

COMPILANDO PROGRAMAS COM MAKEFILE

DCE792 - AEDs II (Prática)

Atualizado em: 7 de agosto de 2024

Iago Carvalho

Departamento de Ciência da Computação



O processo de compilação de um código é uma tarefa bem complexa e segue alguns passos predefinidos

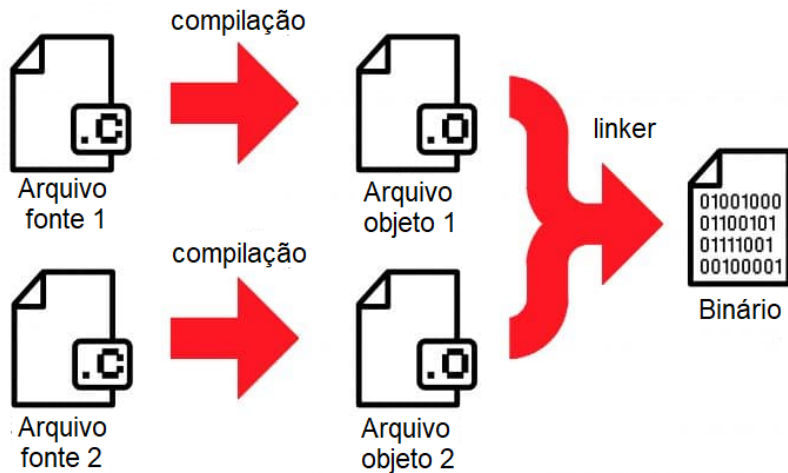
1. Os arquivos-fonte são transformados em arquivos objeto pelo compilador
2. Os arquivos objeto são ligados pelo *linker*
3. Um arquivo binário é criado

Muitas vezes, a IDE que você utiliza compila seu código com um único botão

Outras vezes, será necessário você iteragir com o compilador

- Escolher as diretivas de compilação
- Quais arquivos serão linkados e em que ordem
- Como o binário será criado
- ...

COMPILAÇÃO



MAKEFILE

Um **makefile** é um arquivo de configuração que explicita a forma como um binário deve ser gerado

- Ele contém instruções de compilação do seu código
- É utilizado para organizar o código desenvolvido e facilitar sua reprodução/divulgação
- Automatiza tarefas rotineiras como a limpeza dos arquivos temporários criados durante a compilação

Um makefile também pode acelerar o processo de compilação

- Ele é capaz de evitar a recompilação de arquivos de código que não foram alterados
- Por exemplo, se seu programa utiliza 120 bibliotecas e você altera apenas uma, o *make* descobre qual arquivo foi alterado e compila apenas a biblioteca necessária

EXEMPLO DE MAKEFILE

Vamos começar com um exemplo simples. Um pequeno código com 4 arquivos

- main.c
- helloWorld.c
- helloWorld.h
- makefile

O código está disponível em nosso Github [▶ Link](#)

EXEMPLO DE MAKEFILE

```
C main.c
dce792 > makefile > primeiro_exemplo > C main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "helloworld.h"
4
5 int main(){
6     helloworld();
7     return (0);
8 }

C helloworld.c
dce792 > makefile > primeiro_exemplo > C helloworld.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void helloworld(void){
5     printf("Hello World!\n");
6 }

C helloworld.h
dce792 > makefile > primeiro_exemplo > C helloworld.h
1 #ifndef _H_TESTE
2 #define _H_TESTE
3
4 void helloworld(void);
5
6 #endif


M makefile
dce792 > makefile > primeiro_exemplo > M makefile
1 # Makefile simples
2
3 all: hello
4
5 hello: main.o helloworld.o
6     gcc -o hello main.o helloworld.o
7
8 main.o: main.c helloworld.h
9     gcc -o main.o main.c -c -W -Wall -ansi -pedantic
10
11 helloworld.o: helloworld.c helloworld.h
12     gcc -o helloworld.o helloworld.c -c -W -Wall -ansi -pe
13
14 clean:
15     rm -rf *.o *~ hello
```

EXEMPLO DE MAKEFILE

```
M makefile
dce792 > makefile > primeiro_exemplo > M makefile


1  # Makefile simples
2
3  all: hello
4
5  hello: main.o helloWorld.o
6      gcc -o hello main.o helloWorld.o
7
8  main.o: main.c helloWorld.h
9      gcc -o main.o main.c -c -W -Wall -ansi -pedantic
10
11 helloWorld.o: helloWorld.c helloWorld.h
12     gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
13
14 clean:
15     rm -rf *.o *~ hello
```

