POLITÉCNICO DE LEIRIA | ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Sistemas Operativos – Licenciatura em Eng.ª Informática

Ficha 3 – Introdução à Linguagem de Programação C em Linux

Versão 2.4

Duração: 1 aula

© **2024**: {miguel.negrao, miguel.frade, patricio.domingues, vitor.carreira, loureiro, nfonseca, rui, nuno.costa}@ipleiria.pt

Sumário

1 Introdução

O Linux adotou o compilador `gcc` ou *GNU Compiler Collection* de uso livre e de elevada reputação e a linguagem `C` por ser compacta, estruturada e eficiente. Nas aulas será empregue a norma `C11` da linguagem `C` suportada na íntegra pela versão do `gcc` presente na imagem da máquina virtual.

1.1 Edição

Durante a resolução das fichas, os estudantes podem utilizar os programas como editores gráficos de código-fonte:

- `gedit` (para ambientes GNOME),
- `kate` (para ambientes KDE),
- `featherpad` (para ambientes LXQT), ou
- `code` (Visual Studio Code, já instalado na VM fornecida)

Estes editores encontram-se disponíveis no ambiente de trabalho. Em alternativa poderão instalar o editor `Sublime Text 2/3` (não está incluído na VM).

Para a linha de comando, poderão utilizar editores de texto não dependentes de ambiente gráfico tais como `vim`, `jed`, ou o `pico`.

Para fazer uso de um qualquer destes editores, digite o nome do mesmo seguido do ficheiro a editar. Exemplo

```
1 user@linux:~$ gedit ficheiro.c &
```

Para além dos editores acima referidos, poderão utilizar IDEs (Integrated Development Environments) mais avançados tais como o Geany ou o Eclipse CDT. Para mais informações consulte a documentação de cada um. Caso tenha as ferramentas da máquina virtual instaladas, pode trocar ficheiros entre o Windows e a máquina virtual. No caso do VMware Workstation, pode aceder à janela de configuração da forma documentada nas imagens que se seguem. configurando as pastas partilhadas.

1.2 Partilha de ficheiros entre o sistema hóspede e o sistema hospedeiro

No VMWare, a partilha de ficheiros entre o Lubuntu (sistema hóspede) e o Windows (sistema hospedeiro) pode ser ativada através do menu "*Virtual Machine Settings*" (ver Figuras 1, 2), ativando-se a opção "*Shared Folders*", selecionando-se o diretório da máquina local a partilhar com a máquina virtual.

