

# Módulo 12 - Compilação de um projeto

## Processo de compilação do gcc

Um programa em C (e em qualquer outra linguagem de programação) raramente é escrito em somente uma folha. Para organizar isto, vamos trabalhar com o exercício 31

Divida o exercício 31 em um arquivo main, outro header e outro com as funções

```
// Chame o executável pelo terminal passando também a operação e os valores
// soma 3 4
// multiplica 4 5 5
// divide 10 2
// subtrai 2 1
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int soma(int argv, char *argc[]);
int multiplica(int argv, char *argc[]);
int subtrai(int argv, char *argc[]);
int divide(int argv, char *argc[]);

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

    if(argv < 3){
        printf("Argumentos insuficientes");
        return 0;
    }

    if(strcmp(argc[1], "soma") == 0){
        soma(argv, argc);
    }
    else if(strcmp(argc[1], "multiplica") == 0){
        multiplica(argv, argc);
    }
    else if(strcmp(argc[1], "subtrai") == 0){
        subtrai(argv, argc);
    }
    else if(strcmp(argc[1], "divide") == 0){
        divide(argv, argc);
    } else {
        printf("Comando nao existe");
    }

    return 0;
}
```

```

int soma(int argv, char *argc[]){
    int result = 0;
    for(int i = 2; i < argv; i++){
        result += atoi(argc[i]);
    }
    printf("%d", result);
}

int multiplica(int argv, char *argc[]){
    int result = 1;
    for(int i = 2; i < argv; i++){
        result *= atoi(argc[i]);
    }
    printf("%d", result);
}

// subtrai 5 4
int subtrai(int argv, char *argc[]){
    int result = atoi(argc[2]);

    for(int i = 3; i < argv; i++){
        result -= atoi(argc[i]);
    }
    printf("%d", result);
}

int divide(int argv, char *argc[]){
    float result = (float) atoi(argc[2]);

    for(int i = 3; i < argv; i++){
        result /= (float) atoi(argc[i]);
    }
    printf("%.2f", result);
}

```

Perceba que se você tentar compilar utilizando gcc com o main ocorrerá um erro. Isso é óbvio, pois main só faz referência ao arquivo header que contém somente os protótipos das funções. Para que isso funcione, precisamos gerar o arquivo object de cada uma das folhas através do comando:

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

O arquivo .o gerado é apenas seu código fonte traduzido para um código de máquina e depois para um binário. Execute no `Format-Hex main.o` para acessar o conteúdo deste arquivo objeto.

A criação deste arquivo é a penúltima etapa do processo de compilação, mas existem mais, como a expansão de seu código

```
gcc -E filename.c > filename.i
```

ou tradução de seu código fonte para assembly.

```
gcc -S filename.c
```

Veja, por exemplo, o código C seguinte e sua tradução para assembly

```
#include <stdio.h>

int main(){
    printf("Olaaaaa");
    return 0;
}
```

## Assembly

```
.file "hello.c"
.def __main; .scl 2; .type 32; .endef
.section .rdata,"dr"
LC0:
.ascii "Olaaaaa\0"
.text
.globl _main
.def _main; .scl 2; .type 32; .endef
_main:
LFB10:
.cfi_startproc
pushl %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl %esp, %ebp
.cfi_def_cfa_register 5
andl $-16, %esp
subl $16, %esp
call __main
movl $LC0, (%esp)
call _printf
movl $0, %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
LFE10:
.ident "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
.def _printf; .scl 2; .type 32; .endef
```

Mas voltando para o arquivo objeto. Crie um arquivo para cada um de seus códigos fonte.

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

```
gcc -c utils.c
```

Repare que o arquivo header.o não é necessário na linkeditação. Isso porque ele já existe em utils.c e em main, ele serve apenas para referenciar as duas páginas. Outra coisa é o arquivo object do header é muitas vezes maior que os outros dois arquivos, isso é porque ele funciona como uma interface para seu código. Utilize o comando a seguir para comparar os tamanhos do main com o do header

```
du -ha header.o
```

```
du -ha main.o
```

O header é um

Depois de gerado os arquivos objects, você pode linka-los. Assim o arquivo executável estará pronto.

```
gcc main.o utils.o -o calculadora
```

## Makefile

O make já é para estar pré configurado no terminal, se não, peça para o aluno mandar o erro.

Estrutura básica do makefile

```
all:
    echo "Meu primeiro aviso com make"
```

Primeiramente o comando é escrito e depois executado. Para resolver isso, coloca o @ antes do echo, o que cria uma execução silenciosa.

