam em `c`
  - `ls m*s`: expande para todos os aquivos que começam com m e terminam com s
  - `ls m?s`: expande para todos os aquivos com três letras que começam com m e terminam com s
  - `ls *.*`: expande para todos os arquivos que tem um ponto no nome
  - `echo *`: exibe todos os arquivos não escondidos (que não iniciam com um ponto), isto é, faz uma espécie de `ls`

Observação importante

- O que a *shell* faz é *expandir* os nomes para o comando executar
- Suponha que você tenha os arquivos: `fred, flintstone, wilma, pedrita, barney, betty`
- Executar: `ls f*` faz a shell executar na verdade `ls flintstone fred`
- Executar `ls a*` faz a shell executar `ls a*`, onde `a*` agora é literal, pois não há arquivos iniciando com `a`, o que implica em retorno de erro do comando `ls`

Expansão de colchetes

- `[set]`: expande para todos símbolos em *set*
- `[!set]`: expande para todos símbolos que *não estão* em *set*
  - `ls [xyz]`: x, y ou z
  - `ls [xyz]*`: qualquer arquivo que inicia com x, y ou z
  - `ls [a-z]`: todas as letras minúsculas
  - `ls -d /home/bcc/[a-z]*19`: mostra os diretórios dos calouros que entraram em 2019
  - `ls -d /home/bcc/[!a-cr-z]*19`: mostra os diretórios dos calouros que entraram em 2019 e cujo login inicia com letras entre `d` e `q`

Expansão de chaves

- Diferente de coringas, que expandem com nomes de arquivos que existem, a expansão de chaves expande para qualquer string da forma especificada.
  - `echo mari{a,o}`: imprime: `maria mario`
  - `mkdir alg{1,2,3}-aulas`: cria os diretórios alg1-aulas, alg2-aulas, alg3-aulas
  - `echo a{r{0,U},ra}s`: `ar0s arUs aras`
  - `mv arquivo.{c,backup}`: Renomeia `arquivo.c` para `arquivo.backp`

Entrada e saída no UNIX

- Ideia simples e brilhante com enorme implicação
  - Entrada e saída padrão
  - Redirecionamento de entrada e saída
  - Pipelines

Entrada e saída padrão

- Entrada padrão: por default é o teclado
- Saída padrão: por default é o monitor de vídeo
- Saída padrão de erros: por default é o monitor de vídeo

Exemplo

O comando `cat` lê linhas da entrada padrão e imprime na saída padrão até que seja digitado um `^D` (control-D).

```
ci1001@fradim:~/tmp$ cat
Alo mamae!
Alo mamae!
Hello world
Hello world
^D
ci1001@fradim:~/tmp$
```

Redirecionamento de entrada e saída

- <: redireciona a entrada padrão
- >: redireciona a saída padrão
- 2>: redireciona a saída padrão de erros
- &>: redireciona a entrada e a saída padrão de erros
- >>: redireciona a saída padrão de erros, mas anexa, ao invéz de sobrescrever

Exemplo

Considere que existe um arquivo cujo conteúdo são duas linhas: Alo mamae! e Hello world:

```
ci1001@fradim:~/tmp$ cat arquivo.txt
Alo mamae!
Hello world
ci1001@fradim:~/tmp$
```

Podemos copiar este arquivo em outro sem usar o comando `cp`

```
ci1001@fradim:~/tmp$ cat < arquivo.txt > arquivo.copia
ci1001@fradim:~/tmp$ cat arquivo.copia
Alo mamae!
Hello world
ci1001@fradim:~/tmp$
```

Um exemplo bacana

Aqui tentamos listar três arquivos, um deles não existe e o `ls` acusa erro na tela:

```
ci1001@fradim:~/tmp$ ls teste{,.pub,.old}
ls: não é possível acessar 'teste.old': Arquivo ou diretório não encontrado
teste  teste.pub
```

Aqui direcionamos a saída padrão de erros para um arquivo erro.txt. O `ls` exibe os arquivos existentes e cria um novo arquivo contendo a mensagem de erro.

```
ci1001@fradim:~/tmp$ ls teste{,.pub,.old} 2> erro.txt
teste  teste.pub
ci1001@fradim:~/tmp$ cat erro.txt
ls: não é possível acessar 'teste.old': Arquivo ou diretório não encontrado
```

Pipelines

- Um *pipe*, denotado `|`, pode ser traduzido como *tubo*
- É uma maneira de ligar a saída de um programa para a entrada padrão de outro programa.
- Dois ou mais programas conectados por *pipes* é um *pipeline*
- É um conceito muito elegante no UNIX, permite, por exemplo, aplicação de filtros variados até produzir a saída desejada pelo usuário.
- Um exemplo para três comandos: `comando1 | comando2 | comando3`

Exemplo

- `grep` é um programa que filtra linhas de arquivos
- `wc` é um programa que conta linhas de um arquivo quando usada a opção `-l`
- Suponha que queiramos contar quantas funções foram definidas em um programa em *Pascal*.

Exemplo

O *grep* imprime apenas as linhas que contém a palavra function no início de uma linha.

```
aula28$ grep ^function floodfill.pas
function ler_cor (jogo: tipo_jogo): integer;
function acabou (jogo: tipo_jogo): integer;
function inunda_vizinho (var jogo: tipo_jogo;cor_velha,cor_nova,x,y: integer): boole
function testa_vitoria (jogo: tipo_jogo): boolean;
function distancia (e1,e2: elemento): real;
function sorteia_cor (jogo: tipo_jogo;cor_velha: integer): integer;
function escolhe_cor (jogo: tipo_jogo): integer;
aula28$
```

Exemplo

Poderíamos direcionar esta saída para um arquivo e depois usar o *wc*:

```
aula28$ grep ^function floodfill.pas > saida.txt
aula28$ wc -l saida.txt
7 saida.txt
```

Exemplo

É desnecessário ter que criar o arquivo temporário, basta usar o *pipe* logo após a saída do *grep*:

```
aula28$ grep ^function floodfill.pas | wc -l
7
```

Outro exemplo

- O comando `cut` seleciona colunas em um arquivo
- O comando `sort` ordena arquivos
- Suponha que queiramos imprimir os nomes completos, de forma ordenada, que aparecem na segunda coluna deste arquivo:

  ```
  fulano: Fulano de tal: 1968
  beltrano: Beltrano da silva: 1934
  sicrano: Sicrano de Souza: 1945
  ```

Solução

Usamos o *cut* com as opções `-d:` (define o separador como `:`) e `-f2` (imprime a segunda coluna), mas a saída está fora de ordem.

```
ci1001@fradim:~/tmp$ cut -d: -f2 dados.txt
 Fulano de tal
 Beltrano da silva
 Sicrano de Souza
```

Ligando esta saída com um *pipe* para o comando *sort*:

```
ci1001@fradim:~/tmp$ cut -d: -f2 dados.txt | sort
 Beltrano da silva
 Fulano de tal
 Sicrano de Souza
```

Background e *foreground*

- O UNIX é um sistema multitarefas, vários *jobs* podem rodar ao mesmo tempo
- Existem processos simultâneos de muitos usuários, inclusive vários deles são do seu usuário
- Os processos podem rodar mesmo você não estando logado

Background e *foreground*

- Quando você digita um comando, seu *prompt* fica preso até que o comando termine
- Este *job* está rodando em *foreground*
- Você pode colocá-lo em *background* colocando um `&` após o comando
- Isto vai liberar seu *prompt* para você digitar outros comandos enquando o primeiro roda.

