Sistemas Operativos

Programação em C para ambientes *Linux*

Patricio Domingues

Atualizado: 2019

Agenda

- Agenda
- Ciclo de desenvolvimento
- Compilação em C
- Préprocessador
- Utilitário make
- Sintaxe do makefile
- Exemplo de makefile

- "Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."
  - John F. Woods

Ciclo de desenvolvimento

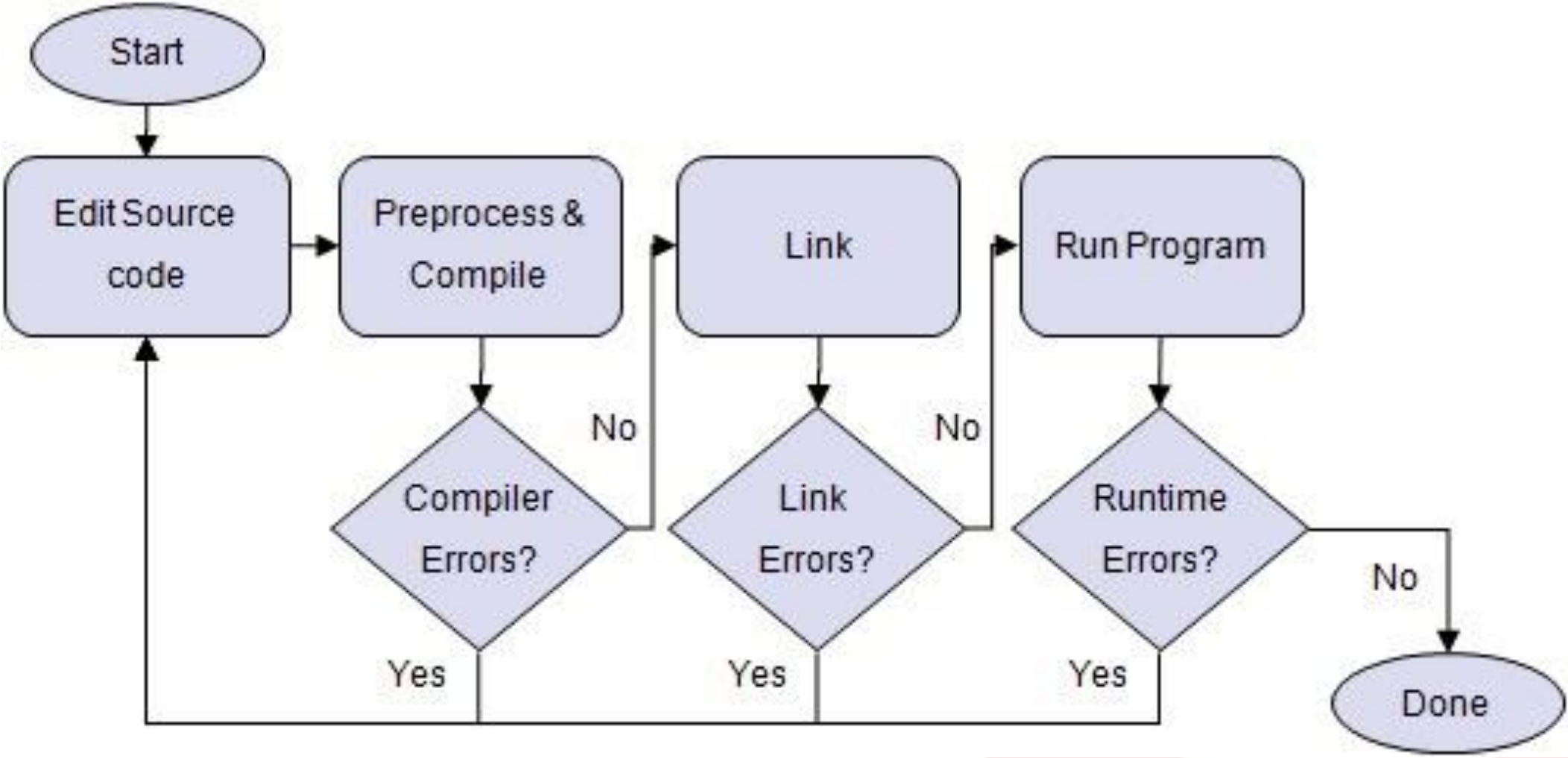


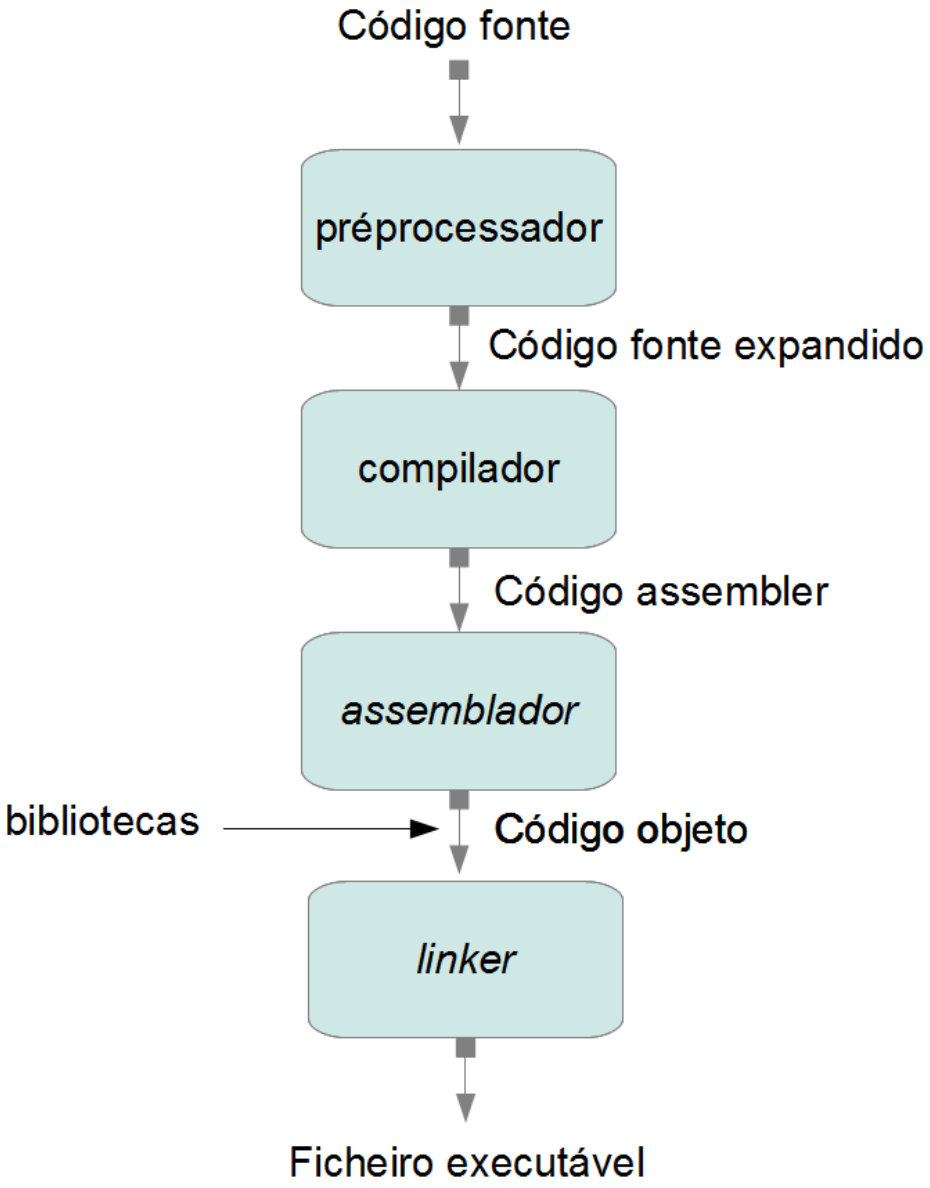
Imagem: http://w3processing.com/index.php?subMenuItemId=222

Compilação em C

- Um programa em C é composto por um ou mais ficheiros de código fonte (.c e os respetivos .h)
- Cada ficheiro é processado primeiro pelo pré-processador originando um ficheiro de texto
- O ficheiro de texto é processado pelo compilador (i.e., é compilado) originando um ficheiro de código objeto (*assembler*)
- Todos os ficheiros objetos são agregados pelo "linker", criando-se o ficheiro executável

