**Ficha 4 – Programação em C**

**As ferramentas `make` e `gengetopt`**

**Tópicos abordados:**

– *Makefiles* e a ferramenta *make*

– Macros

– Tratamento de erros e depuração

– Exercícios

**Duração prevista: 2 aulas**

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

# 1 Makefiles e a ferramenta `make`

A compilação de um programa que esteja contido num único ficheiro é bastante simples. Mas se o programa estiver dividido em vários ficheiros, os passos necessários para obter o executável aumentam. Vamos supor que um programa está dividido em três ficheiros: `main.c`, `iodat.c` e `dorun.c`, com os ficheiros ".h": `iodat.h` e `dorun.h`. Para compilar o programa será necessário utilizar os seguintes comandos:

```
gcc -c main.c
```

Cria objeto do *main* (`main.o`)

```
gcc -c iodat.c
```

Cria objeto do *iodat* (`iodat.o`)

```
gcc -c dorun.c
```

Cria objeto do *dorun* (`dorun.o`)

```
gcc -o program main.o iodat.o dorun.o
```

Fase de *linkagem*, em que é criado o executável `program`

Assim, sempre que quisermos compilar o programa, teremos de escrever os quatro comandos acima indicados. Poderia utilizar-se um *script* para automatizar o processo, mas o facto é que sempre que se alterasse um ficheiro (".c" ou ".h"), todos os ficheiros do projeto seriam recompilados, mesmo que isso não fosse necessário. Imagine-se as consequências deste comportamento num projeto com uma centena ou mesmo milhares de ficheiros de código fonte.

Como facilmente se depreende, essa solução é muito pouco prática e pouco cuidadosa no que respeita aos recursos. De modo a automatizar a gestão de programas com vários ficheiros com código fonte, emprega-se a ferramenta *make*.

A ferramenta *make* recebe informação referente às dependências e às formas de construir programas através de um ficheiro descritivo, que, por omissão, se designa por *makefile* ou *Makefile* (embora o *make* permita qualquer nome para o ficheiro de descrição de dependências).

Cada instrução num ficheiro *makefile* é constituída por duas partes:

- A linha das dependências, com o seguinte formato:

<alvo> : <linha_de_dependências>

- Um ou mais comandos com texto, que têm que começar sempre por um <TAB>.

Para o exemplo acima apresentado, é criado o ficheiro chamado *makefile* com a seguinte estrutura:

```
# ficheiro makefile
program:main.o iodat.o dorun.o
    gcc -o program main.o iodat.o dorun.o

main.o:main.c iodat.h dorun.h
    gcc –c main.c

iodat.o:iodat.c iodat.h
    gcc –c iodat.c

dorun.o:dorun.c dorun.h
    gcc –c dorun.c
```

**Listagem 1 - Exemplo de um *makefile***

Por exemplo, na primeira instrução (inicia-se na 2.ª linha do ficheiro) é indicado que o programa depende de três ficheiros. Ao digitar-se "*make program*" na linha de comando, antes de executar os comandos, o *make* verifica primeiro cada uma das dependências. Se alguma das dependências não existir ou não estiver atualizada (o ficheiro respetivo foi alterado) é compilada, sendo que só depois é que o comando é executado. Da Listagem 1 também se depreende que o conteúdo de uma linha após o caractere # é interpretado como comentário.

## 1.1 Grafo de dependências

De seguida, está representado um grafo de dependências do exemplo anterior. De salientar o uso das cores para melhor transmitir a ideia das dependências entre ficheiros nas várias fases do processo de compilação.