Exemplo

Neste exemplo o libreoffice foi chamado em *background*, o *prompt* é liberado. A *shell* mostra o número do processo que está rodando, no caso 4252. O comando *jobs* mostra todos os seus processos que estão em *background*. Quando o processo termina a *shell* avisa.

```
ci1001@fradim:~/tmp$ libreoffice &
[1] 4252
ci1001@fradim:~/tmp$ <prompt liberado>
ci1001@fradim:~/tmp$ jobs
[1]+  Executando              libreoffice &
ci1001@fradim:~/tmp$
[1]+  Concluído              libreoffice
ci1001@fradim:~/tmp$
```

Entrada e saída em *background*

- Não se deve ter E/S com *jobs* em *background*
- Se tiver entrada, o *job* vai ficar esperando
- Se tiver saída, sua *shell* ficará poluída com a saída e vai te atrapalhar
- Por isso é aconselhavel usar redirecionamento de E/S nestes casos

```
$ comando < entrada_comando > saida_comando
```

Prioridade de processos

- Todo processo no Linux tem uma prioridade definida pelo Sistema Operacional.
- O usuário pode alterar esta prioridade com o comando *nice*, como forma de boa educação quando tem um processo demorado para não atrapalhar outros usuários.
- Os comandos *top* ou *ps* permitem ver a prioridade atual dos processos

Símbolos especiais

- A *shell* tem símbolos com significado especial, abaixo segue a lista deles
- Se quisermos usar o símbolo propriamente dito é preciso fazer o que é chamado *quoting*, que significa proteger o símbolo
- Isto pode ser feito usando-se ', ou " ou \

Exemplo

- `echo 2 * 3 > 5 eh verdadeiro`
- Você vai criar um arquivo cujo nome é 5 e cujo conteúdo é 2 seguido de todos os arquivos no seu diretório seguido da string 3 eh verdadeiro
- Isto porque o `> 5` direciona a saída para o arquivo de nome 5, e o `*` expande para todos os seus arquivos.
- Algumas maneiras de fazer isso corretamente:
- `echo 2 \* 3 \> 5 eh verdadeiro`
- `echo 2 '*' 3 '>' 5 eh verdadeiro`
- `echo 2 "*" 3 ">" 5 eh verdadeiro`

Observação

- Fazer *quoting* com ' é diferente de fazer com "
- Veremos isso em outra parte deste curso
- ' é chamado *quote forte*
- " é chamado *quote fraco*

Lista de símbolos especiais

~	diretório HOME
‘	substituição de comando
#	comentário
$	expressões variáveis
&	coloca processo em *background*
*	coringa
(	inicia *subshell*
)	encerra *subshell*
\	faz *quote*
|	pipe
[	inicia expansão por conjunto
]	encerra expansão por conjunto

Lista de símbolos especiais

{	inicia comando de bloco
}	encerra comando de bloco
;	separador de comandos
'	*quote* forte
"	*quote* fraco
<	redireciona entrada padrão
>	redireciona saída padrão
/	separador de *pathname*
?	coringa de um caracter
!	NOT lógico

Teclas de controle

- Algumas teclas de controle podem ser usadas para ajudar a controlar processos
- A lista padrão está no próximo slide
- Pode-se modificar estas teclas

Lista de teclas de controle

CTRL-C	interrompe o comando atual
CTRL-D	fim da entrada
CTRL-\	interrompe o comando atual
	se CTRL-C não funcionar
CTRL-S	interrompe saída na tela
CTRL-Q	restaura saída na tela
DEL ou CTRL-?	apaga caracter
CTRL-U	apaga toda a linha de comando
CTRL-Z	suspende o processo
CTRL-L	limpa a tela

Comandos básicos úteis para iniciantes

Comando	Breve explicação
man	interface para os manuais on-line
bc	linguagem que suporta uma calculadora
cat	concatena arquivos e imprime na saída padrão
cp	copia arquivos e diretórios
date	imprime ou define a data/hora do sistema
diff	compara arquivos linha por linha
file	determina o tipo do arquivo
find	procura arquivos na hierarquia de diretórios

Comandos básicos úteis para iniciantes

finger	exibe informações completas de um login name
grep	imprime linhas que casam com um padrão
head	imprime a primeira parte de um arquivo
less	mostra um arquivo com pausa
mv	renomeia um arquivo
sort	ordena as linhas de arquivos texto
tail	imprime a parte final de arquivos
touch	cria arquivo ou altera timestamps se existe
wc	imprime número de linhas, palavras e caracteres

Comandos básicos úteis para iniciantes

jobs	mostra o status dos seus jobs
kill	envia um sinal para um processo
nice	modifica a prioridade de um processo
nohup	roda um comando imune a *hangups* com saída para um arquivo
ps	reporta o estado dos processos correntes
top	mostra processos do Linux

Alguns comandos builtin

help	Exibe informações sobre comandos builtin
alias	Define ou mostra apelidos
bg	Coloca job em *background*
cd	Muda de diretório
echo	Imprime argumentos na saída padrão
history	Exibe ou manipula o histórico de comandos
kill	Envia um sinal a um job
type	Exibe informações sobre o tipo do comando

2.3 Exercícios

1. Uma família está para receber um bebê em casa e quer naturalmente escolher um bom nome para ele ou ela. Você vai ajudar esta família nesta tarefa, produzindo uma lista de nomes (não vamos nos preocupar com os sobrenomes). Você sabe que existem cerca de 400 alunos no curso de Ciência da Computação e que todos eles têm contas no sistema computacional do DInf. No diretório `/home/bcc` você encontra os *login names* de todos eles. O desafio deste problema é:

   - usando o comando *man*, aprender a usar os comandos *finger*, *cut*, *grep* e *sort*;
   - usando opções apropriadas destes comandos e tudo o que você aprendeu na aula anterior, obter um arquivo de nome *nomes_de_informatas.txt* que contém uma lista ordenada e sem repetição de todos os nomes (prénomes) dos usuários que tem conta sob o diretório `/home/bcc`.
   - criar um arquivo de nome *acha_nomes.sh* contendo os comandos executados para obter a lista de nomes solicitada acima. Este é o arquivo que deve ser entregue.

2. Uma boa maneira de aprender a programar em *C* é refazer todos os programas que foram feitos em *Pascal*, pois desta maneira o estudante se concentra na novidade, que é a sintaxe da linguagem *C*. O desafio deste problema é:

- eu gostaria de localizar todos os meus programas feitos em *Pascal* no primeiro período e fazer uma cópia deles em um único diretório de nome `Pascal2C`. O problema é que não lembro onde eu os escrevi e eles provavelmente estão espalhados;
- Crie este diretório e, usando o comando *find* integrado com o comando *cp*, faça uma cópia de todos os arquivos que tem extensão `.pas` que estão sob seu diretório HOME neste novo diretório.
- Usando o comando *wc* e mais algumas coisas, descubra quantos arquivos foram copiados.
- criar um arquivo de nome *copia_pascal.sh* contendo os comandos executados para resolver os dois problemas acima. Este é o arquivo que deve ser entregue.

2.4 Shell básico

O básico da *shell* envolve as seguintes noções fundamentais:

- caracteres especiais
- entrada e saida, redirecionamento e pipes
- substituições (expansões)
- background e foreground e controle de processos
- edição de linhas de comando
- custumização do ambiente (.bashrc, etc)
- variaveis
- arquivos, permissão, etc
- Status de saída
- Scripts e Funções
- Variáveis da *shell*
- Comandos

Alguns destes tópicos já foram explicados, veremos o restante.

Edição de linhas de comando

- A *shell* possui mecanismos para agilizar o trabalho do usuário, tais como navegar na linha de comando para corrigir algo, recuperar comandos previamente digitados (*history*), etc.
- Estes recursos estão muito bem explicados na literatura, recomendamos o livro *Learning the bash shell*, capítulo 2.

Customização do ambiente de trabalho

- A *shell* de cada usuário vem configurada pelo administrador do sistema, mas pode ser alterada pelo usuário;
- Os arquivos importantes são:
  - *.bash_profile*, *.bash_logout* e *.bashrc*;
- Estes recursos estão muito bem explicados na literatura, recomendamos o livro *Learning the bash shell*, capítulo 3.
- Muito do que pode ser feito nesta customização depende do bom entendimento do uso de variáveis da *shell*, que explicaremos a seguir.

