ISEP – LEEC - Sistemas Computacionais 2019/2020

Ficha 1 – Revisões da linguagem C e utilização de sistemas Linux

Objetivos

- Contactar com a distribuição de Linux que irá ser utilizada nas aulas PL e utilizar as ferramentas disponibilizadas para desenvolver um programa introdutório em C.
- Conhecer alguns dos procedimentos para gestão de processos através *shell*: `ctrl+c, &, ps, kill`.
- Conhecer alguns dos comandos para utilização do sietma de ficheiros: `cd, pwd, ls, mv, mkdir, rm`.
- Revisões da linguagem C.

Introdução

Linguagem de programação C

Nesta unidade curricular é usada a linguagem de programação C. Como apoio para o estudo da linguagem C, poderá utilizar, por exemplo, um dos seguintes livros:

"The C Programming Language", Brian W. Kernighan, Dennis M. Ritchie. Prentice Hall, 1988
"Linguagem C", Luís Damas. FCA - Editora de Informática, 1999.

Sistemas GNU/Linux

Esta unidade curricular usa o sistema operativo (SO) GNU/Linux como base para a aplicação dos tópicos lecionados.
Em 1983, Richard Stallman fundou o projeto GNU com o objetivo de desenvolver todas as peças de um sistema operativo semelhante ao UNIX, mas completamente gratuito. No início da década de 1990, Linus Torvald contribuiu com a peça central do sistema operativo: o núcleo (*kernel*). O núcleo de um SO oferece as primitivas básicas usadas pelas bibliotecas e aplicações do sistema: gestão de processos, gestão de memória e gestão de entrada/saída de dados e da interface HW/SW dos vários dispositivos que constituem o sistema computacional. A primeira versão do Linux foi lançada em 1992.
Sendo um *software open source*, todo o código usado na programação do Linux está disponível gratuitamente. A maior parte do código do Linux é escrito na linguagem C. Apenas uma pequena parte, por questões de eficiência, é escrito em *assembly*.

Os sistemas operativos GNU/Linux são disponibilizados na forma de distribuições. Uma distribuição consiste numa combinação de componentes de *software*, selecionados de forma a servir da melhor forma um público alvo. Existem distribuições generalistas, que procuram disponibilizar as versões mais recentes das aplicações mais populares; distribuições dedicadas apenas a áreas mais específicas (multimédia, robótica, matemática); e, como último exemplo, outras que apostam na "leveza", de forma a poderem a ser utilizadas em computadores mais antigos, podendo mesmo recorrer a

versões mais antigas de determinados componentes. Cada distribuição tem a sua própria nomenclatura no que diz respeito a numerações de versões. É possível inspecionar individualmente a versão de cada componente de software da distribuição, inclusive do próprio *kernel* (o Linux propriamente dito). A versão do Linux da distribuição usada num dado sistema pode ser obtida executando o comando "`uname -a`" numa *shell* desse sistema. Pode verificar qual a versão mais recete do Linux em `http://www.kernel.org/.`

Do ponto de vista do utilizador, as distribuições generalistas GNU/Linux oferecem funcionalidades semelhantes às encontradas em sistemas operativos comercias para computadores pessoais, tais como o Microsoft Windows e o Mac OS X. Entre as distribuições generalistas mais populares encontram-se o Mint, Ubuntu, Fedora e openSuse (ver `http://distrowatch.com/`). Devido a esta grande variedade de distribuições torna-se mais difícil encontrar um guia único que consiga cobrir todas as especificidades de cada distribuição Linux, existindo diversas publicações, umas mais genéricas, outras mais orientadas a distribuições específicas.