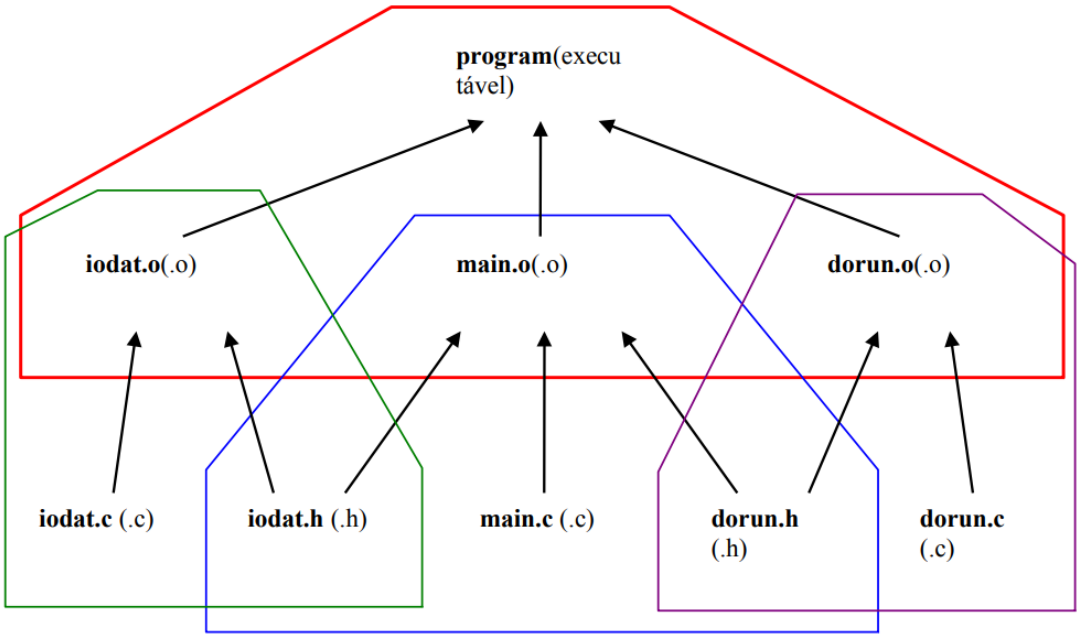


**Figura 1 - Grafo de dependências do makefile da Listagem 1**

**Atenção: num ficheiro descritivo *make* (*makefile*) as linhas com comandos têm de começar sempre por um *tab*.**

Para verificar se um ficheiro *makefile* está correto no que respeita a *tabs*, pode executar-se o seguinte comando:

```
cat -Te <nome-do-ficheiro-make>
```

A opção `-T` faz com que os *tabs* apareçam como ^I;

A opção `-e` leva a que seja colocado um $ no fim de linha.

Para executar o ficheiro *Makefile* basta chamar o comando *make*. Por omissão (só especificando o comando *make*), é executada a primeira entrada do ficheiro *makefile*. No caso anterior o comando *make* é equivalente ao comando "`make program`". No entanto, pode especificar-se quais as opções a executar do *makefile*.

```
$ make
$ make program
$ make iodat.o
```

**Listagem 2 - exemplos de execução da ferramenta `make`**

O comportamento do utilitário *make* pode ser alterado através das opções passadas pela linha de comando. Para o efeito sugere-se que consulte a página do manual eletrónico (`man make`).

**Dica:** a opção **`-n`** do utilitário `make` (`make -n`) permite executar o `make` *a seco* (*dry run* na designação anglo-saxónica), isto é, o `make` somente apresenta os comandos que iria executar, não os executando na realidade.

## 1.2 Macros

No ficheiro *Makefile* podem ser definidas macros (ou variáveis "macro"), cujo comportamento se assemelha às macros do pré-processador da linguagem C. A definição de uma macro processa-se do seguinte modo:

```
<nome_da_macro> = string
```

**Listagem 3 - Definição de uma macro**

A sua utilização segue a seguinte sintaxe:

```
$(nome_da_macro)
```

**Listagem 4 - Utilização de uma macro**

Segue-se um exemplo, com base no exemplo da Listagem 1. Neste caso, foi definida a macro OBJS que contém a lista de todos os ficheiros objetos (**.o**) necessários à criação do executável:

```
OBJS = iodat.o main.o dorun.o

program: $(OBJS)
  gcc –o program $(OBJS)
(...)
```

**Listagem 5 - Exemplo macro**

É ainda frequente o uso de macros para especificar opções a serem passadas ao compilador. Por exemplo, as opções para ativação de avisos (*warnings*, ou seja `-Wall` e `-Wextra`) e acréscimo de informação de depuração (*debugging*, opção `-g`), são frequentemente especificadas através de uma macro denominada CFLAGS.

```
# ficheiro makefile
# macros
OBJS = main.o iodat.o dorun.o
CFLAGS = -std=c11 -Wall –Wextra –g

program:$(OBJS)
    gcc -o program $(OBJS)

main.o:main.c iodat.h dorun.h
    gcc $(CFLAGS) –c main.c

iodat.o:iodat.c iodat.h
    gcc $(CFLAGS) –c iodat.c

dorun.o:dorun.c dorun.h
    gcc $(CFLAGS) –c dorun.c
```