Esquema >>

Ciclo de Compilação em C



Programa exemplo

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

- Compilação via gcc (linha de comando)
  - `gcc -g -Wall -Wextra prog.c -o prog.exe`
  - `-g: acrescenta info de depuração`
  - `-Wall: ativação dos warnings`
  - `-Wextra (ou –W): ativação de mais warnings`
  - `-o nome: nome do ficheiro executável`

- Próximos slides: análise às diferentes etapas

Após préprocessador

- gcc -E prog.c
  - Ativa apenas préprocessador
  - Produz ficheiro de código C com **1148** linhas

```
# 1 "helloWorld.c"
# 1 "<built-in>"
# 1 "<command line>"
(…)
# 2 "helloWorld.c" 2
int main(void)
{
 printf("Programa em C\n");
 return 0;
}
```

Após compilação

- `gcc -S prog.c`
  - opção -S: produz código "assembler"

```
.file   "helloWorld.c"
        .def    ___main; .scl    2;      .type   32;      .endef
        .section .rdata,"dr"
LC0:
        .ascii "Programa em C\12\0"
        .text
.globl _main
        .def    _main;   .scl    2;      .type   32;      .endef
_main:
        pushl   %ebp
(…)
        call    _printf
        movl    $0, %eax
        leave
        ret
        .def    _printf; .scl    3;      .type   32;      .endef
```

Préprocessador (1)

- Programa que processa os ficheiros de código fonte
- É executado antes da fase de compilação daí o prefixo de "pré"
  - Processa "diretivas" do pré-processador
  - As diretivas são identificadas por um "#"
- Exemplos

  #include <stdio.h>
  - A diretiva "#include" indica ao pré-processador que deve substituir literalmente a linha "#include <stdio.h>" pelo contéudo do ficheiro "stdio.h".
  - Os ficheiros ".h" contém protótipos de funções, definições de variaveis globais, etc.
  - Os ficheiros ".h" do sistema estão em /usr/include

Préprocessador (2)

- gcc -E prog.c
  - Opção -E: ativa apenas préprocessador
  - Produz ficheiro de código C com 1148 linhas

```
# 1 "helloWorld.c"
# 1 "<built-in>"
# 1 "<command line>"
(…)
# 2 "helloWorld.c" 2
int main(void)
{
 printf("Programa em C\n");
 return 0;
}
```

#include (1)

- A primitiva #include serve para incluir **ficheiros .h**
- Um ficheiro .h está (usualmente) associado a um **ficheiro .c**
  - Exemplo:
    - util.c >> util.h
- Um ficheiro .h contém:
  - Protótipos de funções "públicas"
  - Protótipos de variáveis globais "públicas"
  - Exemplos
    - **double CalculaSQRT(double); /* Protótipo */**
    - **extern int G_Contador; /* Variável global */**

Ficheiro .h (exemplo)

- Exemplo de ficheiro .h

```
#ifndef __UTIL_H__          /* Proteção contra "multi-includes" */
#define __UTIL_H__
typedef unsigned char           UINT8;    /* Definição de tipo (alias) */
typedef signed char             INT8;
typedef unsigned short          UINT16;
typedef signed short            INT16;
typedef unsigned int            UINT32;
typedef signed int              INT32;
typedef unsigned long long      UINT64;
typedef signed long long        INT64;
typedef unsigned int            size_t;

/**  Public functions - prototypes */
void serial_write(UINT8 *p_param);
int  serial_read(void);
size_t _strlen(const char* s);
int _strncmp(const char *s1, const char *s2, size_t n);
#endif
```

#include (2)

- Distinção na indicação do ficheiro <...> e "..."
  - **#include <ficheiroSistema.h>**
    - Compilador procura o ficheiro .h nos diretórios do sistema (/usr/include no linux)
  - **#include "ficheiroUtilizador.h"**
    - Compilador procura o ficheiro .h nos diretórios do utilizador (usualmente diretório corrente)
  - Exemplo

```
#include <stdio.h>
#include <math.h>
#include "util.h"
#include "compute.h"
```

Préprocessador (3) - constantes

- A directiva #define permita a definição de constantes que são substituídas pelo pré-processador
- Exemplo

```
#define CIDADE                    "LEIRIA"
```

- Define a constante CIDADE
- Sempre que surgir CIDADE no código fonte, o préprocessador faz a substituição por "LEIRIA"
- Por convenção, as constantes do préprocessador são definidas em maiúsculas