EXEMPLO DE MAKEFILE - TARGETS


dce792 > makefile > primeiro_exemplo >  makefile

1 # Makefile simples

2

 all: hello


4

 hello: main.o helloWorld.o

6

gcc -o hello main.o helloWorld.o


7

 main.o: main.c helloWorld.h

9

gcc -o main.o main.c -c -W -Wall -ansi -pedantic


10

 helloWorld.o: helloWorld.c helloWorld.h

12

gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic

13

 clean:

15

rm -rf *.o *~ hello

EXEMPLO DE MAKEFILE - PRE-REQUISITOS

```
M makefile
dce792 > makefile > primeiro_exemplo > M makefile

1  # Makefile simples
2
3  all: hello
4
5  hello: main.o helloWorld.o
6      gcc -o hello main.o helloWorld.o
7
8  main.o: main.c helloWorld.h
9      gcc -o main.o main.c -c -W -Wall -ansi -pedantic
10
11 helloWorld.o: helloWorld.c helloWorld.h
12     gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
13
14 clean:
15     rm -rf *.o *~ hello
```

EXEMPLO DE MAKEFILE - RECEITA

```
M makefile
dce792 > makefile > primeiro_exemplo > M makefile
1  # Makefile simples
2
3  all: hello
4
5  hello: main.o helloWorld.o
6      gcc -o hello main.o helloWorld.o
7
8  main.o: main.c helloWorld.h
9      gcc -o main.o main.c -c -W -Wall -ansi -pedantic
10
11 helloWorld.o: helloWorld.c helloWorld.h
12     gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
13
14 clean:
15     rm -rf *.o *~ hello
```

EXEMPLO DE MAKEFILE - TABULAÇÃO

```
M makefile
dce792 > makefile > primeiro_exemplo > M makefile
1  # Makefile simples
2
3  all: hello
4
5  hello: main.o helloWorld.o
6  | gcc -o hello main.o helloWorld.o
7  | TAB
8  main.o: main.c helloWorld.h
9  | gcc -o main.o main.c -c -W -Wall -ansi -pedantic
10 | TAB
11 helloWorld.o: helloWorld.c helloWorld.h
12 | gcc -o helloWorld.o helloWorld.c -c -W -Wall -ansi -pedantic
13 | TAB
14 clean:
15 | rm -rf *.o *~ hello
```

FUNCIONAMENTO DO MAKEFILE

No terminal, ao digitar **make all**, o utilitário make vai executar o target **all** que se encontra na linha 3 do arquivo *makefile*.

- O seu pré-requisito é o arquivo binário *hello*, porém sem receita para **all**

Ele vai para a linha 5 para construir *hello*, e acha os pré-requisitos

1. *main.o*
2. *helloWorld.o*

Antes de partir para a linha 6, o *make* vai produzir os respectivos arquivos objeto

FUNCIONAMENTO DO MAKEFILE

Na linha 8, o *make* vê que o **pré-requisito** para *main.o* é a existência dos arquivos *main.c* e *helloWorld.h* no diretório atual.

- O *make* executa então a receita da linha 9
- Primeiro, ele chama o compilador *gcc*
- A parte *-o main.o* indica que a saída, ou seja, o arquivo produzido será *main.o*
- *-Wall* e *-pendantic* são configurações do compilador
 - Úteis para encontrar erros
 - Serão explicadas daqui a pouco

Terminando de produzir o arquivo objeto *main.o*, o *make* vai para as linhas 11 e 12 para produzir o arquivo objeto *helloWorld.o*

- Semelhante ao processo acima descrito

FUNCIONAMENTO DO MAKEFILE

Com os dois arquivos objeto construídos, o *make* pode então voltar para a linha 6 e cumprir a receita do binário *hello*

O *make* invoca agora o **linker** do *gcc*, e avisa utilizando *-o hello* que a saída será um binário no diretório atual chamado *hello*

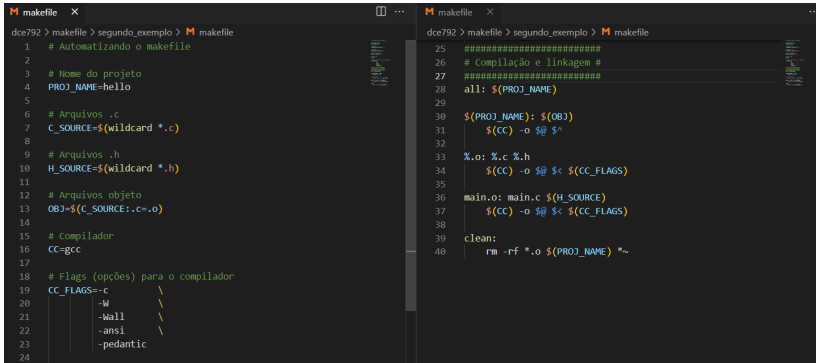
- Indica ao **linker** os dois arquivos objeto *main.o* e *helloWorld.o*

O comando **clean** da linha 14 somente será executado se você escrever no terminal *make clean*.