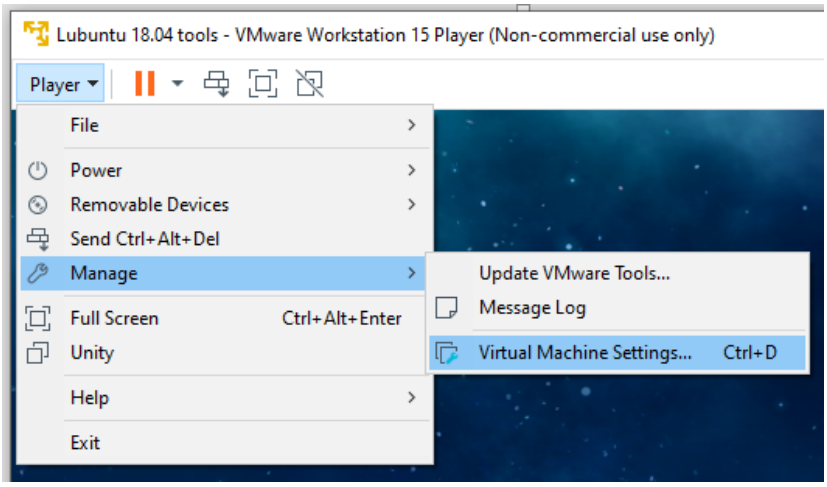

Figura 1: Menu VMware

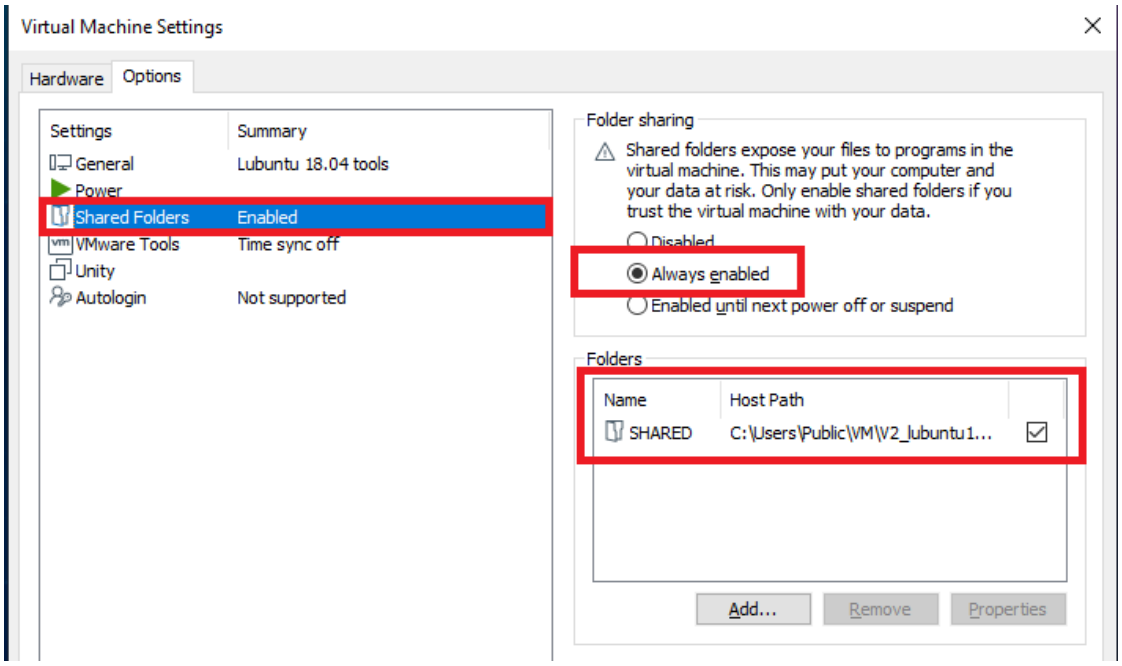

Figura 2: Partilha de pasta no VMware

Após ter sido partilhado, o diretório fica disponível no sistema de ficheiro da máquina virtual Linux, no diretório `/mnt/hgfs/NOME`, em que `NOME` corresponde ao nome do diretório partilhado. Acesso através da linha de comando:

```
1 user@linux:~$ cd /mnt/hgfs/
```

Pode ser necessário editar o ficheiro `/etc/fstab` adicionando a seguinte linha:

```
1 .host:/ /mnt/hgfs fuse.vmhgfs-fuse uid=1000,gid=1001,defaults,allow_other 0 0
```

O ficheiro `/etc/fstab` pode ser editado fazendo `sudo featherpad /etc/fstab`.

1.3 código-fonte

O código-fonte consiste num ficheiro de texto que contém um conjunto de instruções, escritas numa determinada linguagem, que descrevem um determinado programa. De seguida é apresentado o código-fonte, em linguagem `C`, de um programa que simplesmente envia `Olá mundo ...` para o ecrã.

Listagem 1: Ficheiro olamundo.c

```c
#include <stdio.h>

int main(void) {
    printf("Olá mundo...\n");
    return 0;
}
```

2 Compilação

O processo de compilação tem como objetivo converter o código-fonte em código máquina, seja ele final ou intermédio. Neste processo estão envolvidos quatro agentes: o pré-processador, o compilador, o *assemblador* e o *linker*. A Figura 3 mostra o fluxograma do processo de compilação:

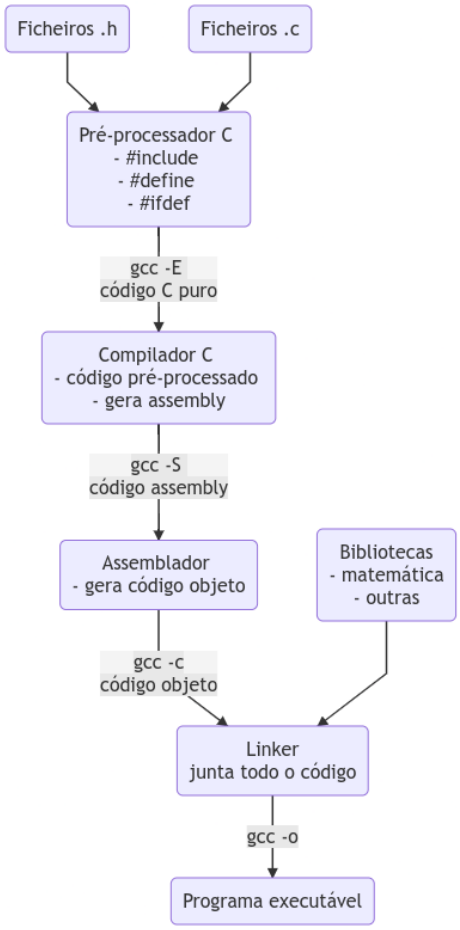

Figura 3: Processo de compilação

O pré-processador tem como finalidade substituir todas as diretivas de inclusão (isto é, os `#includes`) pelo conteúdo propriamente dito e a substituição de macros ao longo do código. O compilador serve para, a partir do resultado do pré-processamento, gerar código assembly pronto a ser *linkado*. O assemblador traduz o código assembly em código máquina não executável. O *linker* constitui a última fase do processo de compilação e tem como objetivo juntar todo o código máquina produzido até então num ficheiro pronto a executar.

O compilador integrado no GCC tem o nome `gcc` e pode ser utilizado de duas formas distintas, dependendo das circunstâncias: a compilação na linha de comandos e via `makefile`s. A compilação usando `makefile`s justifica-se apenas quando o programa é constituído por vários ficheiros ou quando se pretendem executar várias ações sobre o código-fonte. A utilização de `makefile`s será objeto de estudo na próxima ficha.

2.1 Como compilar um programa C

O compilador de linguagem `C` utilizado ao longo das aulas será o `gcc`. De seguida são ilustradas duas maneiras simples de chamar o `gcc`:

1. que compila e cria o ficheiro objeto `MostraArgv.o`:

```
user@linux:SO$ gcc -c MostraArgv.c
```

2. neste caso, compila e cria um executável `olamundo`:

```
user@linux:SO$ gcc -o olamundo olamundo.c
```

O `gcc` possui muitas opções de linha de comando. Na seguinte encontram-se listadas algumas dessas opções e respetivas descrições:

Argumento	Descrição
`-c`	compila para código objeto (extensão `.o`)
`-g`	gera informação de *debug* (necessária para o *debugger*)
`-E`	apenas efetua o processo de pré-processamento
`-S`	efetua o processo de compilação, gerando o assembly (ficheiro com extensão `.s`)
`-l<nome>`	*linka* (liga) ficheiros objeto com a biblioteca (*library*) `<nome>`. Por exemplo, para *linkar* com a biblioteca de matemática `libm.a` é necessário especificar `-lm`, por exemplo: `gcc fich1.c -o fich1 -lm`
`-Wall`	compila com os avisos (*warning*) mais importantes ativados. Recomenda-se que todos os programas sejam compilados com essa opção
`-W` ou `-Wextra`	ativa ainda mais avisos do que `-Wall`
`-Wconversion`	alerta para as conversões numéricas implícitas (p.ex. de **`int`** para **`double`**, *etc.*)
`-Wmissing-prototypes`	alerta para as funções empregues para as quais não tenha sido indicado protótipo

> ***IMPORTANTE***
>
> a compilação deve ser **sempre** feita com os avisos ativados, isto é, especificando-se as opções `-Wall` e `-W` (ou `-Wextra`).
>
> ```
> user@linux:SO$ gcc -Wall -W -o olamundo olamundo.c
> ```

Por omissão, o compilador GCC utilizado nas aulas, adota a norma `gnu11`. Esta norma consiste na norma `C11` mais um conjunto de extensões da GNU. Para forçar a compilação estrita para `C11`, é necessário incluir as seguintes opções na linha de comando:

```
user@linux:SO$ gcc -std=c11 -pedantic -o olamundo olamundo.c
```

Para saber mais sobre o `gcc`:

- `man gcc`
- `info gcc` (necessita instalação do pacote gcc-doc)
- Manual eletrónico `gcc` da FSF-GNU

2.2 *Linkagem* com bibliotecas

De modo a permitir a reutilização de código, grande parte das distribuições GNU/Linux disponibilizam bibliotecas de funções que são instaladas como pacotes ou bibliotecas. Para utilizar estas bibliotecas tem que se dizer ao compilador que as ligue (*link*) ao nosso programa e para isso deve usar-se a opção: `-l<nome_da_biblioteca>`.

```
1 user@linux:SO$ gcc -o program main.o -lm
```

Com a linha de comandos anterior, está a ser *linkada* ao nosso programa a biblioteca de funções matemáticas através da opção `-lm`. A opção `-l<nome_da_biblioteca>` procura um ficheiro chamado `lib<nome_da_biblioteca>` nas diretorias de bibliotecas do sistema operativo. Para o exemplo anterior, seria possível, **mas não recomendado**, substituir o comando por:

```
1 user@linux:SO$ gcc -o program main.o /usr/lib/x86_64-linux-gnu/libm.a
```

2.3 Lab 1

Gere o programa executável do código-fonte listado na Listagem 1.

> ***NOTA***
>
> Para executar o ficheiro resultante poderá ser necessário indicar o caminho, fazendo referência à diretoria atual, ou seja, `./olamundo`

3 Linguagem C

De seguida apresentam-se dois tópicos importantes quando se faz programação em `C`.

3.1 Argumentos da função `main()`

Em `C`, é a partir da função `main()` que se podem recolher os parâmetros passados na linha de comando, daí a referida função adotar a seguinte sintaxe:

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

- `argc` - contém o número de argumentos passados para o programa. Caso não seja passado nenhum parâmetro, argc terá o valor 1. argv[0] terá sempre o nome do executável.
- `argv` - contém uma lista de parâmetros passados, incluindo o nome do executável.

Em baixo está um programa em C que recebe parâmetros de entrada:

Listagem 2: Ficheiro args.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("\n\n");
    printf("Numero de argumentos: %d\n", argc);
    printf("\n");
    printf("Listagem dos argumentos\n");

    for (int i = 0; i < argc; i++) {
        printf("\tArgumento[%d]= %s\n", i, argv[i]);
    }
    printf("\n");

    return 0;
}
```

3.2 Lab 2

Gere o programa executável do programa da Listagem 2 e execute o mesmo com diferentes argumentos.

3.3 Inclusão de ficheiros - referências cíclicas

A linguagem C permite escrever um qualquer programa dividindo-o logicamente por vários ficheiros de código-fonte. Para se fazer uso dos ficheiros de código a partir do principal, é usada a diretiva de pré-processamento `#include`. Ao nível da compilação, deverão ser fornecidos ao compilador todos os ficheiros `.c` na fase de compilação e todos os ficheiros `.o` na fase de *linkagem*, conforme mostra a Figura 4:

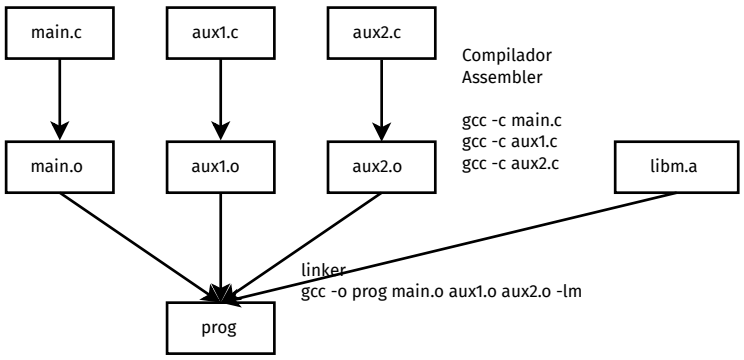

Figura 4: Funcionamento do `gcc`

Nesta abordagem de escrever programas, é comum criarem-se referências cíclicas, ou seja, o ficheiro `A` incluir o `B` e o `B` incluir o `A` originando erros de compilação. A fim de evitar este problema, deverão existir preocupações na codificação de forma a só permitir a inclusão de cada ficheiro apenas uma vez. Isto pode ser conseguido utilizando a directiva `#define` do pré-processador no ficheiro `.h`, da seguinte forma:

```
#ifndef _NOME_H_
#define _NOME_H_

    // protótipos
    // tipos de dados

#endif
```

em que `NOME` designa o nome do ficheiro.

3.4 Lab 3

1. Tendo em conta o seguinte código-fonte listado em baixo (ver Listagens 3, 4, 5, 6, 7), crie o ficheiro `compile.sh` que permita:
   - compilar individualmente cada ficheiro `.c` para um ficheiro `.o`;
   - criar o executável `lab3` com base nos ficheiros `.o` criados na alínea anterior;
2. Adicione ao ficheiro `funcoes_aux.h`, a função `raiz()`, que calcula a raiz quadrada do resultado de `div_e_soma()`.
   - Invoque a função no ficheiro `main.c`.

- Volte a compilar o programa. Confirme que não foi possível criar o executável e corrija o problema.

Listagem 3: Ficheiro main.c

```c
#include <stdio.h>

#include "funcoes.h"
#include "funcoes_aux.h"

int main(void) {
    float numA, numB;
    printf("Introduza um numero: ");
    scanf("%f", &numA);
    printf("Introduza outro numero: ");
    scanf("%f", &numB);
    printf("\nA soma = %f", soma(numA, numB));
    printf("\ndiv_e_soma = %f\n", div_e_soma(numA, numB));
    return 0;
}
```

Listagem 4: Ficheiro funcoes.h

```c
#ifndef _FUNCOES_H_
#define _FUNCOES_H_

float soma(float a, float b);

#endif
```

Listagem 5: Ficheiro funcoes.c

```c
#include "funcoes.h"

float soma(float a, float b) { return a + b; }
```

Listagem 6: Ficheiro funcoes_aux.h

```c
#ifndef _FUNCOES_AUX_H_
#define _FUNCOES_AUX_H_

float div_e_soma(float dividendo, float divisor);

#endif
```

Listagem 7: Ficheiro funcoes_aux.c

```c
#include "funcoes_aux.h"
#include "funcoes.h"

float div_e_soma(float dividendo, float divisor) {
    if (divisor == 0)
        return soma(dividendo, divisor);

```

```
8       return (dividendo / divisor) + soma(dividendo, divisor);
9   }
```

4 Boas práticas

A legibilidade do código-fonte é diretamente proporcional à utilização de boas práticas. Nesta secção serão mencionadas algumas boas práticas transversais a qualquer linguagem de programação. Recomenda-se a leitura do livro "The Art of Readable Code" para uma análise mais profunda sobre esta temática.

4.1 Condições de proteção

Todas as condições excecionais deverão utilizar uma condição de proteção (**return** prematuro) em vez de utilizar condições em cascata. Segue-se um exemplo da função `str_compare()` com um só ponto de saída:

```
1   int str_compare(char *str1, char *str2) {
2       int result = 0;
3       if (strlen(str1) < strlen(str2)) {
4           result = -1;
5       } else {
6           if (strlen(str1) > strlen(str2)) {
7               result = 1;
8           } else {
9               for (int i = 0; i < strlen(str1) && result == 0; ++i) {
10                  if (str1[i] < str2[i]) {
11                      result = -1;
12                  } else if (str1[i] > str2[i]) {
13                      result = 1;
14                  }
15              }
16          }
17      }
18      return result;
19  }
```

O código anterior é difícil de ler por obrigar a analisar 4 níveis de indentação para avaliar corretamente o fluxo da função.

Utilizando o conceito de condições de proteção, o código anterior poderia ser reescrito da seguinte forma:

```
1   int str_compare(char *str1, char *str2) {
2       if (strlen(str1) < strlen(str2)) {
3           return -1;
4       }
5
6       if (strlen(str1) > strlen(str2)) {
7           return 1;
8       }
9
10      for (int i = 0; i < strlen(str1); ++i) {
```