Préprocessador – constantes (4)

- Para as constantes que não tenham valor definido, pode ser definido um valor através da linha de comando
  - -DNOME_MACRO=VALOR
- Exemplo

```
#define VALOR
int main(void){
        printf("Valor=%d\n",VALOR);
        return 0;
}
```

gcc **-DVALOR=2** -Wall -Wextra a.c -o a.exe

  - VALOR é definido com 2

Préprocessador (5) - macros

- A directiva #define permita ainda a definição de macros que também são substituídas pelo pré-processador
- Uma macro **não** é uma função

  #define MAX(x,y) (x>y ? x : y)
- **MAX(x,y)** é uma macro com dois parâmetros
- Sempre que o pré-processador encontrar "MAX(x,y)" irá proceder à substituição pela expressão C "(x>y ? x : y)"

- **Exemplo**

  int a= MAX(10,20);

  Passa a...

  int a = (10>20? 10 : 20);

Exercício >>

Exercício

Exercício

Definir uma macro (nome: **ABS**) que devolve o valor absoluto de um número inteiro

- Sugestão: usar o operador *ternário*

#define ABS(a) **...**

Préprocessador (6) - condicional

- O pré-processador suporta directivas de *inclusão condicional*
- São directivas com início e fim, que delimitam um conjunto ou mais conjuntos de linhas de código
  - #if...#endif
  - #ifdef ... #endif
  - #ifndef... #endif
  - #ifdef...#else...#endif
- Muito empregue para código multiplataforma
- Exemplo

```
#ifdef WIN32          /* WIN32 identifica API windows */
   printf("WIN32\n");
#else
   printf("NOT WIN32\n");
#endif
```

Préprocessador (7) - exemplos

- Condicional

```
#define DEBUG
...
#ifdef DEBUG
    printf("mensagem qd DEBUG está definido\n");
#endif
```

Mais info: Capítulo 10,
"Practical C Programming", O'Reilly

- printf("Linha:%d, Ficheiro fonte: %s\n", __LINE__,__FILE__);
- __FILE__ e __LINE__ são definidos pelo pré-processador e correspondem ao ficheiro e à linha corrente

Préprocessador (8)

- Uso do "*#ifndef*" para precaver múltiplos includes

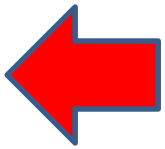
```
#ifndef __UTIL_H__
#define __UTIL_H__
(…)
#endif
```

- Sempre que se cria um ficheiro .h, deve-se fazer uso de:
  - #ifndef __NOME_FICHEIRO_H__
  - #define __NOME_FICHEIRO_H__
  - …
  - #endif

A macro `assert` (1)

- **`assert`**
  - Tradução "afirmar"
- Macro empregue para validar código
- Caso a validação falhe (**valor lógico FALSO**), a aplicação é terminada
- Exemplo

```
#include <stdio.h>
#include <assert.h>
size_t num_chars(const char *ptr){
        assert(ptr != NULL );
        const char *workptr = ptr;
        size_t num_elms = 0;
        while( workptr && (*workptr!='\0')){
                num_elms++;
        }
        return num_elms;
}
int main(void){
        char *Ptr = NULL;
        size_t num_elms = num_chars(Ptr);
        printf("num_elms='%zu'\n", num_elms);
        return 0;
}
```



```
a.exe: numchars.c:5:
num_chars: Assertion `ptr
!= ((void *)0)' failed.
Aborted (core dumped)
```

A macro `assert` (2)

- Destativar a macro assert
  - A macro assert não deve ser mantida em código de produção
  - Podem ser desativadas definindo-se a macro NDEBUG

- Desativar

```
gcc -DNDEBUG ...
```

OU no código

```
#define NDEBUG
```

static_assert – C11

- Norma C11 (2011)
- `static_assert( constant-expr, string-literal);`
  - Se a expressão for zero, então a mensagem com a string é mostrada
  - `static_assert` é avaliada durante a compilação
  - `assert` é avaliado em tempo de execução

- Exemplo

```
static_assert(sizeof(void*) == 4,
 "ERROR: 64-bit code not supported");
```

Utilitário make (1)