Variáveis da *shell*

- Variáveis na *shell*
- Variáveis de ambiente (*builtin*)
- Convenção: nomes de variáveis sempre em maiúsculas

Variáveis na *shell*

- Declaração: `NOME=valor`
  - Exemplo1: `USER=fulano`
  - Exemplo2: `USERNAME="Fulano de Tal"`
- Para usar o valor da variável: preceder o nome da variável com `$`
  - Exemplo1: `echo $USER`
  - Exemplo2: `echo ${USERNAME}`
  - Exemplo3: `echo O nome do ${USER} eh ${USERNAME}`
- Para deletar uma variável: `unset NOME`
- Toda variável que não existe *shell* trata como *string* vazia

Quoting de variáveis

- Pode ser usado com aspas simples (*quote*) forte ou duplas (*quote fraco*)
- echo "$USER"
  - Imprime: `fulano`
- echo '$USER'
  - Imprime: `$USER`

Detalhe

- `TESTE="string com espacos"`
- `echo $TESTE`
  - Imprime: `string com espacos`
- `echo "$TESTE"`
  - Imprime: `string com espacos`
- A *shell* trata este *quote* como uma palavra única

Variáveis de ambiente

- São variáveis que a *shell* usa para controlar várias coisas:
  - comportamento do *history*, *prompt*, recebimento de email, localização de comandos
- Uma de especial interesse é a variável `PATH`
  - Exemplo: `echo $PATH`
  - Imprime: `/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/home/ci1001/bin`

Subprocessos

- Quando se digita um comando é aberto uma *subshell*
- Ela não necessariamente conhece todas as variáveis
- As variáveis de ambiente são conhecidas
  - `export -p` mostra as que são de ambiente
- `export` com argumentos torna a variável
  - `export TESTE`
  - `export TESTE="alguma coisa"`
- Dica:
  - `PS1="\W > "`
  - Gostou? Então: `echo 'PS1="\W > "' >> ~/.bash_profile`

Permissões

- Todo arquivo no UNIX, incluindo diretórios, tem permissão de acesso para o dono, o grupo a qual pertence o dono e para outros usuários
- O comando *chmod* modifica a permissão padrão
- O comando *umask* define a máscara para a permissão padrão do usuário
- O comando *ls -l* mostra as permissões atuais do arquvi

```
drwxrwxrwx <- o "d" indica diretório
-rwxrwxrwx <- o "-" indica arquivo normal
 ^^^^^^^^^
 |||||||||
 ||||||+++ <- permissão para outros usuários
 |||+++    <- permissão para o grupo
 +++       <- permissão para o dono
```

- r: permissão de leitura
- w: permissão de escrita
- x: permissão de execução
- X, s, t: permissões menos conhecidas, ver manual do *chmod*

Permissões para diretórios

- r: permite ver o conteúdo do diretório
- w: permite criar arquivos no diretório
- x: permite executar comandos no diretório

chmod

- O comando *chmod* permite alterar a permissão:
  - `chmod g-rwx arquivo`
  - `chmod 700 arquivo`

Status de saída

- Todo comando executado produz (de maneira invisível para o usuário) um *status de saída*
- É um valor inteiro (entre 0 e 255) que todo comando retorna para a *shell* e indica o sucesso ou insucesso do comando.
- Usualmente, 0 significa OK e qualquer valor entre 1 e 255 significa que algo deu errado

- A variável especial ? contém o status de saída do último comando executado
- Sem este conceito não é possível programar em *shell* algo de muito relevante, por exemplo, controlar o fluxo de execução do script

Exemplo

```
ci1001@mumm:~$ ls
Aulas  Desktop  shell.txt  tmp  web  work
ci1001@mumm:~$ echo $?
0
ci1001@mumm:~$ ls /root
ls: não foi possível abrir o diretório '/root': Permissão negada
ci1001@mumm:~$ echo $?
2
ci1001@mumm:~$
```

Scripts e Funções

- Um script é um arquivo ASCII contendo comandos da *shell* que pode ser executado provocando a execução dos comandos que ele contém, em suma, é um programa escrito em *shell*
- Uma função é um subprograma que é armazenado em memória e pode ser invocado posteriomente por uma *shell* ou por outra função

Exemplo de script

```
ci1001@fradim:~/tmp$ cat exemplo
     mkdir temp
     chmod 700 temp
ci1001@fradim:~/tmp$ ls -l exemplo
-rw-r--r-- 1 ci1001 especial 36 Ago  1 16:37 exemplo
ci1001@fradim:~/tmp$
```

- Ao executar o script assim: `source exemplo` será executado o comando `mkdir temp` seguido do comando `chmod 700 temp`

Colocando permissão de execução

```
\item \verb-chmod u+x exemplo-: seta permissão de execução para o arquivo
ci1001@fradim:~/tmp$ ls -l exemplo
-rwxr--r-- 1 ci1001 especial 36 Ago  1 16:37 exemplo
```

- Agora podemos digitar: `./exemplo`, que os comandos *mkdir* e *chmod* serão executados em uma *subshell*

Hashbang ou *Shebang*

```
ci1001@fradim:~/tmp$ cat exemplo
#!/bin/bash
     mkdir temp
     chmod 700 temp
ci1001@fradim:~/tmp$ ls -l exemplo
```

- Se quisermos garantir que o script seja executado em *bash*
- O *hashbang* especifica em qual interpretador os comandos serão executados, por exemplo, um script em *zsh* teriam a primeira linha assim:
- `#!/bin/zsh`, mas os comandos teriam que ser escritos na sintaxe do *zsh*

Funções

- Seu uso é fundamental, assim como funções e procedures em *Pascal*
- Além disso, ficam residentes em memória
- Existem duas maneiras de serem declaradas
- Para remover uma função, usar `unset -f`
- Funções não rodam em processos separados

Declarando função, modo 1

```
function cria_diretorio_temp
{
     mkdir temp
     chmod 700 temp
}
```

Declarando função, modo 2

```
cria_diretorio_temp ()
{
     mkdir temp
     chmod 700 temp
}
```

Precedência para execução de comandos

- Quando se executa um comando, a ordem de procura é a seguinte:
  1. aliases
  2. palavras chave (ex. function) ou comandos de controle de fluxo (if, for)
  3. funções

4. *builtins*
5. scripts e programas executáveis, segundo a variável *PATH*

- o *builtin* `type` mostra o que é associado a um certo nome

```
ci1001@fradim:~/tmp$ type -all echo
echo é um comando interno do interpretador
echo é /bin/echo
ci1001@fradim:~/tmp$
```

Variáveis da shell

- Parâmetros posicionais
- Parâmetros posicionais em funções
- Variáveis locais em funções
- Operadores de string
- Substituição de comandos

Parâmetros posicionais

- Contém os argumentos digitados na linha de comando
- Seus nomes são: `1, 2, 3, ...`
- Seus conteúdos são acessíveis assim: `$1, $2, $3, ...`
- `$0` contém o nome do script sendo executado

```
ci1001@fradim:~/tmp$ cat exemplo
#!/bin/bash
     mkdir $1
     chmod $2 $1
ci1001@fradim:~/tmp$ ./exemplo temp 700
```

Padronização

```
ci1001@fradim:~/tmp$ cat exemplo
#!/bin/bash
DIR=$1
PERM=$2
     mkdir ${DIR}
     chmod ${PERM} ${DIR}
```

Variáveis posicionais especiais

- `@` e *: contém todas as variáveis posicionais, menos `$0`
  - `"$*"`: string única que contém todos os parâmetros
  - `"$@"`: igual a `"$1" "$2" "$3" ... "$N"`
- `$#` contém o número de argumentos passados ao script

Parâmetros posicionais em funções

- Funciona do mesmo modo
- Existe uma hierarquia de parâmetros se usarmos os mesmos nomes, tal como em linguagem *Pascal*
- É possível definir variáveis locais
- O uso das chaves (`{}`) ajuda bastante ao escrever scripts, em especial, para parâmetros posicionais a partir do 9

Operadores de string

- A *shell* permite uma série de operações sobre as variáveis
- Elas são muito úteis em algumas situações e permitem manipulações extremamente elegantes.
- Não entraremos em detalhes, mas elas permitem:
  - Assegurar que as variáveis existem
  - Setar um valor default para as variáveis
  - Prevenir erros que resultam de variáveis não definidas
  - Remover porções dos valores das variáveis segundo algum padrão
- Mais detalhes no capítulo 4 do livro *Learning the bash shell*

Substituição de comandos

- Este tipo de expansão é muito útil e seu uso bastante frequente
- Podemos atribuir como valor de uma variável a saída de um comando!
- `$(comando)`
- Exemplo:
  - `$(ls /home/bcc)` tem como valor todos os nomes de diretórios que estão em `/home/bcc`
  - `$(who | cut -d" " -f 1 | sort -u)` tem como valor os nomes de todos os usuários logados na máquina
  - Assim podemos por exemplo mandar um email para todos estes usuários: `mail $(ls /home/bcc)`

Um comando diferente (e chique!)

