Ficha 3 – Programação em C

Introdução à programação

Tópicos abordados:

– Linguagem C (edição, código fonte e compilação)

– Como compilar um programa C (ambiente Linux)

– Projetos compostos por vários ficheiros

– Documentação

– Exercícios

Duração prevista: 1 aula

©2020: {miguel.negrao, vitor.carreira, patricio, mfrade,
loureiro, nfonseca, rui, nuno.costa }@ipleiria.pt

1 Introdução

1.1 Linguagem C e ferramentas GNU

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 7.3 do GCC.

1.2 Edição

Durante a resolução das fichas, os estudantes podem utilizar os programas *gedit* (para ambientes GNOME), *kwrite* (para ambientes KDE) ou *leafpad* (para ambientes LXDE) como editores gráficos de código fonte. Estes editores encontram-se disponíveis no ambiente de trabalho. Em alternativa poderão instalar os editores Sublime Text 2/3 ou Visual Studio Code, bastante em voga. 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:

```
$ gedit ficheiro &
```

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. .

1.2.1 Partilha de ficheiros entre o Lubuntu (sistema hóspede) e o Windows (sistema hospedeiro)

No VMWare, a partilha de ficheiros pode ser ativada através do menu "Virtual Machine Settings" (ver Figura), ativando-se a opção "Shared Folders", selecionando-se o diretório da máquina local a partilhar com a máquina virtual.

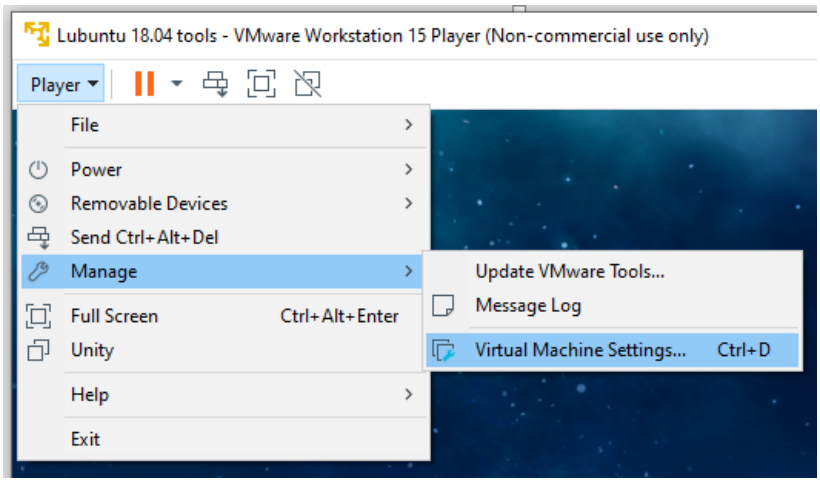

Figura 1 – Menu VMware

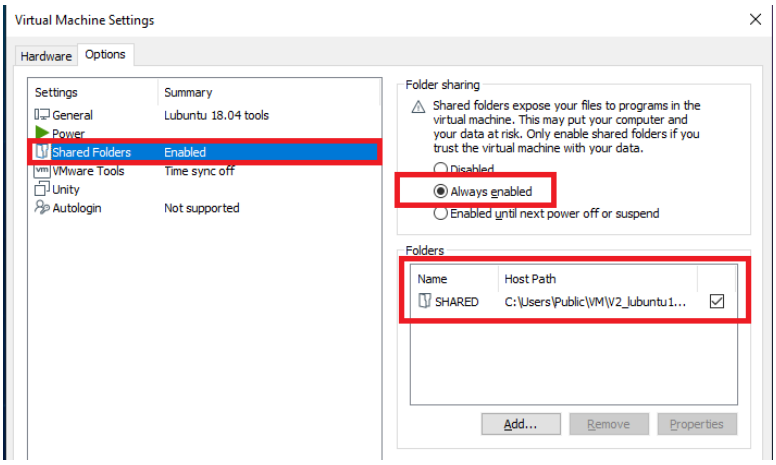

Figura 2 – Partilha de pasta no VMware

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.

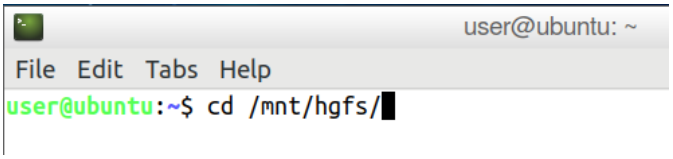

Figura 3 – Acesso ao diretório partilhado através da linha de comando no Linux

1.2.2 Edição com base no Windows

É possível fazer a edição de ficheiros com editores do Windows, como o *notepad++*, *ultraedit*, etc., recorrendo ao diretório partilhado para alojar os ficheiros. Deve ter em atenção, ao utilizar editores do Windows, com a quebra de linha que o editor utiliza, pois, os editores do Windows utilizam os caracteres '\10' e '\13' (CRLF) enquanto o UNIX utiliza apenas o caractere '\10' (CR).

