Makefile

Laboratórios de Informática III
Guião #2-1

Departamento de Informática
Universidade do Minho

2024/25
Luís Ferreira, Nuno Faria

Contents

1 Estrutura básica

Um makefile é um ficheiro utilizado para facilitar a compilação de programas C de maior dimensão. Ele permite não só especificar os vários passos de compilação de uma forma legível, mas também garantir que ao recompilar um programa, apenas os ficheiros que sofreram alterações são novamente compilados, tornando o processo mais eficiente. É executado através do comando `make`, e deve ter o nome `Makefile` para que possa ser corretamente identificado pelo comando [1]. Este ficheiro é composto por um conjunto de regras. De um modo geral, cada regra especifica como devem ser compilados os ficheiros de código-fonte e como ligar (*"linkar"*) esses ficheiros objeto de forma a serem produzidos um ou mais executáveis[2]. Ou seja, do processamento bem sucedido de um Makefile resulta normalmente a produção de um ou mais programas executáveis (bem como de um conjunto de ficheiros intermédios).

```
objetivo: pré-requisitos
    receita
```

Cada regra é composta por um objetivo, um ou mais pré-requisitos, e uma receita. Os **pré-requisitos** são um conjunto de ficheiros (.o ou .c) e/ou objetivos de outras receitas necessários à execução de uma regra. O **objetivo** é normalmente constituído por um ficheiro objeto ou executável que se obtém da execução da receita. A **receita** é um conjunto de instruções/comandos do interpretador de comandos (*"shell"*) que, quando aplicados aos ficheiros descritos pelos pré-requisitos, resultam na produção do objetivo. Os pré-requisitos e objetivo devem estar escritos na mesma linha. A receita pode ocupar várias linhas, uma por cada comando a executar. Cada linha de uma receita tem necessariamente de começar por um **tab (muito importante!)**.

```
# O executável main tem como pré-requisito a existência do ficheiro main.c.
# A receita compila o ficheiro main.c, tendo como resultado o executável denominado main.
main: main.c
    gcc main.c -o main
```

Diferentes regras podem ser encadeadas através dos pré-requisitos. Para executar uma regra, é necessário que todos os seus pré-requisitos sejam preenchidos. Para isso, é possível que seja necessário executar outras regras de forma a atingir esses pré-requisitos.

```
# Esta regra precisa do ficheiro main.o, pelo que irá executar a regra abaixo
main: main.o
    gcc main.o -o main

# Regra que produz (tem como objetivo) o ficheiro main.o
main.o: main.c
    gcc -c main.c -o main.o
```

Ao chamar o comando `make`, sem qualquer argumento, a primeira regra no ficheiro é executada. Por convenção, à primeira regra é dado o nome de *all*. Quando o comando `make` é executado com um argumento, a regra cujo objetivo coincide com esse argumento é executada.

```
all: main

# Esta regra necessita do ficheiro main.o, pelo que irá executar a regra abaixo
main: main.o
    gcc main.o -o main

# Regra que produz (tem como objetivo) o ficheiro main.o
main.o: main.c
    gcc -c main.c -o main.o

clean:
    rm -f main main.o
```

No excerto acima, o comando `make` produz o executável com o nome main. Já o comando `make clean` irá executar a última regra que remove os ficheiros *main* e *main.o*.

[1] É possível especificar um nome de ficheiro diferente usando a opção `-f <filename>` como argumento da linha de comando

[2] Estas regras podem também descrever outros "produtos" do projeto, tais como, por exemplo, documentação, relatórios, etc.

2 Compilação e linkagem

Por motivos de eficiência e legibilidade, deve fazer-se a separação entre passos de compilação de ficheiros objeto, e o processo de linkagem. Para isso, faz-se uso de três **variáveis**, CC, CFLAGS e LDFLAGS para, respetivamente: especificar o compilador a usar; especificar as *flags* a usar para compilação; e especificar as *flags* a usar para linkagem. As variáveis são definidas no início do ficheiro, antes da primeira regra, e podem ser usadas nas regras utilizando o formato `$(NOME_VARIAVEL)`.

```
# Variável CC define o compilador a usar
CC=gcc
# Variável LDFLAGS define flags de linkagem
LDFLAGS=
# Variável CFLAGS define flags de compilação. Notar a opção -c para gerar ficheiros .o
CFLAGS=-Wall -c

# O executável main tem como pré-requisito a existência dos ficheiro main.o e deque.o.
# A receita faz a linkagem dos ficheiros objeto, utilizando as variáveis de flags
#     de linkagem LDFLAGS, tendo como resultado o executável denominado main.
# Tem como pré-requisitos os ficheiros main.o e deque.o, cujas regras serão executadas
#     caso esses ficheiros não existam ou os seus ficheiros fonte tenham sido alterados

all: main

# Passo de linkagem
main: main.o deque.o
    $(CC) main.o deque.o $(LDFLAGS) -o main

# Passo de compilação
main.o: main.c
    $(CC) $(CFLAGS) main.c

# Passo de compilação
deque.o: deque.c
    $(CC) $(CFLAGS) deque.c -o deque.o

# Comando de limpeza
clean:
    rm -f deque.o main.o main
```