Os trabalhos das aulas PL serão elaborados com base na distribuição Fedora (*download* gratuito em `http://fedoraproject.org/get-fedora.html`). O Fedora oferece vários ambientes de trabalho, dos quais se destacam os seguintes:

- Xfce – um ambiente tradicional, com menu de inicio de aplicações e barra de tarefas. Pouco exigentes em termos de requisitos computacionais, o que pode ser importante caso o Linux seja instalado numa máquina virtual tipo VirtualBox ou Vmware.
- Lxde – ainda mais simplista e leve do que o Xfce.
- Gnome – é o ambiente *default* para várias distribuições e, a par com o KDE, um dos que tem maior suporte em termos de desenvolvimento. Tem sido criticado por eliminar algumas funcionalidades tradicionais e por uma filosofia que parece mais adequada para dispositivos com ecrã táctil. Poderá ser um pouco pesado para algumas máquinas, principalmente em ambientes de virtualização.
- KDE – ambiente bastante completo com muitas opções de configuração. Pode ser considerado demasiado complexo para alguns utilizadores. Tal como o Gnome, poderá ser um pouco pesado para algumas máquinas.

Os sistemas operativos GNU/Linux são sistemas operativos multi-utilizador. Cada utilizador pode ter a sua conta, com o respetivo nome de utilizador (*username*) e palavra chave (*password*). Num sistema operativo tipo UNIX, o utilizador com permissões totais sobre o sistema é denominado por *root*. Este utilizador existe em qualquer distribuição. No entanto, não é boa prática trabalhar sob a forma deste utilizador (no caso do Ubuntu, é prática comum usar o comando *sudo* para se obter os privilégios de *root).* Uma boa política de utilização da máquina dita que este utilizador só seja usado em tarefas de administração (por exemplo, instalação/atualização de aplicações e alterações da configuração do sistema). As distribuições mais comuns obrigam à criação de pelo menos uma conta de utilizador durante a fase de instalação, solicitando o respetivo nome do utilizador e palavra chave.

Interface gráfica vs "shell"

Existem sistemas de desenvolvimento gratuitos para Linux com um grau de sofisticação bastante elevado. No entanto, para tornar mais claro todo o processo de desenvolvimento

da aplicação, nestas aulas recorreremos a ferramentas mais simples e à "linha de comando" (a *shell*).

A generalidade dos sistemas operativos modernos (MacOS, Linux, Microsoft Windows, etc.) dispõe de interfaces gráficas com o utilizador, baseadas no paradigma WIMP (*Windows, Icons, Menus and Pointing device*). Tipicamente podemos iniciar qualquer aplicação selecionando o seu nome num menu ou selecionando o ícone de um ficheiro de dados numa determinada listagem.

No entanto, também é possível lançar qualquer aplicação e executar comandos de sistema através da *shell*. Uma das funções principais da *shell* é permitir o lançamento de programas.

É de notar que a interface gráfica não é um componente obrigatório nas distribuições GNU/Linux. É possível instalar uma distribuição do sistema operativo sem qualquer tipo de interface gráfica, tendo como interface com o utilizador apenas uma *shell.*

Exercícios

Exercício 1

Pretende-se escrever e executar o seguinte programa:

```
/* Hello World program */

#include<stdio.h>

int main() {
  printf("Hello World\n");
  return(0);
}
```

Para tal deverá introduzir a listagem no computador e gravá-la como um ficheiro de texto com extensão **"`.c`"** (por exemplo, **"`hello.c`"**). De seguida, são apresentados os passos que deverá seguir para levar a cabo essa tarefa.

Em geral para abrir uma *shell* deverá procurar e executar a aplicação *terminal* ou *console*. É então criada uma nova janela, onde aparece a linha de comando da *shell*.

Para introduzir o código do programa poderá utilizar qualquer editor de texto à sua escolha. Estão disponíveis editores de texto que correm diretamente na consola da *shell*[1] (`nano, vi, emacs`) e editores de texto em modo gráfico (`gedit, kwrite, kate, geany`). Tanto os editores gráficos como alguns dos editores em "modo de texto" oferecem alguma forma de colorir a sintaxe e indentação automática.

Para lançar o editor de texto (neste exemplo, será usado o `geany`) digite o seguinte comando na *shell*, seguido da tecla `Enter`:

```
geany &
```

O comando acima inicia o editor de texto numa nova janela. Caso não fosse digitado o carácter &, a linha de comando ficaria "ocupada" até que a aplicação do editor de texto fosse terminada. Mais à frente, poderá executar o compilador na *shell* sem ter que terminar o processador de texto.