**Listagem 6 - *makefile* com macros `objs` e `CFLAGS`**

***Lab 1***

a) Elabore um ficheiro *Makefile* que possa automatizar a compilação do programa *vogals_v2* (exercício da ficha anterior). Relembra-se que o programa *vogals_v2* é composto por três ficheiros de código: `main.c`, `string_utils.h` e `string_utils.c`. O ficheiro *Makefile* deve ter a macro **OBJS** contendo todos os ficheiros objetos (**.o**) necessário à criação do executável. O ficheiro *Makefile* deve ainda declarar a macro **CFLAGS** contendo as opções de compilação a serem passadas ao compilador.

b) Execute o utilitário *make* com a opção -d (*make* -d). O que é que sucede? **Resposta:** é apresentada (vasta) informação de debugging.

## 1.3 Macros definidas internamente

As seguintes macros encontram-se definidas internamente pelo *make*:

| Macro | Descrição |
|---|---|
| `$(CC)` | Refere-se ao compilador de C que está definido por omissão no sistema. |
| `$@` | Refere-se ao alvo atual (o ficheiro que está a ser gerado). Designada por ***output variable*** na terminologia anglo-saxónica. |
| `$?` | Refere-se sempre à lista de dependências. Esta macro deve ser usada com muito cuidado, porque é substituída apenas pela lista de ficheiros que foram alterados e não por todos os que estão na lista. Se esta situação não for devidamente acautelada, o *makefile* pode ter um comportamento diferente do esperado. |
| `$^` | Refere-se sempre à lista de dependências quer tenham sido alteradas ou não. |
| `$<` | Refere-se à 1ª dependência, usualmente o ficheiro de código fonte. Designada por ***input variable*** na terminologia anglo-saxónica. |

**Tabela 1 - Macros definidas internamente**

```
# ficheiros objeto (.o)
OBJS = iodat.o main.o dorun.o

program: $(OBJS)
  $(CC) –o $@ $^
```

**Listagem 7 - Exemplo do uso de macros internas**

No exemplo anterior o ***$(CC)*** é substituído por *gcc*, ***$@*** substituído por *program* (o nome para programa executável) e ***$^*** pela lista de todas as dependências que neste caso são: *iodat.o*, *main.o* e *dorun.o*. Sendo assim, o comando a executar seria:

```
$ gcc -o program iodat.o main.o dorun.o
```

## 1.4 Regras de sufixos

Todos os programas em C devem ter extensão ".c" e os ficheiros objeto devem ter extensão ".o". O utilitário *make* reconhece estas regras, pelo que não é necessário especificá-las no *makefile*.
Se forem criados ficheiros ".h" (*header files*), as dependências entre os ".c" e os ".h" devem ser indicadas para que, no caso de se alterar um ficheiro ".h", a compilação seja forçada. O compilador *gcc* pode ser empregue para criar a lista de dependências de qualquer ficheiro ".c" através da opção -MM. Por exemplo, para criar a lista de dependências de todos os ficheiros ".c" do diretório corrente, executa-se:

```
$ gcc -MM *.c
```

Em projetos de média/grande dimensão é comum o primeiro alvo ser sempre *all* e existir um alvo *clean* que limpa todos os ficheiros objeto, ficheiros *core dump* e outros não necessários.
A versão final para o ficheiro *makefile* apresentado anteriormente seria:

```
# flags para o compilador
CFLAGS = -std=c11 -Wall -Wextra -g

# Bibliotecas
LIBS = -lm

# ficheiros objeto
OBJS = iodat.o main.o dorun.o

PROGRAM = program
all: $(PROGRAM)

$(PROGRAM): $(OBJS)
  $(CC) -o $@ $(OBJS) $(LIBS)

# Lista de dependências dos ficheiros código fonte
# Pode ser obtida com gcc -MM *.c
main.o:main.c iodat.h dorun.h
iodat.o:iodat.c iodat.h
dorun.o:dorun.c dorun.h
# Indica como transformar um ficheiro .c num ficheiro .o
.c.o:
  $(CC) $(CFLAGS) -c $<

# remove ficheiros sem interesse
clean:
  rm -f *.o core.* *~ $(PROGRAM)
```

**Listagem 8 - Um makefile mais completo**

**Dica 1:** quando se pretende compilar um programa constituído por apenas um ficheiro fonte (e.g. simples.c), pode conseguir-se o efeito desejado através da execução de `make simples`. As regras por omissão do *make* permitem-lhe realizar a compilação.