```
11          if (str1[i] < str2[i]) {
12              return -1;
13          }
14          if (str1[i] > str2[i]) {
15              return 1;
16          }
17      }
18      return 0;
19  }
```

O código anterior não só permite uma leitura mais simples e rápida como também possui um máximo de 2 níveis de indentação. Um aspeto curioso desta prática é que a *keyword* `else` deixa de ser relevante.

4.2 Máximo de 2 níveis de indentação por função

Uma função não deverá possuir mais do que **2 níveis** de indentação. Se tal ocorrer deve-se decompor novamente a função.

4.3 Single Responsibility Principle (SRP)

Uma função apenas deve implementar uma e uma só funcionalidade.

4.4 Don't Repeat Yourself (DRY)

Um programa não deve ter blocos repetidos que implementam a mesma funcionalidade. Sempre que tal ocorrer deve ser criada uma função.

5 Exercícios

5.1 Programa `opposites`

Escreva o programa `opposites` que converte uma *string* para o respetivo oposto. O programa deverá suportar os seguintes pares de opostos: (big <=> small, short <=> tall, high <=> low). Implemente o exercício num único ficheiro com o nome `main.c`. Segue-se um exemplo da execução do programa:

```
1  user@linux:SO$ ./opposites high
2  low
3  user@linux:SO$ ./opposites low
4  high
5  user@linux:SO$ ./opposites asdasd
6  'asdasd' word not found!
```

5.2 Programa vowels

Escreva o programa `vowels` que recebe por parâmetro uma lista de palavras e que para cada palavra apresente na saída padrão o respetivo número de vogais. Implemente o exercício num único ficheiro com o nome `main.c`. Segue-se um exemplo da execução do programa:

```
user@linux:SO$ ./vowels Sistemas Operativos
Sistemas: 3 vowels
Operativos: 5 vowels
```

5.3 Programa vowels_v2

Adicione ao exercício anterior uma função para calcular o número de consoantes. Nesta nova versão `vowels_v2`, coloque as funções relacionadas com o cálculo do número de vogais e consoantes num ficheiro com o nome `string_utils.c`. Coloque no ficheiro `string_utils.h` apenas as funções que são utilizadas no ficheiro `main.c`. Crie o ficheiro `compile.sh` que permita compilar todos os ficheiros e criar o executável `vowels_v2`. O programa deverá exibir o seguinte comportamento:

```
user@linux:SO$ ./vowels_v2 Sistemas Operativos!
Sistemas: 3 vowels, 5 consonants
Operativos!: 5 vowels, 5 consonants
```

5.4 Programa vowels_v3

Adicione ao exercício anterior, uma função para transformar a própria *string* em notação *Basic leet* utilizando o seguinte mapeamento:

```
a => 4, e => 3, g => 6, i => 1, o => 0, s => 5, t => 7
```

Acrescente ainda ao programa a capacidade de receber um argumento adicional que permita indicar qual a funcionalidade pretendida. O argumento poderá tomar um dos seguintes valores: - `--vowels` (mostra as vogais) - `--consonants` (mostra as consoantes), ou - `--leet` (escreve a string na notação *leet*).

Caso não seja especificado nenhum argumento adicional, a aplicação deverá considerar as 3 funcionalidades. O programa deverá ainda escrever uma mensagem relativa à sua utilização caso não seja passado nenhum parâmetro. Exemplos de utilização:

```
user@linux:SO$ ./vowels_v3
Usage: ./vowels_v3 [--vowels | --consonants | --leet] string1 [[string 2]...]

user@linux:SO$ ./vowels_v3 --vowels
Usage: ./vowels_v3 [--vowels | --consonants | --leet] string1 [[string 2]...]

user@linux:SO$ ./vowels_v3 --vowels '|Sistemas|Operativos|'
|Sistemas|Operativos|: 8 vowels

user@linux:SO$ ./vowels_v3 --consonants '|Sistemas|Operativos|'
```

```
|Sistemas|Operativos|: 10 consonants

user@linux:SO$ ./vowels_v3 --leet '|Sistemas|Operativos|'
|Sistemas|Operativos|: |S1573m45|Op3r471v05| [basic leet]

user@linux:SO$ ./vowels_v3 '|Sistemas|Operativos|'
|Sistemas|Operativos|: 8 vowels, 10 consonants, |S1573m45|Op3r471v05| [basic
    leet]
```

Ao compilar o programa poderão ser apresentados alguns avisos. Discuta com o professor a sua causa e possível resolução.