Copie para o editor de texto o programa acima e no final grave-o (*File/Save*) com o nome `hello.c`. Os ficheiros da área do utilizador dos computadores do laboratório são apagados em cada arranque pelo que se recomenda que, no final da aula, armazene os seus ficheiros num dispositivo externo (USB *drive*) ou que copie os ficheiros para alguma área de armazenamento remoto no final da aula (pode inclusive usar a sua conta de aluno na máquina `ave.dee.isep.ipp.pt`).

[1] Para profissionais ativamente envolvidos na administração de máquinas tipo UNIX é essencial o domínio de um editor em modo de texto, pelo simples motivo de que por vezes não é possível aceder à interface gráfica ou esta pode simplesmente não existir.

Nota sobre o sistema de ficheiros do Linux

O Linux usa um sistema de ficheiros hierárquico, cuja raiz é identificada por "/". O diretório de trabalho base de um utilizador "sistc" será "/home/sistc/".

Na *shell*, pode verificar em que diretório se encontra através do comando `pwd` (*print working directory*). Poderá criar novos diretórios usando o comando `mkdir` (por exemplo, "`mkdir mydir`"). Poderá mudar de diretório de trabalho através do comando `cd`. Exemplos:

```
[/home/sistc]$ pwd
/home/sistc
[/home/sistc]$ cd dir_existente
[/home/sistc/dir_existente]$ cd ..
[/home/sistc]$ cd /home/sistc/dir_existente
[/home/sistc/dir_existente]$ cd ~sistc
[/home/sistc]$ cd ~sistc/dir_existente
[/home/sistc/dir_existente]$
```

Poderá listar o conteúdo de um diretório através do comando `ls` (ou "`ls -l`", para obter informação mais detalhada – para outras opções faça "`man ls`"). Outros comandos úteis são o `mv` (mover/renomear), `rm` (remover) e `cp` (copiar).

Ao contrário do paradigma do Microsoft Windows, não são utilizadas letras para indicar as unidades de armazenamento (para analisar a forma como diferentes dispositivos são mapeados no sistema de ficheiros pode executar os comandos `lsblk` ou `mount`). Outros comandos úteis:
- `df`, indica o espaço disponível nas unidades de armazenamento do sistema.
- `du`, pode ser usado para verificar o espaço ocupado por um diretório específico (e suas ramificações).

Selecione novamente a janela contendo a *shell*, mude (`cd`) para o diretório onde gravou o ficheiro `hello.c` e introduza o seguinte comando:

```
gcc hello.c –o hello -Wall
```

Este comando compila o ficheiro `hello.c` e cria o executável `hello` (o nome do executável é indicado pela opção `-o`). A opção `-Wall` faz com que sejam gerados avisos (*warnings*) sempre que forem detetadas construções consideradas duvidosas ou que potencialmente possam levar a erros de execução. Embora os *warnings* não impeçam a geração do executável, em geral deve corrigir o código de forma a evitar qualquer *warning.*

De seguida execute o programa acabado de criar, introduzindo o seguinte comando na *shell*:

```
./hello
```

Exercício 2
(listagem e terminação de processos)

Crie o ficheiro `ex2.c` com o seguinte código:

```
//ex2.c:
#include <stdio.h>
#include <unistd.h>

int main() {
  while(1) {
    printf("Em funcionamento!\n");
    sleep(15);
  }
  return(0);
}
```

a) Execute os seguintes comandos:

```
make ex2
./ex2
```

O programa vai continuar a imprimir mensagens no ecrã de 15 em 15 segundos. Como procederia para terminar este programa sem fechar a janela de consola?
Use a combinação de teclas **CTRL+C** (carregar na tecla CTRL e, de seguida, sem libertar a tecla CTRL, carregar na tecla C).

b) Inicie novamente o programa, mas desta vez com a opção de linha de comando &:

`./ex2 &`

Se esperar algum tempo (quanto?) vai observar que o programa continua a imprimir no ecrã. Neste caso, como poderia proceder para terminar o programa, sem terminar a *shell*? Siga os seguintes passos:

- Execute o comando

  `ps -u` *`username`*

  onde *username* é o nome do utilizador que usou para fazer o *login*. Este comando apresenta-lhe a lista dos seus processos.
- A partir da listagem do comando `ps`, obtenha o código numérico de identificação do processo (PID) referente à execução do programa `ex2`.
- Execute o comando

  `kill` *`PID`*

  onde PID é o identificador obtido no ponto anterior. Este comando envia um sinal de terminação ao processo indicado, que, se não tiver tomado nenhuma disposição especial (a ver futuramente na matéria de sinais), deverá terminar.

Execute novamente o comando **`ps -u` *`username`*** para verificar se o processo foi realmente terminado.

c) Inicie novamente o programa e use a combinação CTRL+Z. A execução do programa é parada, mas este continua em memória, como pode verificar com o comando

`ps -u username`

Introduza o comando **`fg`** na *shell* para retomar a execução do programa.

O processo também pode ser retirado do estado "Parado" através do comando

`kill -CONT PID`

onde PID é o identificador do processo. Use novamente a combinação CTRL+Z e teste o comando.

A recordar:
- Para **terminar um processo** em primeiro plano (*foreground*), use a combinação **CTRL+C** (e não CTRL+Z).

Exercício 3

Que valores são impressos pelo programa abaixo? Justifique.

```
char a = 46;
char *ptr;
ptr = &a;
*ptr = 47;
a = 48;
printf("%d, %d\n", *ptr, a);
a = '0';
printf("%d, %c, %x\n", a, a, a);
```

Exercício 4

(passagem por valor vs passagem por referência)

Que valores são impressos pelo programa abaixo? Justifique.

```
void teste(char p1, char p2, char *p3) {
	p2 = p2 + p1;
	*p3 = 3*p1;
}

int main() {
	char a = 10;
	char b = 20;

	teste(10, a, &b);
	printf("%d, %d\n", a, b);
}
```

Exercício 5
(leitura de valores, ciclos, condições, funções, passagem por valor vs passagem por referência)

a) Escreva um programa que leia dois valores numéricos (inteiros) a partir do teclado e que imprima o resultado da divisão inteira do primeiro valor (dividendo) pelo segundo (divisor) assim como o respetivo resto (use o operador `%`). O programa deverá repetir o procedimento (ler valores e executar divisão) até que o valor introduzido para o dividendo seja 0.

Para ler dados introduzidos pelo utilizador pode usar a já bem conhecida função `scanf`[2]:

```
int scanf (const char *template, ...)
```

Exemplo:

```
int a;
scanf("%d", &a);
```

Crie o respetivo executável e teste o seu programa.

b) Altere o seu programa de forma a que os valores do quociente e do resto sejam calculados numa nova função (**`minha_divisao`**, descrita abaixo) e guardados em variáveis declaradas na função *main*. Para tal, pretende-se que use a passagem de parâmetros por referência. A função deverá obedecer ao seguinte protótipo:

```
void minha_divisao(int dividendo, int divisor, int *ptr_quociente, int *ptr_resto);
```

A impressão do quociente e do resto deverá continuar a ser feita na função *main*.

[2] Para mais detalhes execute na *shell* o comando `man scanf`. Pode também consultar o manual da biblioteca C por exemplo em:
`http://www.gnu.org/software/libc/manual/`
`http://www.die.net/` (páginas *man online* do Linux)
`http://www.cppreference.com/`

Em alguns casos as páginas *man* poderão não estar instaladas por *default.* No caso do Fedora o pacote a instalar é o `man-pages`. A sua instalação pode ser feita através da linha de comando, usando o comando `dnf` com permissões de administrador *(superuser):*

```
su      # para obter as permissões de root. Ser-lhe-á pedida a password de root.
dnf install man-pages
```

Para a distribuição Ubuntu, o comando é:

```
sudo apt-get install manpages manpages-dev manpages-posix manpages-posix-dev
```

Nota sobre páginas *man*

1) Entre outras coisas, as páginas *man* indicam a lista de ficheiros de cabeçalho (.h) que devem ser incluídos para cada função. Se usar uma função sem a respetiva declaração, irá receber a seguinte mensagem de aviso do compilador:

```
warning: implicit declaration of function ‘nome_função’
```

A declaração permite ao compilador verificar se o utilizador está a passar os argumentos corretos a cada função e se os valores retornados correspondem àqueles que o programa espera receber.