- Ele não possui pré-requisitos
- Sua receita exclui todos os arquivos com extensão *.o*, os de *backup* ~ e o arquivo binário *hello* no diretório atual

UM MAKEFILE UM POUCO MAIS
INTELIGENTE

SEGUNDO MAKEFILE



```
M makefile X
dce792 > makefile > segundo_exemplo > M makefile
1  # Automatizando o makefile
2
3  # Nome do projeto
4  PROJ_NAME=hello
5
6  # Arquivos .c
7  C_SOURCE=$(wildcard *.c)
8
9  # Arquivos .h
10 H_SOURCE=$(wildcard *.h)
11
12 # Arquivos objeto
13 OBJ=$(C_SOURCE:.c=.o)
14
15 # compilador
16 CC=gcc
17
18 # Flags (opções) para o compilador
19 CC_FLAGS=-c \
20         -W \
21         -Wall \
22         -ansi \
23         -pedantic
24
25 #####
26 # Compilação e linkagem #
27 #####
28 all: $(PROJ_NAME)
29
30 $(PROJ_NAME): $(OBJ)
31     $(CC) -o $@ $^
32
33 %.o: %.c %.h
34     $(CC) -o $@ $< $(CC_FLAGS)
35
36 main.o: main.c $(H_SOURCE)
37     $(CC) -o $@ $< $(CC_FLAGS)
38
39 clean:
40     rm -rf *.o $(PROJ_NAME) *
```

SEGUNDO MAKEFILE

Nas linha 3 à 23 estão contidas as variáveis para facilitar a customização do arquivo

- Algumas vezes as variáveis são chamadas de macros
- O uso das variáveis é igual a do shell script
 - Lembram-se de ICC?
- Escreve-se `$(VARIABEL)` ou `${VARIABEL}`

Nas linhas 7 e 10 utilizamos a função *wildcard* para obter o nome de todos os arquivos com extensões `.c` e `.h` no diretório em que o *makefile* se encontra.

Na variável `C_SOURCE` o nome de cada arquivo estará separado um do outro por um espaço.

Na linha 13 copiamos todos os nomes da variável `C_SOURCE` para `OBJ`, mas com a substituição da extensão `.c` para `.o`.

SEGUNDO MAKEFILE

Das linhas 19 a 22 utilizamos a barra inversa \ para quebrar uma linha em várias, para tornar a leitura mais fácil.

- Alternativamente, pode-se escrever todas as opções uma na frente da outra
- **ATENÇÃO!:** Não utilize *tabs* aqui, somente espaços
 - Caso contrário, o *make* vai confundir com o comando de receita e não funcionará corretamente

O resultado da linha 28 deste segundo exemplo é igual à linha 3 do primeiro exemplo

O restante das linhas é a automatização de nosso *makefile*

SEGUNDO MAKEFILE

```
33  %.o: %.c %.h
34  |      $(CC) -o $@ $< $(CC_FLAGS)
```

Na linha 33 interpreta-se que o *target* com extensão *.o* terá um pré-requisito com extensões *.c* e *.h* com mesmo nome

O símbolo % pega o *stem* (tronco) do nome, que é utilizado de referência no pré-requisito

- Por exemplo, o *stem* de *helloWorld.o* é *helloWorld*

A linha 34 é relativa às linhas 9 e 12 do primeiro exemplo

- A variável automática @ obtém o nome do target
 - Por exemplo, *helloWorld.o*
- O símbolo < pega o nome do primeiro pre-requisito
 - Por exemplo, *helloWorld.c*

SEGUNDO MAKEFILE

Perceba que as linhas 33 e 34 vão ser executadas para todos arquivos *.c* e *.h* que existirem

- Com exceção do *main.c*

As linhas 36 e 37 foram escrita para o caso especial do *main.o*

- Este não possui um arquivo *main.h*

A linha 31 funciona como a linha 6 do exemplo anterior

- A variável automática \wedge lista todos os pré-requisitos do target
- Invoca o *linker* para finalizar a compilação