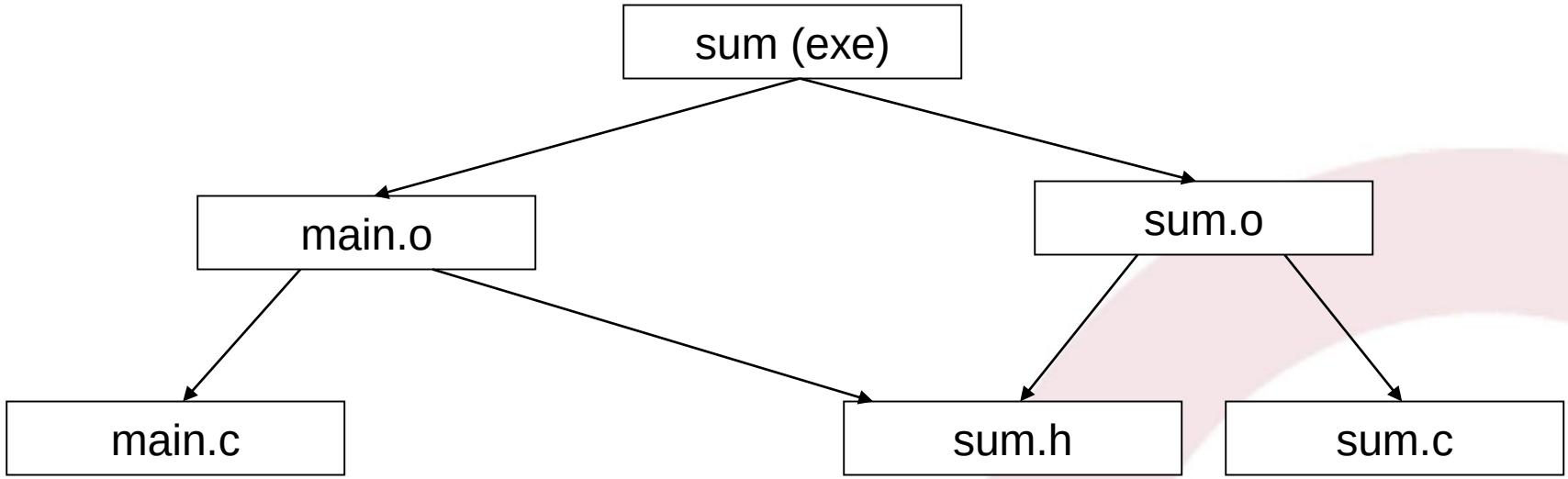
- É recomendado que um programa seja dividido em vários ficheiros de código fonte
  - (+) A alteração de um módulo requer apenas que esse módulo seja recompilado
  - (+) O tempo de compilação aumenta exponencialmente com o tamanho do ficheiro de código fonte
  - (+) A distribuição de um projeto por vários ficheiros permite que vários programadores estejam a mexer ao mesmo tempo no projeto (um programador por ficheiro)
- Como gerir a dependência entre ficheiros e automatizar o processo de compilação/linkagem?
  - Utilitário **make**

Utilitário make (2)

- O utilitário **make** interpreta um ficheiro designado de makefile
  - O ficheiro makefile contém:
    - Estrutura do projeto
      - Ficheiros e dependências
    - Instruções para a compilação do programa
- O make gere dependência entre ficheiros
  - Não está limitado a ficheiros de programas “c”

Utilitário make (3)

- Exemplo
  - Programa composto por 3 ficheiros
    - main.c, sum.c e sum.h
    - sum.h é empregue (via #include) em sum.c e main.c
    - O executável deverá chamar-se “sum”



Dependência entre ficheiros

.h e dependências

- Automatizar a determinação das dependências de um ficheiro de código c relativamente aos ficheiros .h
  - Ficheiros .h que são incluídos (#include) no ficheiro .c
- Uso do gcc com a opção -MM → **gcc -MM *.c**

- Exemplo
  - Template das aulas práticas
  - gcc -MM *.c
    - `debug.o: debug.c debug.h`
    - `main.o: main.c debug.h semaforos.h`
    - `semaforos.o: semaforos.c semaforos.h`

Ficheiro makefile

Ficheiro com 3 regras (# é indicador de comentário)

Regra = linha de dependência seguida de linha(s) de ação

```
sum: main.o sum.o
  gcc -o sum main.o sum.o
main.o: main.c sum.h
  gcc -c main.c
```

 # Regra para "sum.o"

```
sum.o: sum.c sum.h
  gcc -c sum.c
```

Sintaxe do makefile

```
main.o: main.c sum.h
        gcc -c main.c
```

Ponto 1: main.o é o ficheiro alvo, isto é, a linha descreve o que deve ocorrer quando main.o é mais antigo do que os dois ficheiros de que depende.