- Existe um comando *builtin* na *shell* que é diferente, meio estranho até, mas muito útil
- É o comando com a construção seguinte:
- `[ expr ]`
- Este comando retorna 0 (true) ou 1 (false) dependendo da avaliação da expressão condicional *expr*

Exemplo de uso

- `[ -a arquivo ]`: testa se *arquivo* existe
- `[ -d arquivo ]`: testa se *arquivo* existe e é diretório
- `[ -f arquivo ]`: testa se *arquivo* existe e é um arquivo regular
- `[ -h arquivo ]`: testa se *arquivo* existe e é um link simbólico

Uso do comando

- Ele é usado normalmente em combinação com outros comandos da *shell* que serão melhor explicados na próxima aula.
- Usado sozinho não faz muito sentido
- Veremos hoje um tipo de uso, mas é preciso conhecer um outro conceito antes

Listas

- Uma lista é uma sequência de um ou mais pipelines separados por um dos operadores seguintes:
- `;`, `&`, `&&`, `||`
- e opcionalmente terminado por um dentre os seguintes:
- `;`, `&`, <newline>

Listas

- `&&` é um *AND*
  - `comando1 && comando2`
  - *comando2* só executa se *comando1* retornou status de saída zero
- `||` é um *OR*
  - `comando1 || comando2`
  - *comando2* só executa se *comando1* retornou status de saída diferente de zero

Exemplos

- `[ -d arquivo ] && echo OK || echo "NOT OK"`
- Se *arquivo* existe imprime na tela `OK`, senão imprime `NOT OK`
- Funciona como uma espécie de *if then else*

2.5 Exercícios

1. Escreva um script que faz backup de sí mesmo, isto é, se copia em um arquivo de mesmo nome do seu script, mas com extensão `.bkp` no lugar de `.sh`. Exemplo: seu script se chama `script.sh` você deve criar uma cópia dele chamada `script.bkp`. Dica: use o comando *cat* e os parâmetros posicionais apropriados. Para lidar com o nome do arquivo tem muitas maneiras de fazer isso na *shell*:
   - com o comando `basename` (simples e eficaz).
   - usando a construção com variáveis entre chaves: `${NOME}` com operadores de variáveis apropriados (extremamente elegante).
   - se encontrar outras, use a vontade.
2. Crie uma variável que contenha todos os arquivos do seu diretório home (`/home/bcc/seu_login`) que tenham sido modificados nas últimas 24 horas. Dica use o comando *find* com a opção correta. A substituição de comandos faz o restante.
3. `tar` é um programa para armazenar vários arquivos em um único, este é conhecido como *tarball*. O nome *tar* é vem de *Tape ARquive*. Em sua origem era utilizado para escrever dados em dispositivos de entrada e saída sequenciais, como por exemplo, um dispositivo de fita. O uso básico era tipicamente para fazer backup de arquivos. O interessante é que ele também pode ser usado para se armazenar o *tarball* em um disco rígido. No modo básico o *tar* aglomera os arquivos desejados sem comprimí-los, mas mantendo informações importantes do *filesystem* tais como nome, datas (*time stamp*), dono, permissões de acesso e a própria organização dos diretórios. Por exemplo, você pode fazer um backup da sua área HOME contendo todos os seus arquivos em um único arquivo de nome `meubackup.tar`. Usando opções apropriadas ele também permite que seus arquivos possam ser comprimidos usando compressores como por exemplo o *gzip* ou o *bzip*, e você terá então um único arquivo de nome `meubackup.tar.gz`. Este último pode ser copiado para outra máquina ou, dependendo do tamanho, até ser mandado como um anexo de email. Neste exercício, você vai criar um script de nome `backup.sh`, que conterá comandos para você criar um backup de toda a sua área home. Você deverá fazer o seguinte:
   - criar um *tarball* de toda a sua área home, menos os arquivos que iniciam com o padrão `.[a-z]*`;
   - copie este arquivo no diretório `/nobackup/bcc/seu_login/meubackup.tar.gz`;
   - faça um `cd` para o diretório `/nobackup/bcc/seu_login`;

- restaure o seu backup neste diretório a partir do arquivo; *tarball* que você copiou.

Confira que você tem uma cópia exata do seu diretório HOME no diretório `/nobackup`.

Obs.: Pode ser curioso você colocar um backup em um diretório sob `/nobackup`, mas é um diretório que vocês podem usar para colocar coisas temporárias e que não atrapalham o sistema de backup do DInf. Como é um exercício, e também porque você não quer estourar sua quota, sugerimos este lugar para conter temporariamente o resultado deste exercício. Depois pode ser apagado. Cuidado com o comando `rm -rf`. Se você não usar direito vai perder tudo que tem. Lembre-se que o diretório `nobackup` não tem backup! Por exemplo, este comando apaga tudo do seu diretório HOME: `rm -rf ~` e você **não quer** executá-lo.

2.6 Programação em shell

Controle de fluxo

- Desvios
  - *if/else*
  - *case*
- Laços
  - 2 tipos de *for*
  - *while*
  - *until*
- Selecão
  - *select*

IF/ELSE

- Permite executar uma sequência de comandos dependendo da avaliação de *condições*
- A *shell* possui um grande conjunto de testes *builtin*
- A sintaxe do *if/else* é assim:

```
if condicao
then
    comando
elif condicao  # elif eh opcional mas pode haver varios dele
then
    comando
else
    comando
fi
```

IF/ELSE

- Se quiser economizar linhas, deve-se colocar alguns ;

```
if condicao ; then
    comando
elif condicao ; then
    comando
else
    comando
fi
```

Status de saída e return

- Diferentemente das linguagens convencionais, a "condição" é uma lista de comandos e não uma expressão booleana
- O que é testado é o *status de saída* do último comando da lista de comandos
- O valor zero (0) é considerado "verdadeiro" (OK), qualquer outro valor entre 1 e 255 é considerado "falso" (erro)
- O mesmo vale para a construção `elif`

Exemplo

```
filename=$1
word1=$2
word2=$3
if grep $word1 $filename || grep $word2 $filename; then
    echo "$word1 or $word2 is in $filename."
fi
```

Exemplo 2

```
filename=$1
word1=$2
word2=$3
if grep $word1 $filename && grep $word2 $filename; then
    echo "$word1 and $word2 are both in $filename."
fi
```

Exemplo 3

- Queremos fazer uma função para fazer backup do diretório `~/Prog1` se ele existir. O destino do backup vem no primeiro parâmetro posicional.

```
backup ()
{
    DESTINO=$1
    if [ -d ~/Prog1 ] && [ -d $DESTINO ]; then
        rsync -av ~/Prog1 $DESTINO
    else
        if [ ! -d ~/Prog1 ]; then
            echo "~/Prog1 nao existe"
        fi
        if [ ! -d $DESTINO ]; then
            echo $DESTINO nao existe
        fi
    fi
}
```

Exemplo 2

- Queremos implementar o comando pushd

```
pushd ()
{
    DIRNAME=$1
    if cd ${DIRNAME:?"missing directory name."}
    then
        DIR_STACK="$DIRNAME ${DIR_STACK:-$PWD}"
        echo $DIR_STACK
    else
        echo still in $PWD.
    fi
}
```

Melhorando as soluções dos exercícios da aula 2

```
ci1001@mumm:~$ cat ../aula2/acha_nomes.sh
#!/bin/bash
    DESTINO=${OLDPWD}
    if [ -w $DESTINO ] ; then       # testa se tem perm de escrita
        pushd /home/bcc &> /dev/null
        finger * | grep Name: | cut -d: -f3 | \\ # \\ quebra linha
            cut -d\  -f2 | sort -u > $DESTINO/nomes_de_informatas.txt
        popd &> /dev/null
    else
        echo ERRO: nao tem permissao de escrita em $DESTINO
```

Melhorando as soluções dos exercícios da aula 2

```
ci1001@mumm:~$ cat ../aula2/copia_pascal.sh
#!/bin/bash

    if [ ! -d ~/Pascal2C ]; then    # testa se o diretorio ja existe
        mkdir ~/Pascal2C            # como nao existe, ele eh criado
    fi
    find ~ -name "*.pas" -exec \cp '{}' ~/Pascal2C/ \;
```

Usando parâmetros posicionais

- Se você quer garantir que o número correto de parâmetros foi passado

```
ci1001@mumm:~$ cat copia_hd.sh
#!/bin/bash

    if [ "$#" -ne 2 ]; then
        echo "numero incorreto de parâmetros"
        echo 'Uso: $0 <origem> <destino>'
    else
        ORIG=$1
        DEST=$2
        dd if=$ORIG of=$DEST
    fi
ci1001@mumm:~$ ./copia_hd.sh /dev/sda2 /dev/sdb1
```