**Dica 2:** o comando `gcc -MM *.c` escreve para o terminal a lista de dependências dos ficheiros código fonte do diretório corrente.

**Dica 3:** a opção `-p` ou `--print-data-base` do utilitário *make* (`make -p`) mostra no terminal as regras internas definidas por omissão pelo *make*.

***Lab 2***

Acrescente ao ficheiro de *makefile* criado no Lab1 a entrada ***clean***, cujo propósito é a de remover todos os ficheiros criados pela execução do *make*, bem como os ficheiros *core* e ainda os ficheiros cujo nome termina por ~. A entrada *clean* deve ser ativada especificando-se "`clean`" na chamada ao utilitário `make`, isto é:

```
$ make clean
```

## 1.5 Documentação utilitário make

- *man make*
- Livro "*Managing Projects with GNU make*", 3rd edition, O'Reilly, 2005 – disponível online em http://www.oreilly.com/openbook/make3/book/index.csp

# 2 Canal de erro padrão – stderr

É prática comum na programação que as mensagens de erro sejam escritas para o canal de erro padrão, também designado por *stderr* (*standard error*). Na linguagem C, a escrita para o canal de erro padrão pode ser feita através da função ***fprintf***, especificando-se ***stderr*** como destino da mensagem. A função ***fprintf*** é muito semelhante à função ***printf***. A única diferença é que a função ***fprintf*** efetua a escrita formatada para o canal que lhe for indicado como primeiro parâmetro, enquanto a função ***printf*** efetua a escrita no terminal (*stdout*). Em caso de erro devolve EOF. Caso contrário, devolve o número de carateres escritos (exceto os terminadores '\0' das *strings*).

```
fprintf(stderr,"mensagem para o canal de erro\n");
```

# 3 Funções para tratamento de erros e depuração

Sempre que se cria um programa existe a possibilidade de este conter erros (designados em inglês por *bugs*). Uma das técnicas mais vulgar para detetar erros é através do recurso à função *printf()* para imprimir o valor de variáveis ou simplesmente para detetar até onde é que a execução do programa correu sem problemas. Embora esta abordagem seja um pouco limitada, podemos aumentar o seu potencial se a informação que for mostrada tiver conteúdo importante, em vez de, por exemplo, "estou aqui!". Outro aspeto importante é o tratamento de erros, cuja inexistência é uma das causas mais comuns de problemas.

Assim, de forma a sistematizar a depuração e o tratamento de erros decorridos da execução de chamadas ao sistema, criaram-se três funções e três macros (estas macros são diferentes das macros dos *makefiles*). Para obter uma informação mais detalhada destas funções e macros deve **consultar o anexo "Tratamento de erros e depuração"**.

De seguida apresentam-se vários exemplos de uso de cada uma das macros.

## 3.1 DEBUG

O objetivo da macro DEBUG é ser usada para imprimir o valor de variáveis e outra informação que o programador ache útil para detetar erros. Esta macro é semelhante ao *printf*, ou seja, o número de parâmetros de entrada é variável consoante a *string* de formatação. Note-se que a macro acrescenta automaticamente o nome do ficheiro e o número da linha onde foi chamada, bem como um "\n" (mudança de linha) no final da string.

**Dica:** o acesso ao nome do ficheiro de código fonte é feito através da constante de pré-processador __FILE__. O acesso ao número da linha do ficheiro de código fonte é feito através da constante de pré-processador __LINE__.

```
contador = 10;

#ifdef SHOW_DEBUG  /* macro do pré-processador */
  DEBUG ("o valor da variável é %d", contador);
#endif /* macro do pré-processador */
...
```

**Listagem 9 - Exemplo do uso da macro DEBUG**

O resultado da execução do código anterior é o seguinte:

```
[exemplo.c@20] DEBUG - o valor da variável é 10
```

## 3.2 WARNING

O objetivo da macro WARNING é ser usada para o tratamento de erros de chamadas ao sistema, mas que não impliquem o término do programa. A chamada desta macro é igual à macro DEBUG, a diferença reside na informação que é acrescentada automaticamente. Neste caso, além do nome e número da linha do ficheiro, é acrescentada também a descrição correspondente ao erro guardado na variável global *errno*.