3 Regras implícitas e variáveis especiais

Existe um conjunto de regras (e receitas) implícitas que são aplicadas por omissão[3] no processamento de um Makefile nos casos em que este não as especifica de forma explícita. Por exemplo, no caso de regras que têm como objetivo a compilação de código-fonte e geração dos respetivos ficheiros objeto (.o) a seguinte lógica é usada.

```
main.o: main.c

#Por omissão executa a mesma receita que:
main.o: main.c
    $(CC) $(CPPFLAGS) $(CFLAGS) -c main.c -o main.o
```

Assim, podemos escrever:

```
CC=gcc
LDFLAGS=
```

[3] A lista completa de regras e receitas implícitas e explícitas aplicadas no processamento de um Makefile pode ser obtido através da inclusão da opção `-p` nos argumentos do comando `make`.

```
CFLAGS=-Wall -c


all: main

# Passo de linkagem
main: main.o deque.o
    gcc main.o deque.o $(LDFLAGS) -o main

# Passo de compilação
main.o: main.c

# Passo de compilação
deque.o: deque.c

# Comando de limpeza
clean:
    rm deque.o main.o main
```

Existe também um conjunto de variáveis especiais que podem, opcionalmente, ser usadas para tornar o código mais uniforme e sucinto. Por exemplo:

- `$@` - objetivo da regra.
- `$^` - todos os pré-requisitos separados por espaço.
- `$<` - primeiro pré-requisito.

Assim, o passo de linkagem do exemplo anterior pode ser escrito da seguinte forma:

```
# Passo de linkagem
main: main.o deque.o
    gcc $^ $(LDFLAGS) -o $@
```

4 Extra: opções de caminhos e bibliotecas para compilação

É importante salientar que, para projetos de maior dimensão, as opções `-I -L -l` do gcc são de especial importância para garantir que os ficheiros são corretamente compilados e linkados.

Na fase de compilação, é necessário que os ficheiros consigam encontrar corretamente os ficheiros .h dos quais fazem *include*. Para isso, deve ser usada a opção `-I`.

- `-Idir` adiciona a diretoria `dir` à lista de diretorias a pesquisar no momento de compilação de um ficheiro. As diretorias especificadas com esta opção têm prioridade em relação às pré-definidas pelo sistema.

De notar que **não deve existir nenhum espaço entre a opção e o caminho da diretoria especificada**. Por defeito, o compilador procura por ficheiros *header* nos caminhos pré-definidos pelo sistema, e na mesma pasta em que o comando gcc está a ser executado. Para utilizar ficheiros *header* que não estejam em nenhum destes caminhos (e.g., ficheiros *header* definidos localmente dentro de sub-pastas) é necessário utilizar esta opção.

No momento de linkagem, é necessário indicar ao compilador onde se encontram as bibliotecas que efetivamente implementam as funções declaradas nos ficheiros *header*, e que bibliotecas devem ser consideradas. Para isso, são usadas as opções `-L -l`.

- -Ldir adiciona a diretoria *dir* à lista de diretorias a pesquisar para `-l`.
- `-llibrary` incluir a biblioteca *library* no processo de linkagem.

O `gcc` considera um conjunto de caminhos de bibliotecas e de bibliotecas pré-definido, durante o processo de linkagem. Para utilizar bibliotecas que não as pré-definidas, e.g., glib, é necessário utilizar as opções acima descritas. É importante notar que **a ordem com que é feita a linkagem pode afetar o processo de compilação**. As bibliotecas e ficheiros objetos dependentes devem ficar à esquerda das bibliotecas das quais dependem na sequência de linkagem. Veja-se o exemplo abaixo:

```
// A biblioteca glib é usada pelo módulo "glib_hash_table_test",
//  logo deve surgir após esse módulo no processo de linkagem

    gcc main.o glib_hash_table_test.o -lglib2.0 -o hash_table_test
```