CASE

- Sintaxe:

```
case expressao in
    padrao1 )
        sentencas ;;
    padrao2 )
        sentencas ;;
esac
```

Exemplo

```
filename=$1
case $filename in
    *.gif ) exit 0 ;;
    *.tga ) tgatoppm $filename > $ppmfile ;;
    *.xpm ) xmptoppm $filename > $ppmfile ;;
    *.pcx ) pcxtoppm $filename > $ppmfile ;;
    *.tif ) tifftoppm $filename > $ppmfile ;;
    *.jpg ) djpeg $filename > $ppmfile ;;
    * ) echo "formato nao reconhecidoa" ; exit 1 ;;
esac
```

FOR

- Permite iterar sobre uma lista fixa de valores, é um pouco diferente do for do *Pascal*
- Não permite especificar um *range* de valores
- Existem algumas maneiras de executar o *for*
  - `for name [in lista]`
  - `for (( expr1 ; expr2 ; expr3 )) ; do list ; done`

Versão 1

- `for name [in lista]`

```
for name [in lista]
do
    sentencas que podem usar $name
done
```

Exemplo

```
IFS=:

for dir in $PATH
do
    ls -ld $dir
done
```

Exemplo 2

```
for user in $(ls -l /home/bcc)
do
    finger $user | grep Name
done
```

Exemplo 3

```
for i in 1 2 3 4 5
do
    mkdir teste_$i
done
```

Versão 2

- `for (( expr1 ; expr2 ; expr3 )) ; do list ; done`

Exemplo

```
for ((i=1 ; i<=10 ; i++))
do
    echo $i
done
```

WHILE

```
while condicao
do
    sentencas
done
```

- *while* é util quando combinado com aritmética de inteiros, entrada e saída de variáveis e processamento de linhas de comando
- *condição* é uma lista de comandos, o status de saída do último é utilizado para determinar a saída do laço
- executa enquanto a condição é verdadeira (zero)

UNTIL

```
until condicao
do
    sentencas
done
```

- mesmo princípio do *while*, mas o teste é ao contrário
- executa até que a condição seja verdadeira (zero)
- ao contrário de *Pascal* o teste é feito no início do laço

Exemplo de WHILE

```
path=$PATH:

while [ $path ]; do
    ls -ld ${path%%:*}
    path=${path#*:}
done
```

- copia `$PATH` e adiciona um :
- faz `ls` no diretório
- remove o primeiro diretório da lista

Exemplo de UNTIL

```
until cp $1 $2; do
    echo 'tentativa de copia falou, esperando...'
    sleep 5
done
```

- tenta copiar um arquivo em outro até dar certo

Mesma coisa com WHILE

```
while ! cp $1 $2; do
    echo 'tentativa de copia falou, esperando...'
    sleep 5
done
```

- Portanto, *until* é desnecessário...

SELECT

- Não tem equivalente em linguagens tradicionais como *Pascal*
- Permite gerar menus de maneira bem simples

```
select nome [in lista]
do
    sentencas que podem usar $nome
done
```

Exemplo

```
#!/bin/bash
echo "Qual sistema operacional você prefere?"

select os in Ubuntu LinuxMint Windows8 Windows7 WindowsXP
do
    case $os in
        "Ubuntu"|"LinuxMint")
```

```
            echo "Eu tambem uso $os!" ;;
        "Windows8" | "Windows10" | "WindowsXP")
            echo "Que tal tentar o Linux?" ;;
        *) echo "Entrada invalida." ; break ;;
    esac
done
```

Saída para este exemplo

```
Qual sistema operacional você prefere?
1) Ubuntu
2) LinuxMint
3) Windows8
4) Windows7
5) WindowsXP
#? 1
Eu tambem uso Ubuntu!
#? 3
Que tal tentar o Linux?
#? 6
Entrada invalida.
```

2.7 Exercícios

1. Script 1: O comando "logger" registra mensagens nos logs do sistema (veja o manual). Escreva um script em shell de nome logger.sh que:
   - Registre no syslog do sistema a mensagem "Teste do comando logger", incluindo o identificador do processo (PID) que gerou esta mensagem;
   - Mostre em qual arquivo de log a mensagem do item "a" foi registrada e qual o PID usado pelo comando logger para registra-la.
2. Script 2: Escreva um script em shell de nome `my_logger.sh` que:
   - Receba como entrada (por argumento) o arquivo `/var/log/kern.log`
   - Receba um diretório passado como argumento pelo usuário do script
   - Crie uma variável que contenha as três categorias de logs a seguir: "info", "warn" e "error"
   - Verifique se o diretório do item "b" existe e, caso não exista, crie-o
   - Obtenha a data atual do sistema no formato "dd-mm-aaaa" (ex.: 27-08-2019)
   - Busque no arquivo do item "a" as linhas de log de cada categoria contida na variável criada no item "c" e as insira em arquivos distintos dentro do diretório do item "d". Os nomes dos arquivos devem ser formados pela concatenação da data obtida no item "e" + "_" + categoria (item "c") + `.log` (por exemplo, todas as linhas de kern.log que contenham logs de alerta devem estar dentro do arquivo `27-08-2019_warn.log`
   - Após criados, os arquivos de log devem ser compactados em um arquivo de nome `logs.tar.gz` no diretório do item "d"

- Caso o arquivo `logs.tar.gz` do item "g" seja criado, seu script deve registrar uma mensagem de sucesso no syslog. Se ao fim do processo, não houver um arquivo `logs.tar.gz` criado, a mensagem registrada no syslog deve ser de erro.

2.8 Shell avançado

Shell avançado

- aritmética
- vetores
- ferramentas (awk, sed, etc)
- Fica como exercício de leitura!
- Capítulo 6 do livro *Learning de bash shell*

Capítulo 3

Pascal → C

Esta aula traz uma breve revisão da linguagem Pascal e introduz a linguagem C, a ser utilizada na implementação dos algoritmos e estruturas de dados vistos no decorrer do curso.

3.0.1 Pascal: Revisão Geral

Tanto Pascal quanto C são linguagens para programação estruturada criadas entre o fim dos anos 60 e início dos anos 70. Ambas são (ou já foram) utilizadas para o desenvolvimento de sistemas operacionais—o Unix foi feito em C, o Macintosh original em Pascal. Embora estejam contidas no mesmo paradigma de programação, há diferenças fundamentais entre elas, dentre as quais pode-se citar:

- tipagem (Pascal é fortemente tipada, C não);
- transparência no uso de apontadores (em C se utiliza apontadores explícitos) e consequente acesso direto à memória;
- impossibilidade na inicialização de variáveis junto com a declaração;
- nível (Pascal é mais próximo da linguagem natural), estruturação rígida e versatilidade.

Versões mais modernas de compiladores para Pascal, como `http://www.freepascal.org`, resolveram alguns problemas encontrados na linguagem ao longo do tempo, incluindo desempenho. A escolha de uma linguagem depende do tipo de trabalho a ser feito (e possíveis exigências) e da familiaridade do programador. Porém, os conceitos aprendidos na disciplina anterior podem ser facilmente "traduzidos" de Pascal para C.

Para revisar o básico de Pascal, apresenta-se a estrutura de um programa na linguagem:

Listing 3.1: Exemplo de estruturação de um programa em Pascal

```
1 program {NOME DO PROGRAMA}
2 uses {BIBLIOTECAS SEPARADAS POR ','}
3 const {BLOCO DE DECLARACAO DE CONSTANTES GLOBAIS}
```

```
4  var {BLOCO DE DECLARACAO DE VARIAVEIS GLOBAIS}
5  function {DECLARACOES DE FUNCAO, SE HOUVER}
6  { VARIAVEIS LOCAIS }
7  begin
8  ...
9  end;
10 procedure {DECLARACOES DE PROCEDIMENTO, SE HOUVER}
11 { VARIAVEIS LOCAIS }
12 begin
13 ...
14 end;
15 begin { INICIO DO BLOCO PRINCIPAL DO PROGRAMA}
16 ...
17 end. { FIM DO BLOCO PRINCIPAL DO PROGRAMA }
```

Em Pascal, todas as declarações (não se pode inicializar variáveis) são feitas no início, funções e procedimentos são blocos de instruções que, respectivamente, retornam ou não um valor ao final de sua execução e os blocos intermediários (entre um `begin` e um `end`) terminam em ';', enquanto que o bloco principal termina em '.'.