A variável global *errno* é utilizada pelas chamadas de sistema e algumas funções de bibliotecas para indicar qual o erro que ocorreu. Em caso de erro, estas normalmente devolvem o valor -1 ou NULL e atribuem um valor inteiro diferente de zero à variável *errno*. O valor atribuído comunica qual é o erro que ocorreu, existindo macros que associam um nome a cada valor numérico. Por exemplo, a chamada de sistema *open* poderá atribuir o valor EACCES a *errno* o que indica que o programa não tem permissões para aceder ao ficheiro. A cada valor de *errno* está também associada uma *string* que descreve o erro em maior detalhe.

```
pid= 1;
if (kill(pid, 0) != 0)
  WARNING("kill do processo %d", pid);
...
```

**Listagem 10 - Exemplo do uso da macro WARNING**

O resultado da execução do código anterior é:

```
[exemplo.c@17] WARNING - kill do processo 1: Operation not permitted
```

## 3.3 ERROR

A macro ERROR é muito semelhante à WARNING, a diferença reside no facto de conter mais um argumento, o *exit_code*. O objetivo desta macro é ser usada no tratamento de erros de chamadas ao sistema em que, no caso da ocorrência de um erro, não faça sentido a continuação do programa. Assim, esta macro imprime a informação desejada e termina o programa com uma chamada à função *exit(exit_code);*.

```
int fd;
char *file = "/etc/passwd";
...
if ((fd = open(file, O_CREAT | O_EXCL)) == -1)
  ERROR(1, "O ficheiro %s já existe", file);
...
```

**Listagem 11 - Exemplo do uso da macro ERROR**

O resultado da execução do código anterior é:

```
[exemplo.c@20] ERROR - O ficheiro /etc/passwd já existe: File exists
```

## 3.4 Exemplo

De seguida apresenta-se um pequeno exemplo que faz o uso das macros descritas atrás e o respetivo *makefile*.

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include "debug.h"

int main(void){
  int fd;
  char *file = "/etc/passwd";
  int a = 234, pid = 1;
  float b = 3.1415;

  printf("Exemplo de tratamento de erros e depuração\n\n");
  #ifdef SHOW_DEBUG
    DEBUG("O valor de 'a' é: %d, e o valor de 'b' é:%.4f",a,b);
  #endif

  if (kill(pid, 0)!=0){
    WARNING("kill do processo %d (nao termina o programa)",pid);
  }
  if ((fd=open(file, O_CREAT | O_EXCL)) == -1){
```

```
    ERROR(1, "O ficheiro %s já existe", file);
  }

  printf("nunca aparece, porque open dá sempre erro\n");
  return 0;
}
```

**Listagem 12 - Ficheiro exemplo.c**

```
# flags para o compilador
CFLAGS = -std=c11 -Wall -W
# ficheiros objetos

OBJS = exemplo.o debug.o
PROGRAM = program

all: $(PROGRAM)
debugon: CFLAGS += -D SHOW_DEBUG
debugon: $(PROGRAM)

$(PROGRAM): $(OBJS)
  $(CC) -o $@ $(OBJS)

# Lista de dependências dos ficheiros código fonte
debug.o: debug.c debug.h
exemplo.o: exemplo.c debug.h

# Indica como transformar um ficheiro .c num ficheiro .o
.c.o:
  $(CC) $(CFLAGS) -c $<
```

**Listagem 13 - *Makefile* da listagem anterior com entradas para depuração**

### *Lab 3*

Gere o programa executável do exemplo anterior através da respetiva *makefile*. Compare as diferenças entre o alvo *all* e o alvo *debugon*.

### *Lab 4*

Tendo em conta o seguinte código fonte:

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

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

**Listagem 14 - Ficheiro main.c**

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

**Listagem 15 - Ficheiro funcoes.h**

```
#include "funcoes.h"