De seguida são apresentados dois ficheiros com o mesmo conteúdo, para verificar a formatação nos dois sistemas:

Ficheiro Windows – teste.txt:

Teste<CR><LF>

Aos ficheiros em Windows<CR><LF>

Ficheiro Linux – teste.txt:

Teste<LF>

Aos ficheiros em Linux<LF>

Nota: <CR> = Carriage Return = '\13' e <LF> = Line Feed = '\10'

Os editores de texto mais avançados detetam automaticamente o tipo de terminação de linha. Para editores mais antigos ou de linha de comando, pode-se utilizar o utilitário do UNIX *fromdos* para resolver este problema. De seguida apresenta-se o modo de funcionamento deste comando:

```
$ fromdos <ficheiro_a_converter>
```

Nota: para aceder aos utilitários de conversão é necessário instalar o pacote **tofrodos**

```
$ sudo apt install tofrodos
```

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 "Ola mundo…" para o ecrã.

```
#include <stdio.h>
int main(void)
{
   printf("Ola mundo...\n");
   return 0;
}
```

Figura 4 – Código fonte do programa "Olá mundo…"

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*.

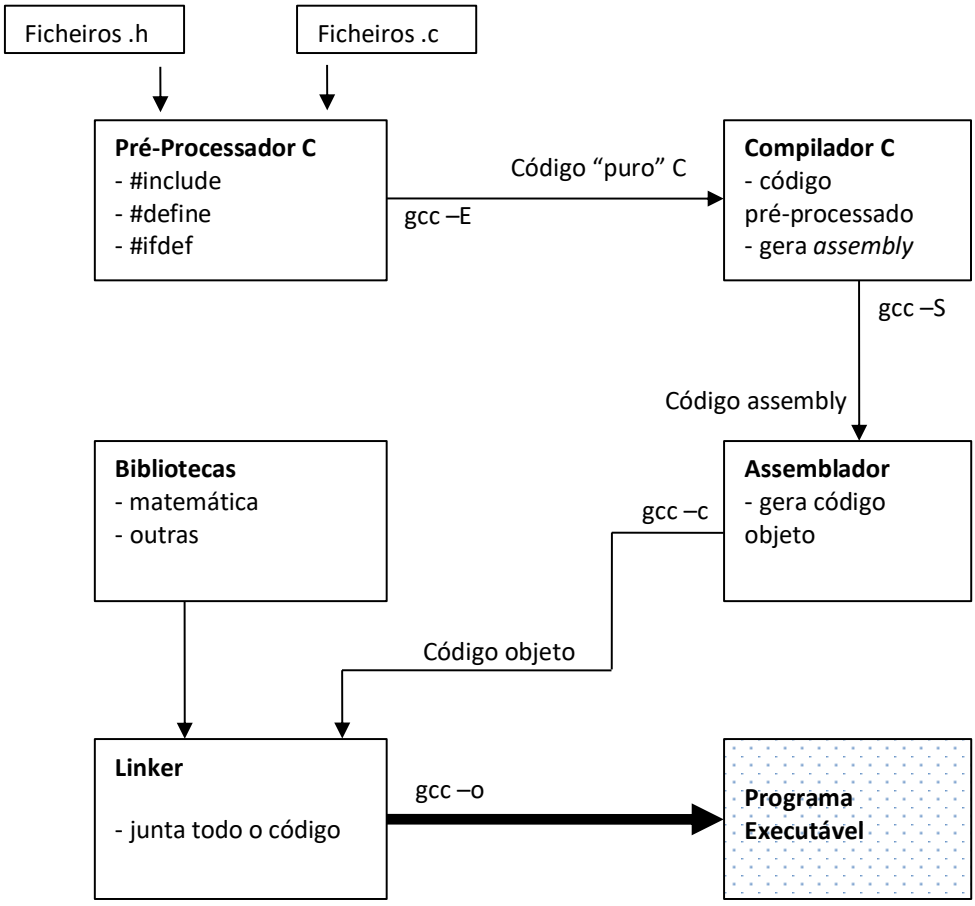

Figura 5 – Fluxograma com o processo de compilação

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 *makefiles*. A compilação usando *makefiles* 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 *makefiles* 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***.

```
$ gcc -c MostraArgv.c
```

Compila e cria o ficheiro objeto "MostraArgv.o"

```
$ gcc –o olamundo olamundo.c
```

Compila e cria um executável "olamundo"

O ***gcc*** possui muitas opções de linha de comando. Na Tabela 1 encontram-se listadas algumas dessas opções e respetivas descrições.

`-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*" ficheiros objeto com a biblioteca (*library*) **nome**. Por exemplo, para "*linkar"* com a biblioteca de matemática – libm.a – é necessário especificar -lm 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.
`-Wextra ou -W`	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.