Um programa completo escrito em Pascal para, dado um vetor de inteiros, encontrar o elemento que contém o maior número e seu índice, é ilustrado a seguir:

Listing 3.2: Programa em Pascal para encontrar o elemento máximo contido em um vetor

```
1  program CalulaMaximo;
2  uses crt;
3
4  const
5    n   =  10;
6
7  type
8    Vetor = array[1..n] of integer;
9
10 var
11   A             : Vetor = (50, 30, 0, 40, 21, 99, 100, 2, 9, 65);
12   i, numero, maxi : integer;
13
14 (* Dado um vetor de entrada,
15  *    encontra o maior elemento *)
16 function Max(var A : Vetor): integer;
17 var
18   i, Temp : integer;
19 begin
20   Temp := A[1];
21   for i := 2 to n do
22     if Temp < A[i] then
23     begin
24       Temp := A[i];
25       numero := i;
26     end;
27   max := Temp;
28 end;
29
30 begin
```

```
31    writeln('Vetor para busca:');
32    for i := 1 to n do begin
33      writeln(A[i]: 10);
34    end;
35    maxi := Max(A);
36    writeln('Pressione qqer tecla para continuar...');
37    readKey;
38    writeln('Maior elemento = ', maxi);
39    writeln('Indice no vetor = ', numero);
40  end.
```

No exemplo, foi utilizada a *unit* CRT, um módulo para lidar com a linha de comando (console). A variável global "numero" é modificada dentro da função "Max" e utilizada no bloco principal do programa. O retorno dessa função (`max := Temp;`) atualiza uma outra variável global, "maxi", impressa no console durante a execução do bloco principal. A saída desse programa é (duas últimas linhas):

```
Maior elemento = 100
Indice no vetor = 7
```

3.0.2 C: Introdução

Visando a familiarização com a sintaxe básica de C, o programa em Pascal mostrado anteriormente foi convertido para a linguagem:

Listing 3.3: Programa em C para encontrar o elemento máximo contido em um vetor

```
1  /* programa CalulaMaximo */
2  #include<stdio.h>
3
4  #define n 10
5
6  typedef int Vetor[n];
7
8  Vetor A = {50, 30, 0, 40, 21, 99, 100, 2, 9, 65};
9  int i, numero, maxi;
10
11 /* Dado um vetor de entrada, encontra o maior elemento */
12 int Max(Vetor A) {
13   int i, Temp;
14   Temp = A[0];
15   for (i = 1; i < n; i++) {
16     if (Temp < A[i]) {
17       Temp = A[i];
18       numero = i;
19     }
20     maxi = Temp;
21   }
22   return maxi;
23 }
24
```

```
25 int main() {
26   printf("Vetor para busca:\n");
27   for (i = 0; i < n; i++)
28     printf("%10d\n", A[i]);
29   maxi = Max(A);
30   printf("\nPressione ENTER para continuar...\n");
31   getchar();
32   printf("Maior elemento = %d\n", maxi);
33   printf("Indice do vetor = %d\n", numero);
34   return 0;
35 }
```

É possível notar que, com pouco esforço, mapeia-se programas simples escritos em Pascal para C. Conforme se pratica a sintaxe de C, esses pequenos programas ficam mais enxutos, pois não é preciso ficar restrito à estruturação rígida de Pascal.

A seguir, o mesmo programa é alterado para ressaltar outras nuances das linguagens (por exemplo, a leitura de dados a partir de arquivos de texto) e para adicionar a funcionalidade de obtenção também do valor mínimo. A saída desse programa é (duas últimas linhas):

```
Maior elemento = 100
Indice no vetor = 6
```

O programa abaixo (em C na próxima página) é uma modificação feita sobre a primeira versão, desta vez com os números inteiros atribuídos ao vetor em um arquivo lido durante a execução. Considera-se o arquivo "vetor.txt" presente no mesmo diretório de execução do programa, cujo conteúdo (um inteiro por linha) é:

```
50
30
0
40
21
99
100
2
9
65
```

3.0.3 Pascal e C: Leitura de Arquivos de Texto

Listing 3.4: Programa em Pascal para encontrar o elemento máximo usando um arquivo de texto

```
program CalulaMaximo;
const  n      = 10;
type  Vetor = array[1..n] of integer;
var
   A             : Vetor;
   arq           : text;
   i, numero, maxi : integer;

function Max(var A : Vetor): integer;
var  i, Temp : integer;
begin
   Temp := A[1];
   for i := 2 to n do
      if Temp < A[i] then begin
         Temp := A[i];
         numero := i;
      end;
   max := Temp;
end;

begin
   assign(arq,'./vetor.txt');
   reset(arq);
   writeln('Vetor para busca:');
   for i := 1 to n do
   begin
      readln(arq, numero);
      A[i] := numero;
      writeln(A[i]: 10);
   end;
   maxi := Max(A);
   writeln('Maior elemento = ', maxi);
   writeln('Indice no vetor = ', numero);
   close(arq);
end.
```

Listing 3.5: Programa em C para encontrar o elemento máximo usando um arquivo de texto

```
/* programa CalulaMaximo */
#include<stdio.h>

#define n 10

typedef int Vetor[n];

Vetor A = {50, 30, 0, 40, 21, 99, 100, 2, 9, 65};
FILE *arq;
int i, numero, maxi;

```

```
12 /* Dado um vetor de entrada, encontra o maior elemento */
13 int Max(Vetor A) {
14   int i, Temp;
15   Temp = A[0];
16   for (i = 1; i < n; i++) {
17     if (Temp < A[i]) {
18       Temp = A[i];
19       numero = i;
20     }
21   }
22   return Temp;
23 }
24
25 int main() {
26   arq = fopen("vetor.txt", "r");
27   printf("Vetor para busca:\n");
28   for (i = 0; i < n; i++) {
29     fscanf(arq, "%d", &A[i]);
30     printf("%10d\n", A[i]);
31   }
32   maxi = Max(A);
33   printf("Maior elemento = %d\n", maxi);
34   printf("Indice do vetor = %d\n", numero);
35   fclose(arq);
36   return 0;
37 }
```

EXERCÍCIO: Dado um arquivo com os seguintes valores, faça um programa em C que o leia e encontre: o maior número presente no vetor, o menor número **dentre eles** e o menor **positivo**.

```
1
300
-5
-6
13
10
27
-2
8
33
```

Capítulo 4

Linguagem C

4.1 Introdução

TBD.

4.2 Entrada e Saída

Existem diversas funções para receber dados de entrada, cada uma com suas características próprias. A seguir, serão listadas algumas dessas funções, com as respectivas explicações de funcionamento e exemplos.

- `scanf`
- `sscanf`
- `fscanf`
- `getch`
- `getchar`
- `getche`
- `gets`
- `fgets`

4.3 O Laço FOR em C

Summary of Different Ways of Implementing For Loop

Form	Comment
for (i=0 ; i < 10 ; i++) Statement1;	**Single** Statement

for (i=0 ;i <10; i++)
{
Statement1;
Statement2;
Statement3;
}
	Multiple Statements within for

for (i=0 ; i < 10;i++) ;	For Loop with no Body (**Carefully Look at the Semicolon**)
for (i=0,j=0;i<100; i++,j++) Statement1;	**Multiple initialization** & Multiple **Update** Statements Separated by Comma
for (; i<10 ; i++)	**Initialization not used**
for (; i<10 ;)	**Initialization & Update not used**
for (; ;)	**Infinite** Loop,Never Terminates

Fonte: http://www.c4learn.com/c-programming/c-for-loop-types/

Exercícios

1. TBD.

   (a) TBD.