ORGANIZANDO OS ARQUIVOS

TERCEIRO MAKEFILE

```
M makefile
dce792 > makefile > terceiro_exemplo > M makefile
1  # Organizando os arquivos no makefile
2
3  # Nome do projeto
4  PROJ_NAME=hello
5
6  # Arquivos .c
7  C_SOURCE=$(wildcard ./source/*.c)
8
9  # Arquivos .h
10 H_SOURCE=$(wildcard ./source/*.h)
11
12 # Arquivos objeto
13 OBJ=$(subst .c,.o,$(subst source,objects,$(C_SOURCE)))
14
15 # compilador
16 CC=gcc
17
18 # Flags (opções) para o compilador
19 CC_FLAGS=-c      \
20               -W    \
21               -Wall \
22               -ansi \
23               -pedantic
24
25 # Comando utilizado como target do clean
26 RM = rm -rf
27
28 #####
29 # Compilação e linkagem #
30 #####

M makefile
dce792 > makefile > terceiro_exemplo > M makefile
28 #####
29 # Compilação e linkagem #
30 #####
31 all: objFolder $(PROJ_NAME)
32
33 $(PROJ_NAME): $(OBJ)
34     @ echo 'Construindo o binário usando o linker GCC: $@'
35     $(CC) $^ -o $@
36     @ echo 'Binário pronto!: $@'
37     @ echo ' '
38
39 ./objects/%.o: ./source/%.c ./source/%.h
40     @ echo 'Construindo target usando o compilador GCC: $<'
41     $(CC) $< $(CC_FLAGS) -o $@
42     @ echo ' '
43
44 ./objects/main.o: ./source/main.c $(H_SOURCE)
45     @ echo 'Construindo target usando o compilador GCC: $<'
46     $(CC) $< $(CC_FLAGS) -o $@
47     @ echo ' '
48
49 objFolder:
50     @ mkdir -p objects
51
52 clean:
53     @ $(RM) ./objects/*.o $(PROJ_NAME) *~
54     @ rmdir objects
55
56 .PHONY: all clean
```

TERCEIRO MAKEFILE

Aqui nós temos que ter uma estrutura especial de diretórios, como abaixo

- O arquivo *makefile* fica no diretório raiz
- Arquivos *.c* e *.h* ficam no diretório *source*

```
✓ terceiro_exemplo
  ✓ source
    C helloWorld.c
    C helloWorld.h
    C main.c
    M makefile
```


Normalmente a ferramenta *make* imprime na tela do terminal cada linha da receita a ser executada

Para não poluir visualmente o terminal, colocamos no começo da linha o caractere `@` para suprimir essas impressões

Além disso, utilizamos o comando *echo* para deixar mensagens do que está sendo feito e colocar espaço entre uma compilação e outra na tela do terminal

TERCEIRO MAKEFILE

```
iagoac@DESKTOP-BFIS0ST:~/github/dce792/makefile/terceiro_exemplo$ make
Construindo target utilizando o compilador GCC: source/helloWorld.c
gcc source/helloWorld.c -c -W -Wall -ansi -pedantic -o objects/helloWorld.o

Construindo target utilizando o compilador GCC: source/main.c
gcc source/main.c -c -W -Wall -ansi -pedantic -o objects/main.o

Construindo o binário utilizando o linker GCC: hello
gcc objects/helloWorld.o objects/main.o -o hello
Binário pronto!: hello
```

Na linha 56 do arquivo *makefile* vemos o target `.PHONY` (alvo falso).

Nela colocamos como pré-requisito os targets que não possuem arquivos de mesmo nome.

Uma das razões para usar *phony target* é para não dar conflito com arquivos que sejam criados com mesmo nome

- Por exemplo, se existisse um arquivo chamado *all.c* ou *clean.c*

TERCEIRO MAKEFILE

No fim, temos uma estrutura de diretórios limpa e simples

- O arquivo executável fica no diretório raiz
- Arquivos *.c* e *.h* ficam no diretório *source*
- Arquivos objeto são localizados no diretório *objects*

```
▼ terceiro_exemplo
  ▼ objects
    ≡ helloWorld.o
    ≡ main.o
  ▼ source
    C helloWorld.c
    C helloWorld.h
    C main.c
```

OPÇÕES DO COMPILADOR

Opção	Descrição
gcc -ansi	Escolhe o C versão 90
gcc -c	Compila arquivos objeto sem linkagem
gcc -g<level>	Informações de debug para o GDB
gcc -o<level>	Otimizador automático de código
gcc -pedantic	Mostra os erros da ISO C
gcc -w	Desabilita todos os warnings
gcc -Wall	Liga todos os warnings padrão
gcc -Wextra	Liga warnings adicionais

- -g0: Sem informações de *debug*
- -g1: Poucas informações de *debug*
- -g: Padrão
- -g3: Todas as informações de *debug* disponíveis

- -o, -o1: Um pouco de otimização
- -o0: Opção padrão
- -o2: Otimiza um pouco mais que -o1
- -o3: Otimiza ainda mais que -o2
- -os: Otimiza para tamanho de código. Equivalente a -o2 sem as opções que podem aumentar o tamanho do código
- -ofast: Otimiza mais ainda que -o3. Pode fazer com que o programa tenha erros
- -og: Otimiza a experiência de *debug*
- -oz: Otimiza agressivamente para tamanho ao invés de velocidade

PRÓXIMA AULA:

VERIFICANDO ERROS COM
VALGRIND E GDB