Tabela 1 – Algumas opções do gcc

Importante: a compilação deve ser **sempre** feita com os avisos ativados, isto é, especificando-se as opções *`-Wall`* e *`-Wextra`* *(ou `-W`)*.

```
$ gcc -Wall -Wextra -o olamundo olamundo.c
```

Por omissão, o compilador GCC utilizado nas aulas (versão 7.3), 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:

```
$ gcc -std=c11 -pedantic -Wall -Wextra -o olamundo olamundo.c
```

2.1.1 Para saber mais

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

(http://www.gnu.org/software/gcc/onlinedocs/)

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>.

```
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:

```
gcc -o program main.o /usr/lib/libm.a
```

Lab 1

Gere o programa executável do código fonte da Figura 1.

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

3 Linguagem 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:

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

Figura 6 – Função *main*

- **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.

```
#include <stdio.h>
int main(int argc, char *argv[]){
   printf("\n\n");
   printf("Número 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;
}
```

Figura 7 – Programa que recebe parâmetros de entrada

Lab 2

Gere o programa executável do programa da figura anterior e execute o mesmo com diferentes argumentos.

3.2 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*.

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
```

Figura 8 - Directiva *#define*

em que NOME designa o nome do ficheiro.

Lab 3

Tendo em conta o seguinte código fonte:

```
#include <stdio.h>
#include "funcoes.h"
#include "funcoes_aux.h"

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

Figura 9 - Ficheiro main.c

```
#ifndef _FUNCOES_H_
#define _FUNCOES_H_
  float soma(float a, float b);
#endif
```

Figura 10 - Ficheiro funcoes.h

```
#include "funcoes.h"

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

Figura 11 - Ficheiro funcoes.c

```
#ifndef _FUNCOES_AUX_H_
#define _FUNCOES_AUX_H_
  float div_e_soma(float dividendo, float divisor);
#endif
```

Figura 12 - Ficheiro funcoes_aux.h

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

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

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

Figura 13 - Ficheiro funcoes_aux.c

Acrescente ao laboratório, um ficheiro (**compile.sh**) que permita:

1) Compilar individualmente cada ficheiro **.c** para um ficheiro **.o**;

2) Criar o executável **lab3** com base nos ficheiros **.o** criados na alínea anterior;

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.

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*" (http://shop.oreilly.com/product/9780596802301.do) 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. Exemplo:

```
int str_compare(char *str1, char *str2) {
    int result = 0;
    if (strlen(str1) < strlen(str2)) {
        result = -1;
    } else {
```

```
        if (strlen(str1) > strlen(str2)) {
             result = 1;
        } else {
            for (i = 0; i < strlen(str1) && result == 0; ++i) {
                if (str1[i] < str2[i]) {
                    result = -1;
                } else if (str[i] > str2[i]) {
                    result = 1;
                }
            }
        }
    }
    return result;
}
```

Figura 14 – Função str_compare: versão com um só ponto de saída.

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:

```
int str_compare(char *str1, char *str2) {
    if (strlen(str1) < strlen(str2)) {
        return -1;
    }

    if (strlen(str1) > strlen(str2)) {
        return 1;
    }

    for (i = 0; i < strlen(str1); ++i) {
        if (str1[i] < str2[i]) {
            return -1;
        }
        if (str[i] > str2[i]) {
            return 1;
        }
    }
    return 0;
}
```

Figura 15 – Função str_compare: versão com condições de proteção.

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

Uma função apenas deve implementar uma e uma só funcionalidade.

4.4 DRY (Don't Repeat Yourself)

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

5 Exercícios

1- 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**".

```
$ ./opposites high
low
$ ./opposites Low
high
$./opposites asdasd
'asdasd' word not found!
```

2- Escreva o programa **vogals** 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**".

```
./vogals Sistemas Operativos
Sistemas: 3 vogals
Operativos: 5 vogals
```

3- Adicione ao exercício anterior uma função para calcular o número de consoantes. Nesta nova versão (**vogals_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 **vogals_v2**. O programa deverá exibir o seguinte comportamento:

```
./vogals_v2 Sistemas Operativos!
Sistemas: 3 vogals, 5 consonants
Operativos!: 5 vogals, 5 consonants
```

4- 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: `--vogals` (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:

```
./vogals_v3
Usage: ./vogals_v3 [--vogals | --consonants | --leet] string1 [[string 2]...]

./vogals_v3 --vogals
Usage: ./vogals_v3 [--vogals | --consonants | --leet] string1 [[string 2]...]

./vogals_v3 --vogals '|Sistemas|Operativos|'
|Sistemas|Operativos|: 8 vogals

./vogals_v3 --consonants '|Sistemas|Operativos|'
|Sistemas|Operativos|: 10 consonants

./vogals_v3 --leet '|Sistemas|Operativos|'
|Sistemas|Operativos|: |S1573m45|Op3r471v05| [basic leet]

./vogals_v3 '|Sistemas|Operativos|'
|Sistemas|Operativos|: 8 vogals, 10 consonants, |S1573m45|Op3r471v05| [basic leet]
```

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