Vejamos um exemplo que requer o uso destas opções. Considere a seguinte estrutura de ficheiros:

```
programa
    -| Makefile
    -| src
        --| main.c
        --| deque.c
    -| includes
        --| deque.h
```

Imaginemos ainda que a estrutura deque faz uso da glib.
O ficheiro `Makefile` tomaria a seguinte forma:

```
CC=gcc
# diretoria "includes" e diretoria para headers da glib adicionados às diretorias onde
#    o compilador deve procurar por headers
CFLAGS=-Wall -c -Iincludes -Ipath/to/glib/include
# Diretoria da biblioteca do glib e biblioteca glib adicionadas às diretorias onde
#    o compilador deve procurar por bibliotecas
LDFLAGS=-Lpath/to/glib/library -lglib-2.0

all: main

# Passo de linkagem
main: src/main.o src/deque.o
    gcc src/main.o src/deque.o $(LDFLAGS) -o main

# Passo de compilação
src/main.o: src/main.c

# Passo de compilação
src/deque.o: src/deque.c

# Comando de limpeza
clean:
    rm src/deque.o src/main.o main
```

A ferramenta `pkg-config` é útil na gestão das opções acima mencionadas. Uma vez que os caminhos de bibliotecas e de ficheiros *header* de bibliotecas instaladas variam de sistema para sistema, esta ferramenta permite que os caminhos sejam corretamente definidos, desde que a ferramenta esteja instalada e corretamente configurada no sistema (disponível por omissão em Ubuntu). As opções `--cflags` e `--libs` devem ser usadas para obter as configurações corretas para os processos de compilação e linkagem, respetivamente. Para além disso, deve ser especificada a bibliteca que se quer configurar.

Assim, para o caso da biblioteca `glib` do exemplo anterior, as variáveis poderiam ser definidas da seguinte forma:

```
CC=gcc
# De notar que o comando pkg-config deve estar envolto pelos acentos
CFLAGS=-Wall -c -Iincludes -Ipath/to/glib/include `pkg-config --cflags glib-2.0`
# Diretoria da biblioteca do glib e biblioteca glib adicionadas às diretorias onde
#    o compilador deve procurar por bibliotecas
LDFLAGS=`pkg-config --libs glib-2.0`
```

5 Exemplo completo

```
# Compiler used
CC=gcc

# Flags to use when compiling
#  - Wall: show all warnings
#  - Iinclude: directory for the local header files
#  - `pkg-config --cflags glib-2.0`: glib header files directories (optional, only needed if using glib)
CFLAGS=-Wall -Iinclude `pkg-config --cflags glib-2.0`

# Flags to use when linking
#  - `pkg-config --libs glib-2.0`: glib library (optional, only needed if using glib)
#  - `-lm`: math library (optional, only needed if using glib)
LDFLAGS=`pkg-config --libs glib-2.0` -lm


# Executable called 'program'.
#
# To generate it, we link all the .o files and the glib and math libraries.
# We use £(NAME) to use the variable NAME.
# £@ is equivalent to the target. In this case, £@ evaluates to: program
# £^ evaluates to the dependencies: In this case, it evaluates to: src/main.o src/file1.o src/file2.o
# In this example, we have the following rule:
#  gcc `pkg-config --libs glib-2.0` -lm -Wall -Iinclude `pkg-config --cflags glib-2.0` src/main.o src/file
#
# Since this is the first rule, it will be the one called when executing 'make' without any arguments.
# In other words, 'make' is equivalent to 'make program'.
#
# To execute this rule, the make will first need to execute rules 'src/main.o', 'src/file1.o',
# and 'src/file2.o', as specified in the list of dependencies (right)
program: src/main.o src/file1.o src/file2.o
	$(CC) $(CFLAGS) $^ $(LDFLAGS) -o $@

# Here we generate the object files for each source.
# An empty rule is equivalent to £(CC) £(CPPFLAGS) £(CFLAGS) -c <left param> <right param>.
# As such, we set the target (left) as the name of the object file to be generated
# and the name of the source file in the list of dependencies (right).

# This specific rule is equivalent to:
# gcc -Wall -Iinclude `pkg-config --cflags glib-2.0` -c -o src/main.o src/main.c
src/main.o: src/main.c

# Equivalent to the one above
#src/main.o: src/main.c
#	£(CC) £(CFLAGS) -c -o £@ £^

src/file1.o: src/file1.c

src/file2.o: src/file2.c


# The clean will remove the executable (program) and object files (*.o).
# To run it, we must execute 'make clean'
clean:
	rm -f program src/*.o
```