Ponto 2: main.o depende de main.c e sum.h

Ponto 3: a segunda linha indica qual é a ação a ser executada quando a regra é ativada (i.e., quando main.o está desatualizado). Neste caso é chamado o gcc para compilar main.c, reconstruindo o ficheiro main.o

Ponto 4: a segunda linha inicia-se SEMPRE por um tab (se forem espaços, o make dá erro)

makefile avançado (1)

```
# Makefile empregue nas aulas práticas
# Bibliotecas a incluir
LIBS=#-pthread
# Flags para o compilador
CFLAGS=-Wall -W -g -Wmissing-prototypes #-ggdb
# nome do executavel
PROGRAM=initC
# Nome do ficheiro de opcoes do gengetopt
PROGRAM_OPT=#prog_opt
```

Continua >>

makefile avançado (2)

```
# Objectos necessarios para criar o executavel
PROGRAM_OBJS=initC.o debug.o ${PROGRAM_OPT}.o
.PHONY: clean all: ${PROGRAM}
# compilar com depuracao
depuracao: CFLAGS += -D SHOW_DEBUG
depuracao: ${PROGRAM}

${PROGRAM}: ${PROGRAM_OBJS}
   ${CC} -o $@ ${PROGRAM_OBJS} ${LIBS}
```

Continua >>

makefile avançado (3)

```
# Dependencias
initC.o: initC.c debug.h #${PROGRAM_OPT}.h
${PROGRAM_OPT}.o: ${PROGRAM_OPT}.c ${PROGRAM_OPT}.h

debug.o: debug.c debug.h
semaforos.o: semaforos.c semaforos.h

#como compilar .o a partir de .c .c.o: ${CC}
   ${CFLAGS} -c $<
# Como gerar os ficheiros do gengetopt
${PROGRAM_OPT}.h: ${PROGRAM_OPT}.ggo
   gengetopt < ${PROGRAM_OPT}.ggo --file-name=${PROGRAM_OPT}
```

Continua >>

makefile avançado (4)

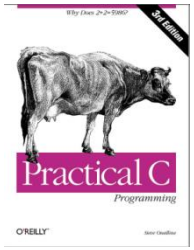
```
clean: rm -f *.o core.* *~ ${PROGRAM} *.bak
   ${PROGRAM_OPT}.h ${PROGRAM_OPT}.c

docs: Doxyfile
   doxygen Doxyfile
Doxyfile:
   doxygen -g Doxyfile

indent:
   dos2unix *.c *.h indent ${IFLAGS} *.c *.h
```

A desenvolver nas práticas

- Ferramentas auxiliares (referidas no makefile)
  - gengetopt
    - Utilitário que permite criar automaticamente o código fonte para tratamento dos parâmetros da linha de comando (argv e argc)
    - Ficheiro de entrada ".ggo"
  - doxygen
    - Documentação integrada no código (tipo **javadoc**)
  - indent
    - Ferramenta para a formatação do código fonte
    - Permite aplicar vários estilos
  - pmccabe
    - Métrica pmccabe associada à complexidade do código
  - cppcheck
    - Ferramenta para deteção de anomalias no código (potenciais erros)

Bibliografia (1)

- ✓ "Practical C Programming", Steve Oualline, O'Reilly, 3rd edition, 1998
  - Capítulo 10: pré-processador



- ✓ "Linguagem C", Luís Damas, FCA Editora

- ✓ Recursos online
  - The C language (online) – "The C Book"
    - http://publications.gbdirect.co.uk/c_book/
  - The C pre-processor (online)
    - http://gcc.gnu.org/onlinedocs/cpp/

Bibliografia (2)

- ✓ "GNU Gengetopt", http://www.gnu.org/s/gengetopt/gengetopt.html
- ✓ GNU indent, http://www.gnu.org/software/indent/
- ✓ GCC – GNU Compiler Collection, https://gcc.gnu.org/
- ✓ doxygen, http://www.doxygen.org