2) Tipicamente, um sistema Linux tem páginas *man* para diversas componentes do sistema operativo (*shell*, bibliotecas, chamadas ao sistema, etc.). Por esse motivo poderão existir palavras-chave que são comuns a diferentes componentes do sistema. De forma a facilitar a organização e pesquisa, as páginas *man* estão organizadas em capítulos (poderá ver a organização típica das páginas *man* por exemplo em `http://linux.die.net/man/`). Em alguns casos torna-se necessário indicar a secção onde se encontra o comando ou função que se pretende consultar. A título de exemplo, execute as consultas abaixo numa consola Linux:

```
man printf
man 3 printf
man read
man 2 read
```

Exercício 6
(cadeias de caracteres, parâmetros da função `main`)

Modifique o programa do exercício 5b) de forma a que quando forem indicados dois valores na linha de comando (por exemplo, “`./divisao 10 2`”) o programa execute a função `minha_divisao` (exercício 5b), usando os valores indicados na linha de comando, e termine a sua execução. Nos restantes casos, o programa deverá comportar-se como descrito no exercício 5b).

Para tal, a função `main` deverá ter os argumentos definidos tal como indicado abaixo:

```
int main(int argc, char* argv[])
```

Para converter uma *string* para um inteiro pode usar a função `atoi`.

Crie o respetivo executável e teste o seu programa.

Exercício 7
(funções, apontadores)

a) Analise as seguintes **implementações**[3] das funções `strcpy` e `memcpy` da biblioteca standard da linguagem C.

```
char *strcpy(char *dst, const char *src) {
  char *cp = dst;
  while (*cp++ = *src++);
  return dst;
}

//equivalente à implementação acima
char *strcpy(char *dst, const char *src) {
  char *cp = dst;
  while(1) {
    *cp = *src;
    if(*cp == 0)
      break;
    cp++; src++;
  }
  return dst;
}
```

```
/* versão simplificada - normalmente copia
sizeof(int) bytes em cada iteração */

void *memcpy(void *dst, const void *src, size_t
n) {
  char *s = (char *) src;
  char *end = s + n;
  char *d = (char *) dst;

  while(s != end)
    *d++ = *s++;

  return dst;
}
```

Qual a principal diferença entre ambos?

b) Implemente um programa que copie para os vetores `str2` e `data2` o conteúdo dos vetores `str1` e `data1`, respetivamente, usando as funções mais adequadas da biblioteca standard da linguagem C. O programa deverá também imprimir no ecrã a quantidade de memória ocupada por cada variável (e.g., `printf("%ld\n", sizeof(str1))`). Justifique os resultados.

```
char str1[] = "Teste!";
int data1[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
char str2[1000];
int data2[1000];
```

c) Apresente uma possível implementação da função `strcmp` que retorne 0 em caso de igualdade e -1 em todos os outros casos.

d) Apresente uma possível implementação da função `memcmp` que retorne 0 em caso de igualdade e -1 em todos os outros casos.

[3] "Implementação de uma função" refere-se à ***definição*** da mesma, i.e., ao *cabeçalho* da função (tipo do valor de retorno, nome da função e lista de parâmetros) juntamente com o seu *corpo* (definido entre as chavetas).

O código fonte do corpo destas funções não está habitualmente disponível nos sistemas operativos, uma vez que este não é necessário nem para a criação nem para a execução dos programas. Estas funções estão presentes já na sua versão compilada (código objeto) nas bibliotecas do sistema (ficheiros .so em Linux, ficheiros .dll em Windows). Estas bibliotecas serão então usadas na altura da execução dos programas que façam uso das respetivas funções.