Apêndice A

SSH

- Secure Shell
- Permite a comunicação segura (criptografada) entre duas máquinas
- Essencial para quem quer se logar em outro sistema com segurança

1.0.1 Outros programas da família

- scp: para cópia segura de arquivos remotos
- sftp: para transferência segura de arquivos
- sshfs: para montagem remota do seu HOME (pacote sshfs)

1.0.2 Chaves RSA

- RSA é um protocolo de segurança baseado em uma parte pública e outra privada
- A parte pública pode ser conhecida por qualquer um
- A parte privada você deve guardar a sete chaves
- A parte privada pode ter segurança adicional pelo uso de uma passprhase

1.0.3 Configurando suas chaves

```
marcos@groo:~/tmp$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/marcos/.ssh/id_rsa): teste
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in teste.
Your public key has been saved in teste.pub.
The key fingerprint is:
SHA256:0NSadaQFr0VskHUC5wEor8IaAeVzEWdZyi8YE+rvqXc marcos@groo
The key's randomart image is:
+---[RSA 2048]----+
| .. +.oooo=XO .  |
|.. . *o+. +B++   |
| .+ + +o.+.o+    |
| ..o + o+  o     |
|  .o. ..S .      |
|  ..o ..         |
|   o..           |
```

```
|  ....E          |
|  .oo.           |
+----[SHA256]-----+
```

1.0.4 Configurando chaves

- Foram gerados dois arquivos: `teste` (chave privada) e `teste.pub` (chave pública)
- Coloque `teste.pub` na máquina destino em `~/.ssh/authorized_keys`
- Agora você pode fazer `ssh user@host_destino`
- Bastando digitar sua passprhase.

1.0.5 Logando no DInf

- Existe uma máquina de nome ssh.c3sl.ufpr.br que permite login de casa
  - `ssh <seulogin>@ssh.c3sl.ufpr.br`
- Esta máquina dá acesso às outras máquinas do departamento, em particular a máquina `orval` (16 cores, 70Gb RAM, nobreak, gerador) ou aos terminais dos laboratórios

1.0.6 SSH – parte 2

1.0.7 Criando um par de chaves

```
Enter file in which to save the key (/home/meuusuario/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

- Uma *passphrase* é uma espécie de senha, que pode ser uma frase longa
- Exemplo: `We Will Rock You! by Queen, 1977 album News of the World.`
- É a criptografia da sua chave privada
- Se alguém roubar esta chave e não conhecer a *passphrase* você está mais seguro
- Ela pode ser vazia, mas isto não deve ser feito.
- Se quiser vazia, basta apertar `ENTER`
- O recomendado é escolher uma e digitá-la duas vezes

1.0.8 Criando um par de chaves

```
Your identification has been saved in /home/meuusuario/.ssh/id_rsa.
Your public key has been saved in /home/meuusuario/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:kNs5FPDUDjb9V2AarZH4Dt/iPktTaAWbuwdo6KQTL/A meuusuario@meucomputador
The key's randomart image is:
+---[RSA 2048]----+
|      ..oo .ooo. |
|       ++.+ oB. .|
|      o.o+ o+o.. |
|       = oo.++.  |
|    . o S o+=o.  |
|     o * o .++.  |
|      E o  .+..  |
|       o   .oo   |
|           .oo   |
+----[SHA256]-----+
```

- a *randomart* não serve para nada além de ser bonitinha
- `SHA256` é o tipo de criptografia utilizado

1.0.9 Configurando suas chaves

- Seu par de chaves está agora no seu diretório
- `/home/meuusuario/.ssh`
- Foram criados dois arquivos neste diretório:
  - `id_rsa`: sua chave privada
  - `id_rsa.pub`: sua chave pública
- Guarde muito bem sua chave privada!
- Por exemplo, em um *pendrive*, ou no seu computador mesmo, mas garanta segurança dele também!

1.0.10 Sua chave pública tem esta cara

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDKOxdpijGNfcI7hYXhn57hZnZSlHSVqQFTPMDzqJ/F
2S77rVVgwnxM0L/QJcWe1bTpd/9KllgLnId2Cpc/CWtisi4/U/pl11HKev8y/vAWz20DIgJOmh0c0HEs
OusELGmLaMRtK0AKuqUyLv9TmSowcq/4i4dzLPN7kqLSnA+6Ox0gvbHqnZFN9FqswTEKBblCvoBWNP47
FsaBwmCMbUZ1W01s0QaPtp4RBtVaxMLhexgeNv2xacxekkFIimZqIpK9A7xbBsvxrucucTOxv1W0AA3F
k6qUYxhSbAPdEtdNlmW/RePKuT1XSNmfh54glTtS+mBG4OSYNsTvGU3cDbpt meuusuario@meucomputador
```

- a parte `meuusuario@meucomputador` serve apenas para você saber onde esta chave foi gerada, na prática não serve para mais nada

1.0.11 Sua chave privada tem esta cara

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,E72291EFB35DCC40EFF09BFB16DBF8AD

rP1y0v/Kj9FSAvV9oCGxwCUD/xn0skyb8lzAtIThFVtAhG20YYNbzTA9YkREN/eo
iy6zXs8Vm4BPFDo1cPZ+LlfYiZMNecvIlghaE3kLJghxIAWgbKpn2W5Dm0g96E2Y
otlRd1ytjfcKAvT1TXjBacxEKy+KdmSwrcmnCqOczeLFoZ5PKFXamG/GgDPIfGyB
cp42K7svWgQmjrGb5Iw2STZ6brLmybFt7bWIBhQK2NOOsDFYNXEoPS3sf7Yvg2Na
s2y5VMd/c6/Fdrh3kF8dSMXbMT1HM2/Uqz9VWUEiy5AYbSUhpMetJhUCt5s53Sz0
crvsYwFlAHN2m34NUVpWvqAFrpQMn90ib3AaHcDPTOCXgvG6ZJ9VM29D973KMtSS
Al6SUdbKQ8v42opXzrB4x7hywTrCyylLCGjI/RGaCE/ovbaig6+QzpjePvLv2eYD
xNE2e5wGo+Q92ywM7eTQRhJ9YHmIFXtGQSgEui9DV8TuCJYhN6/47F0vsIuvJJV+
027KqmIfP97isX6PxjKCFBZVFMWhTah13BvckfvutGlT0aZIaN5dXkg57uk4cLsI
vAqBZO/MwSGjZXKZVCn1zMwrK5qXPfOfPiJNf7iNzOPBTtSFvCn9XBdX6iILrbDz
```

```
fzd0+55gI7xJl/euS2wBGN8/xy0YNV3EXXQhXih6dC57Qh+5eoN376KNxxIo86ga
6GJdd6ctIxHUAGNElX3L7nESIFTA6u0LNsboiazz0p5efOozX84KmyyoQbJj8ica
UwHTVbLDzFYhYbtWI7frsfcq9N/oeqxzTFPXfqDGZDE3srMckzGZ8E0hPf3Uut4E
AY14hlPZRhIOAEuJpygS0YwwMF6doPSiVse9V5rmfKUM0KJGBat9SJFsJnwde+oF
AjS8mz6JP09cW8W/B5d2Gr9ewZEH+o5F6KdS3gxbKzMuQuJKSOWlBd4UakebURsY
tB8ugfaatwF8iIcFQA73c7m8K6tWNprRX82VnAiVjAbxB7PgRL+uNSOmAM9fv5Wa
XypsjIn2BQZM9EdKvabd3yR3rQOZWwqm0BxmCEKOYB+w650QdWwP1K4rsVWTf6Wk
reiaLrakF9CXqvwCU/xXE6B7MWDPhFWcVRpK+XRqcIHI1lZtUX/0UORXC9fzWIg7
rJ+l3+cNKXW6o9Adwjcvcqbm9brjmLmMPx7RfqMgy0ZM1Sxxe3QoUegK+pMeymbv
61NGmV3vQY0E5pIpre8VOy1z202AyXlZzhX3itxKV6ilwxNOImZhj+tNK1iPIQXf
BfEJBRhIjYirRbbUWXCwf88ZbWVdVPvno9cwngBtkMDbV7CYpQaYrDqiAbjSr8WR
0MXHtCDtrfSykl5drA9T/DvN2nLTPjOyOPltSlvjt+QQN8V4FV1pNRnqtDAKqpMs
vuPel/F18yOwtcycajbrcYFUWgAFoLFCKbARZKxTTNfMDaGBZ53TlGhgCjokg96X
7rSPHHzNkeix0I2ohP0rHFL0pAlSPXyUJlKZRW/hsIwHqkPSAP2z932wrjAcgSvs
3vQC6DX3Vdyxumio/UHte9Ky7yLuYvpwRp4QcL9m7eK9ZxOoJJFFU5+YbAg1EW5B
-----END RSA PRIVATE KEY-----
```

1.0.12 Observações

- Observe a linha contendo `Proc-Type: 4,ENCRYPTED`
- Quer dizer que você criptografou com alguma *passphrase*
- Senão, sua segurança teria sido quebrada, a menos do fato de que:
- **A chave só foi mostrada porque é fictícia!**

1.0.13 Colocando sua chave em outro computador

- Copie sua chave na máquina de seu desejo, por exemplo, na máquina de nome `ssh` do dinf:
- `scp ~/.ssh/id_rsa.pub meuusuarionodinf@ssh.c3sl.ufpr.br:`
- Para isto será necessário digitar sua senha de forma aberta
- Ou então, copie a chave pública em um *pendrive* e traga fisicamente para o dinf se não quiser digitar sua senha aberta na Internet. . .

1.0.14 Colocando sua chave em outro computador

- Logue-se no dinf, remota ou presencialmente, digitando sua senha
- Execute este comando: `cat id_rsa.pub >> ~/.ssh/authorized_keys`
- Se você não tiver um diretório `.ssh` então crie um com a permissão correta (*macalan* é o nome verdadeiro da máquina cujo apelido é *ssh*):