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

**Listagem 16 - Ficheiro funcoes.c**

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

**Listagem 17 - Ficheiro funcoesaux.h**

```
#include "funcoesAux.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);
}
```

**Listagem 18 - Ficheiro funcoesaux.c**

Acrescente ao laboratório, uma *makefile* que facilite a compilação do programa. Utilize a seguinte linha de comando para determinar a lista de dependências:

```
gcc -MM *.c
```

# 4 Utilitário gengetopt

O utilitário *gengetopt* permite gerar, automaticamente, uma função C capaz de interpretar os argumentos da linha de comandos. O funcionamento deste programa baseia-se na interpretação de um ficheiro de configuração. É usual que o nome do ficheiro de configuração tenha a extensão **.ggo**. O formato do ficheiro de configuração é descrito de seguida.

## 4.1 Ficheiro de configuração

O ficheiro de configuração é um ficheiro normal, de texto, onde para cada parâmetro do programa se faz corresponder uma linha neste ficheiro. Um parâmetro formato longo é identificado por "--" (e.g., --all), enquanto um parâmetro curto tem apenas um "-" (e.g, -a).

```
purpose "Ficheiro de configuração para um programa exemplo"
package "Nome do programa"
version "Versão"
option "Opção_formato_longo" Opção_formato_curto (letra) "Descrição" tipo (string, int, etc) "valor_omissão" obrigatório (yes ou no)
```

O parâmetro "*valor por omissão*" não é obrigatório e para escrever comentários deve usar-se o caractere #. Depois de se ter criado o ficheiro de configuração será necessário compilar o mesmo para gerar o código fonte a utilizar no nosso programa. Para isso deve-se utilizar o seguinte comando:

```
$gengetopt < config.ggo
```

**Nota:** pode ser necessário instalar o utilitário *gengetopt* através da seguinte linha de comando.

```
sudo apt-get install gengetopt
```

De seguida é apresentado um exemplo. Vamos supor que estamos a escrever um programa que poderá ter os seguintes parâmetros de entrada:

**--nome** "Escola Superior de Tecnologia e Gestão de Leiria"

**--valor** 170

**--idade** 7

**--habitantes** 123232122

**--tonelagem** 12.5

**--margem** 0.000005

Neste cenário, deverá ser criado um ficheiro de configuração com a seguinte informação:

```
# config.ggo
# Ficheiro de configuração do programa SO

purpose "Este programa tem como objetivo…"
package "SO"
version "1.0"

# Options
option  "nome"       n "Parâmetro nome"       string     no
option  "valor"      v "Parâmetro valor"      int       yes
option  "idade"      i "Parâmetro idade"      short      no
option  "habitantes" b "Parâmetro habitantes" long      yes
option  "tonelagem"  t "Parâmetro tonelagem"  float      no
option  "margem"     m "Parâmetro margem"     double     no
```

**Listagem 19 - Exemplo de um ficheiro de configuração (config.ggo)**

Cada linha iniciada por *option* define uma opção. Assim, a 2.ª coluna define o nome longo da opção (e.g., *--nome*), a 3.ª coluna define o nome curto da opção ("-n"), a 4ª coluna define o texto a ser mostrado quando é ativada a opção de ajuda (*--help*) da aplicação. A 5.ª coluna identifica o tipo de dado ao qual deve obedecer o parâmetro fornecido pelo utilizador. Por exemplo, no caso do parâmetro idade, o parâmetro a fornecer pelo utilizador deverá ser um inteiro do tipo *short.* Assim, se o utilizador providenciar um valor para o parâmetro idade que não seja um inteiro (e.g, *--idade ABC*), o código gerado pelo `gengetopt` irá terminar a aplicação indicando que o valor indicado para o parâmetro não corresponde ao tipo de dado definido. Finalmente, a 6.ª coluna indica se o parâmetro é obrigatório (*yes*) ou não (*no*).