Por sua vez, os ficheiros .h possuem apenas a ***declaração*** (*protótipo*) das funções, eventualmente com a declaração de tipos de dados e constantes a utilizar com essas funções, e são usados apenas na fase de compilação do código C. Por exemplo, a declaração da função `strcpy` é simplesmente:
`char *strcpy(char *dst, const char *src)`

Exercício 8
(*struct, typedef*)

Pretende-se escrever um programa que permita fazer a leitura de campos relativos ao registo de automóveis e armazená-los numa variável. Os dados a guardar para cada automóvel são a matrícula (vetor de 6 caracteres, **não** terminado com `'\0'`), o nome do proprietário, até um máximo de 80 caracteres (*string,* cadeia de caracteres terminada com `'\0'`) e o valor comercial do automóvel.

Cada registo deverá obedecer à seguinte estrutura:

```
#define VEHIC_MAXPLEN 81
typedef struct {
  char plate[6];
  char owner[VEHIC_MAXPLEN];
  double value;
} vehic_t;
```

8.1) Implemente uma função `void vehic_print(vehic_t *v)` que imprima no ecrã os campos da estrutura apontada por `v`. O campo `value` deverá ser impresso apenas com duas casas decimais (`"%.2lf"` na formatação do `printf`). A função deverá ser testada com o seguinte programa:

```
int main(){
  vehic_t v1 = {{'1', '4', '7', '9', 'H', 'P'},
                  "Carlos Reis Salvador Almada",
                  2999999.99};
  vehic_print(&v1);
  return 0;
}
```

Exemplo de manipulação de variáveis estruturadas

Considere a seguinte declaração do tipo de dados `mystruct_t` e das variáveis `s1` e `ptr`:

```
typedef struct {
      int i;
      char c;
      float f;
} mystruct_t;

mystruct_t s1;
mystruct_t *ptr = &s1;
```

Todas as instruções abaixo atribuem o valor 1 ao campo `i` da variável `s1`:

```
s1.i = 1;
(*ptr).i = 1;
ptr->i = 1;
ptr[0].i = 1;
```

Observações: o campo da matrícula não contém uma *string* válida, uma vez que não é terminada com `'\0'`. Por esse motivo não pode simplesmente fazer o `printf` com a opção `"%s"`. Uma possível abordagem é utilizar o `printf` com a *string* de formato `"%.6s"`*;* outra possibilidade é usar a função `fwrite`*,* usando o `stdout` como *stream* de destino.

8.2 – Implemente uma função `void vehic_read(vehic_t *v)` que solicite os dados do veículo ao utilizador (nome do proprietário, matrícula e valor comercial) e preencha a estrutura apontada por `v` com os dados obtidos do teclado. A função deverá ser testada com o seguinte programa:

```
int main() {
      vehic_t v1;
      vehic_read(&v1);
      vehic_print(&v1);
      return 0;
}
```

Observações:
a) O campo `owner` (proprietário) deverá ser lido com a função `fgets` e não com o `scanf`*,* uma vez que o nome poderá conter espaços. Elimine o carácter de fim de linha (`'\n'`) escrevendo o carácter indicador de fim de *string* (`'\0'`) na posição do (`'\n'`).

b) No caso do campo `plate` (matrícula), copie apenas os 6 caracteres iniciais do texto introduzido pelo utilizador para o respetivo campo da estrutura `vehic_t`; use a função `fgets` para fazer a leitura da matrícula para um vetor auxiliar e de seguida utilize a função `memcpy` para copiar apenas os primeiros 6 caracteres para o respetivo campo da estrutura.

c) Para o campo `value`, caso faça a leitura do valor do tipo `double` usando a função `scanf`, deverá ter o cuidado de consumir o fim de linha (p.e., com uma chamada à função `getchar`) após a chamada ao `scanf`. Alternativamente, poderá utilize a função `fgets` seguida da função `atof`. Assuma que o valor lido tem no máximo 15 dígitos e eventualmente o separador decimal.