  ```
  seuusuarionodinf@macalan:~$ mkdir ~/.ssh
  seuusuarionodinf@macalan:~$ chmod og-w ~/.ssh
  ```

- Isto é, somente você pode escrever neste diretório

1.0.15 Pronto!

- O arquivo `authorized_keys` pode ter várias chaves, por exemplo
- Uma do seu laptop
- Outra do seu computador desktop
- Uma outra do computador do seu trabalho
- O ideal é que você tenha então três chaves privadas com três *passphrases* diferentes...

1.0.16 Testando

- Agora volte para sua casa e execute
- `ssh meuusuarionodinf@ssh.c3sl.ufpr.br`
- Será pedida sua *passphase*, digite-a e você estará logado na *macalan*, vulgo *ssh* (veja no próximo slide):

1.0.17 Logado remotamente na *macalan*!

```
Welcome to Linux Mint 18.3 Sylvia (GNU/Linux 4.19.16+ x86_64)

Welcome to Linux Mint
 * Documentation:  http://www.linuxmint.com
======================================================================

Macalan (alias ssh) tem poucos recursos de memoria e processadores,
com limites rigidos de processos, memoria e arquivos abertos.

          >>> NAO DEVE SER USADA PARA PROCESSAMENTO <<<

Esta maquina deve ser usada apenas como acesso a outras servidoras.

Use uma das maquinas abaixo para jobs:

Servidoras de uso geral, para qualquer usuario:
- orval

Servidoras exclusivas para grupos:
- fradim: exclusiva para professores
- mumm: exclusiva para C3SL
Last login: Thu Aug 15 11:34:30 2019 from 10.254.229.23
seuusuarionodinf@macalan:~$
```

1.0.18 Observações finais

- Agora você pode fazer *ssh* para qualquer máquina interna do DInf!
- Por exemplo, a orval tem grande capacidade de CPU e RAM, use-a bem!
- Outro exemplo: `ssh h17`: um terminal de um dos laboratórios do DInf.
- Quando quiser (e puder) pode se logar no cluster HPC (High Performance Computer), uma espécie de supercomputador do C3SL.

1.0.19 Exercícios

- Se você não tem ssh instalado em seu computador, instale imediatamente!
  - Nas distros variantes de Debian, `apt install openssh-server openssh-client sshfs`
- Crie pelo menos duas chaves, cada uma para ser usada em diferentes situações (uso normal, uso de superusuário, etc).
- Aprenda a configurar o arquivo `~/.ssh/config`
- No seu computador, verifique os arquivos `/etc/ssh/sshd_config` e `/etc/ssh/ssh_config`. Veja se sua máquina está segura contra invasores. Procure na Internet dicas de como configurar corretamente estes arquivos.
- Copie algum arquivo do DInf para sua casa usando *scp*.

1.0.20 Criando um par de chaves

- No computador da sua casa, digite:
- `ssh-keygen -t rsa`

```
Enter file in which to save the key (/home/meuusuario/.ssh/id_rsa):
```

- Qual local você quer guardar sua chave privada?
- Se teclar `ENTER` ela vai ficar no lugar padrão indicado na mensagem
- Se não quiser, talvez porque você já tenha uma chave lá e quer criar uma segunda, basta digitar um nome qualquer de arquivo, com caminho completo ou não
- Se não usar caminho completo vai criar no diretório corrente

Apêndice B

RSYNC

- É uma ferramenta rápida e versátil para cópia de arquivos locais ou remotos
- É um dos programas usados como exemplo de qualidade e eficiência
- O rsync permite manter sincronizados arquivos em diferentes máquinas ou diretórios
- Ele copia apenas o que é diferente de um local para outro
- Pode ser usado com compressão e em conjunto com o *ssh*

2.0.1 Sincronizando sua conta do DInf no computador da sua casa

- Para sincronizar seu HOME no DInf no seu computador, basta executar no seu computador:
  - `rsync -av <seulogin>@ssh.c3sl.ufpr.br: destino`
- Da primeira vez ele vai copiar tudo
- Nas próximas vezes, somente o que tiver sido modificado deste a última vez

2.0.2 Exercícios

- Se você não tem instalado em seu computador, instale imediatamente!
  - Nas distros variantes de Debian, `apt install rsync`
- Faça uma cópia dos seus arquivos do DInf na sua casa, a título de backup
- Entenda a diferença entre estas maneiras de chamar o *rsync*:
  - `rsync -av <seulogin>@ssh.c3sl.ufpr.br: destino`
  - `rsync -av <seulogin>@ssh.c3sl.ufpr.br/: destino`
  - `rsync -av <seulogin>@ssh.c3sl.ufpr.br: destino/`
  - `rsync -av <seulogin>@ssh.c3sl.ufpr.br/: destino/`

Apêndice C

SSHFS

- Montagem remota e segura (criptografada) de um sistema de arquivos
- Permite montar um diretório remoto de maneira transparente para o usuário
- Facilita muito trabalhar em casa, por exemplo, com seus arquivos estando no DInf, por exemplo

3.0.1 Montando seu HOME no DInf na sua casa

- Para montar seu HOME do DInf na sua casa:
  - `mkdir dinf`, se o diretório não existir, crie um
  - garanta que ele esteja vazio
  - `sshfs <seulogin>@ssh.c3sl.ufpr.br: dinf`
- Para montar um diretório específico do seu HOME do DInf na sua casa:
  - `mkdir dinf_prog1`, se o diretório não existir, crie um
  - garanta que ele esteja vazio
  - `sshfs <seulogin>@ssh.c3sl.ufpr.br:ProgramacaoI dinf`
- Para desmontar: `fusermount -u dinf`

3.0.2 Exercícios

- Se você não tem instalado em seu computador, instale imediatamente!
  - Nas distros variantes de Debian, `apt install sshfs`
- Monte algum diretório de seu interesse do seu HOME no DInf no computador da sua casa.

Referências

[Banahan u. a. 1991] Banahan, M. ; Brady, D. ; Doran, M.: *The C Book, Featuring the ANSI C Standard*. Addison-Wesley Publishing Company, 1991 (Instruction set)

[Kernighan 1990] Kernighan, Brian W. ; Ritchie, Dennis M. (Hrsg.): *A linguagem de programação C – padrão ANSI*. Editora Campus, 1990

[Leiserson u. a. 2002] Leiserson, C.E. ; Cormen, T.H. ; Rivest, R.L. ; Stein, C.: *Algoritmos: teoria e prática*. ELSEVIER, 2002

[Newham und Rosenblatt 2005] Newham, C. ; Rosenblatt, B.: *Learning the Bash Shell: Unix Shell Programming*. O'Reilly Media, 2005 (In a Nutshell (o'Reilly) Series)

[Sedgewick 1983] Sedgewick, R.: *Algorithms*. Addison Wesley, 1983

[Tenenbaum u. a. 1995] Tenenbaum, A.M. ; Langsam, Y. ; Augenstein, M.J.: *Estruturas de dados usando C*. Pearson Makron Books, 1995

[Wirth 1976] Wirth, Niklaus: *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976

[Ziviani 2007] Ziviani, N.: *Projeto de algoritmos: com implementações em Java e C++*. THOMSON PIONEIRA, 2007