De seguida executa-se o utilitário `gengetopt` para que seja criado um ficheiro ".h" e um ficheiro ".c" com código capaz de gerir os parâmetros passados. Note-se que o utilitário `gengetopt` processa o conteúdo que lhe é indicado através do canal de entrada padrão (stdin – *standard input*), pelo que é necessário fazer uso do redirecionamento de entrada através do símbolo "<".

```
$ gengetopt < config.ggo
```

A execução do `gengetopt` gera, por omissão, os ficheiros "**cmdline.h**" e "**cmdline.c**". A opção ***--file-name=***<*nome pretendido*> permite especificar outro nome para os ficheiros. Para se fazer uso do código, criado automaticamente, basta incluir o *header* file, por exemplo no ficheiro principal, e na função *main* testar que parâmetros foram passados, da seguinte forma:

```
#include "cmdline.h"
struct gengetopt_args_info args_info;
if (cmdline_parser(argc,argv,&args_info) != 0){
  exit(1);
}
```

**Listagem 20 - Utilização das funções criadas automaticamente**

Se tudo correr bem a estrutura *args_info* será preenchida com os parâmetros válidos.

No exemplo seguinte, é testado se os parâmetros opcionais (*--nome* e *–valor*) foram fornecidos via linha de comando. Esta verificação é feita com recurso ao sufixo *_given.* Caso não tenham sido enviados, a condição terá o valor lógico **falso**. Caso contrário, é possível obter os respetivos valores usando o sufixo *_arg*. Por exemplo:

```
if (args_info.nome_given){
  printf("%s",args_info.nome_arg);
}

if (args_info.valor_given){
  printf("%d",args_info.valor_arg);
}
```

**Listagem 21 - Utilização da estrutura devolvida pelas funções do gengetopt**

Dado que o código criado pelo `gengetopt` procede à alocação de recursos é necessário proceder à libertação dos recursos. Para o efeito, deve-se fazer uso da função *cmdline_parser_free()*, que tem o seguinte protótipo:

```
void cmdline_parser_free (struct gengetopt_args_info *args_info);
```

A Listagem 22 apresenta um exemplo de chamada à função `cmdline_parser_free`.

```
#include "cmdline.h"
...
struct gengetopt_args_info args_info;
if (cmdline_parser(argc,argv,&args_info) != 0){
  exit(1);
}
...
/* Libertar dos recursos afetos ao gengetopt */
cmdline_parser_free(&args_info);
```

**Listagem 22 – Uso das funções cmdline_parser e cmdline_parser_free disponibilizadas pelo gengetopt**

### *Lab 5*

Implemente a aplicação **exemplo_opt**. Esta deve disponibilizar as opções definidas no ficheiro *config.ggo* (Listagem 19) através do ficheiro *main.c*.

# 5 Makefile final

A Listagem 23 apresenta o *makefile* que faz parte do *template* de projeto em linguagem C da unidade curricular.

```
# Easily adaptable makefile
# Note: remove comments (#) to activate some features
# author Vitor Carreira
# date 2010-09-26 / updated: 2016-03-15 (Patricio)

# Libraries to include (if any)
LIBS=#-lm -pthread

# Compiler flags
CFLAGS=-std=c11 -Wall -Wextra #-ggdb #-pg
# Linker flags
LDFLAGS=#-pg

# Indentation flags
# IFLAGS=-br -brs -brf -npsl -ce -cli4 -bli4 -nut
IFLAGS=-linux -brs -brf -br

# Name of the executable
PROGRAM=prog

# Prefix for the gengetopt file (if gengetopt is used)
PROGRAM_OPT=args

# Object files required to build the executable
PROGRAM_OBJS=main.o debug.o memory.o $(PROGRAM_OPT).o

# Clean and all are not files
.PHONY: clean all docs indent debugon

all: $(PROGRAM)

# activate DEBUG, defining the SHOW_DEBUG macro
debugon: CFLAGS += -D SHOW_DEBUG -g
debugon: $(PROGRAM)

# activate optimization (-O...)
OPTIMIZE_FLAGS=-O2 # values (for gcc): -O2 -O3 -Os -Ofast
optimize: CFLAGS += $(OPTIMIZE_FLAGS)
optimize: LDFLAGS += $(OPTIMIZE_FLAGS)
optimize: $(PROGRAM)

$(PROGRAM): $(PROGRAM_OBJS)
  $(CC) -o $@ $(PROGRAM_OBJS) $(LIBS) $(LDFLAGS)

# Dependencies
main.o: main.c debug.h memory.h $(PROGRAM_OPT).h
$(PROGRAM_OPT).o: $(PROGRAM_OPT).c $(PROGRAM_OPT).h

debug.o: debug.c debug.h
memory.o: memory.c memory.h

#how to create an object file (.o) from C file (.c)
.c.o:
```

```
  $(CC) $(CFLAGS) -c $<
#  Generates  command  line  arguments  code  from  gengetopt
configuration file
$(PROGRAM_OPT).h: $(PROGRAM_OPT).ggo
  gengetopt < $(PROGRAM_OPT).ggo --file-name=$(PROGRAM_OPT)

clean:
  rm  -f  *.o  core.*  *~  $(PROGRAM)  *.bak  $(PROGRAM_OPT).h
$(PROGRAM_OPT).c

# run doxygen
docs: Doxyfile
  doxygen Doxyfile

# create the Doxyfile configuration file for doxygen
Doxyfile:
  doxygen -g Doxyfile

# entry to create the list of dependencies
depend:
  $(CC) -MM *.c

# entry 'indent' requires the application indent
# (sudo apt-get install indent)
indent:
  indent $(IFLAGS) *.c *.h

# entry to run the pmccabe utility (computes the "complexity" of
# the code). Requires the application pmccabe
# (sudo apt-get install pmccabe)
pmccabe:
  pmccabe -v *.c

# entry to run the cppcheck tool
cppcheck:
  cppcheck --enable=all --verbose *.c *.h
```