Resumo

Processo – designação dada à execução de um programa. Cada processo é identificado por um código numérico, normalmente abreviado por PID (de *process identifier*).

Processo em *foreground* – processo que ocupa a linha de comando e recebe diretamente os inputs do utilizador (é o modo *default* quando se executa um programa).

Processo em *background* – processo em execução sem ocupar a linha de comando e sem receber diretamente os inputs do utilizador. Um processo pode ser lançado em background utilizando a opção & (exemplo: `gedit &`).

Comando `ps` – permite obter a listagem de processos em execução, incluindo nomeadamente o seu PID. O comando `ps -e` apresenta a lista de todos os processos do sistema e o `ps -u` *`user`* mostra apenas os processos do utilizador especificado.

Comando `kill` – pode ser usado para terminar um processo. Depois de conhecido o PID, basta executar `kill` *`PID`* para enviar um sinal com pedido de terminação do programa.

Combinação de teclas CTRL + C – envia um pedido de interrupção ao processo em *foreground*. Por *default*, esse pedido é atendido e o processo é terminado.

Navegação nos diretórios do sistema de ficheiros:

- `cd` – permite alterar o diretório atual.
- `pwd` – imprime o nome do diretório atual
- `ls` – listagem de ficheiros (incluindo diretórios).
- Dispositivos de armazenamento externos (e.g. USB *drives*) são adicionados ao sistema como diretórios normais. Uma forma de descobrir o diretório correspondente a um dispositivo de armazenamento externo a partir da linha de comando é executar o comando `lsblk`. O dispositivo será normalmente associado ao ficheiro `/dev/sdb` ou, dependendo do número de dispositivos de armazenamento no sistema, `/dev/sdc`, `/dev/sdd`, etc.

Exemplos de utilização do `gcc`:

- Nota: na geração de executáveis, a opção `-o` serve para especificar o nome do executável. Se esta opção não for usada, o executável gerado será nomeado `a.out`.
- `gcc teste.c -o teste` – compila o ficheiro `teste.c` e gera, com o código objeto resultante, o executável teste.
- `gcc teste.c` – compila o ficheiro `teste.c` e gera, com o código objeto resultante, o executável `a.out`.
- `gcc teste.c -c` – apenas compila o ficheiro `teste.c`, sem gerar qualquer executável.

Variáveis na linguagem C

- Em geral, as variáveis são mecanismos que permitem armazenar dados na memória principal (informalmente designada por "RAM"). Cada tipo de dados (`char`, `int`, `long`, `float`, `double`) ocupa diferentes quantidades de

memória (usar *macro* `sizeof()`, `e.g., sizeof(long)` retorna o número de *bytes* ocupados por uma variável do tipo `long`)

- As variáveis do tipo apontador (declaradas com o *, e.g., `char *ptr`) podem ser usadas para aceder, de forma indireta, a outras variáveis. São particularmente úteis na passagem de argumentos para as funções e na alocação dinâmica de memória (com a função *malloc* e afins). É essencial perceber perfeitamente o seu funcionamento para uma boa compreensão da linguagem C.

Utilização de apontadores na linguagem C:

- A expressão `*myptr` é usada para aceder aos dados apontados por `myptr`. É equivalente a `myptr[0]`.
- A expressão `&myvar` é usada para obter um apontador para a variável `myvar`.

Histórico

- 2006-03 – Primeira versão, com exercícios 1 e 5 - Jorge Estrela da Silva (jes@isep.ipp.pt)
- 2009-03 – Adição dos exercícios 3, 4 e 6, edições diversas - Jorge Estrela da Silva (jes@isep.ipp.pt)
- 2014-03 – Adição do exercício 2, edições diversas - Jorge Estrela da Silva (jes@isep.ipp.pt)
- 2015-03 – Adição do exercício 7, edições diversas - Jorge Estrela da Silva (jes@isep.ipp.pt)
- 2018-03 – Adição do exercício 8, edições diversas - Jorge Estrela da Silva (jes@isep.ipp.pt)
- 2020-02-03 – Edições diversas - Jorge Estrela da Silva (jes@isep.ipp.pt)