**Listagem 23 - Makefile final (com *doxygen* e mais alguns utilitários)**

***Lab 6***

Recorrendo ao *template* de projeto em linguagem C da unidade curricular, indique:

a) O que é que é executado com **make indent**?

b) O que é que é executado com **make depend**?

c) O que é que é executado com **make pmccabe**?

d) O que é que é executado com **make cppcheck**?

# 6 Exercícios

## Aula

1- Escreva o programa **conta_letra** que deve receber dois parâmetros: uma *string* e uma letra, respetivamente. O programa deverá mostrar na saída padrão o número de vezes que a letra ocorre na *string*. Caso o programa seja lançado sem os dois parâmetros (por exemplo, apenas é indicado um parâmetro), o programa deve apresentar uma mensagem de erro no canal de erro padrão. O código fonte do programa deve estar organizado nos ficheiros **main.c** e **conta_letra.c/.h**. A gestão da compilação deve ser feita através da adaptação do ficheiro de *makefile* empregue na unidade curricular.

2- Recorrendo à linguagem C, escreva o programa **conta_letra_v2** cujo funcionalidade é idêntica ao programa da alínea anterior. Contudo, o programa **conta_letra_v2** deve estar preparado para processar os seguintes parâmetros da linha de comando:

--string <string> ou -s <string> parâmetro obrigatório do tipo string

--letra <caractere> ou -c <caractere> parâmetro obrigatório do tipo caractere

--help exibe ajuda

A gestão dos parâmetros deve ser feita através do utilitário *gengetopt*. Deve ainda ser empregue o *template* de projeto disponibilizado na unidade curricular.

3- Como sabe, uma variável inteira com *n* bits permite representar valores inteiros entre 0 e $2^n$-1. Por exemplo, com 5 bits conseguem-se representar os valores inteiros entre 0 e 31. Considerando bytes, tem-se que um byte (i.e., oito bits) permite representar valores entre 0 e 255 ($2^8$-1), dois bytes (16 bits) permitem a representação entre 0 e 65535 e assim por diante.

   a. Pretende-se que implemente, em linguagem C, a função `bytes_for_int` que devolve o número de bytes necessário para representar um valor inteiro cujo máximo seja `max_value`. O protótipo da função é:
   ```
   int bytes_for_int(unsigned int max_value);
   ```
   A função deve ser desenvolvida no ficheiro de código **bytes_for_int.c**.

   b. Elabore a aplicação `bytes_for_int` que deve receber obrigatoriamente o parâmetro **--num/-n <number>** através da linha de comando. Caso o parâmetro seja um número inteiro positivo, a aplicação mostra na saída padrão o número de bytes necessários para representar o parâmetro *number*. Caso não seja passado parâmetro ou esse não seja conforme, a aplicação deve terminar com uma mensagem de erro apropriada.

A aplicação deve ser implementada com recurso i) ao projeto *template* da unidade curricular e ii) ao utilitário `gengetopt` para processamento dos parâmetros da linha de comando.