O comando make executado no terminal procura por um arquivo makefile e executa ele

O makefile trabalha com dependências, ou seja, ele depende de outros parâmetros para ser executado. No nosso caso, esses parâmetros serão os arquivos necessários para a criação do executável. Como vimos, esses são os arquivos objects (.o), que, por sua vez, precisam dos arquivos fonte (.c) para serem gerados. Antes de passar isso para o makefile, observamos a forma geral:

```
alvo: dependencia
    comandos
    comandos
```

- alvo: rotina a ser executada

- dependencia: condição de existência de algum arquivo, algo que executa outro alvo
- comandos: comandos do terminal a serem executados, como `gcc main.o utils.o` por exemplo

O alvo padrão para o makefile é chamado de all. Criando um makefile para mais de um alvo

```
all: mensagem1
    @echo "mensagem 2"

mensagem1:
    @echo "mensagem 1"
```

Acima, a dependência (ou pré-requisito) para executar all é executar mensagem1. A rotina all é chamada primeiro, mas para que ela seja executada, é necessário que mensagem1 seja executada antes.

Rode `make -n` para ver a rotina de execução do programa acima.

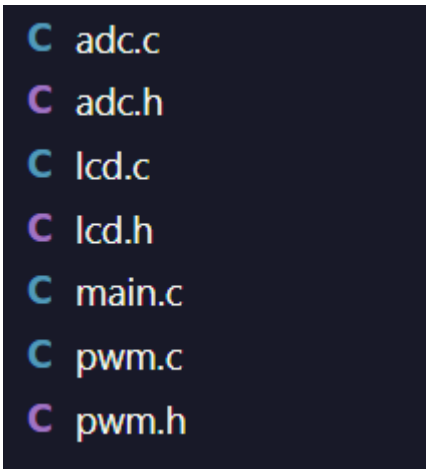
Também é possível rodar `make mensagem1` para executar algo diretamente.

Veja abaixo a execução de um programa que executa alguma ação

```
all: diretorio
    @echo "Pasta criada"

diretorio:
    @echo "criando nova pasta"
    mkdir pasta
```

Agora criando um make file para nosso programa com a seguinte arvore:



```
C adc.c
C adc.h
C lcd.c
C lcd.h
C main.c
C pwm.c
C pwm.h
```

```

all: lcd.o pwm.o adc.o main.o
    gcc lcd.o pwm.o adc.o main.o -o main.exe
    @echo "Arquivo main.exe criado"

main.o:
    gcc -c main.c -o main.o

lcd.o:
    gcc -c lcd.c -o lcd.o

pwm.o:
    gcc -c pwm.c -o pwm.o

adc.o:
    gcc -c adc.c -o adc.o

```

É importante ressaltar que depois de gerado os objects, se você executar o make novamente, não serão criados novos arquivos, ou seja, qualquer alteração que você realiza no código e executa novamente não surtirá efeito. Então você terá que apagar eles um a um. Por isso é bom criar uma rotina para apagar estes arquivos:

```

clean:
    rm lcd.o pwm.o adc.o main.o

```

## Mais sobre makefile

Um Makefile é um arquivo de texto simples usado para especificar dependências e instruções de compilação para compilar e vincular arquivos de código-fonte. Os Makefiles são comumente usados no desenvolvimento de software para automatizar o processo de compilação. Abaixo está um exemplo básico de um Makefile:

```

# Makefile para um programa C simples

# Compilador e flags do compilador
CC = gcc
CFLAGS = -Wall

# Arquivos de origem e executável alvo
SRCS = main.c file1.c file2.c
OBJS = $(SRCS:.c=.o)
TARGET = meu_programa

# Alvo padrão
all: $(TARGET)

# Vincule os arquivos de objeto para criar o executável

```

```
$(TARGET): $(OBJS)
    $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Compile os arquivos de origem em arquivos de objeto
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

# Limpe os arquivos de objeto e o executável
clean:
    rm -f $(OBJS) $(TARGET)
```

Neste Makefile:

1. Definimos variáveis para o compilador ( `CC` ), flags do compilador ( `CFLAGS` ), arquivos de origem ( `SRCS` ), arquivos de objeto ( `OBJS` ) e o executável alvo ( `TARGET` ).
2. O alvo padrão é `all` , que depende de `$(TARGET)` .
3. A regra para criar o executável alvo especifica as dependências (arquivos de objeto) e o comando de compilação.
4. A regra para compilar os arquivos de origem em arquivos de objeto usa uma regra de padrão para corresponder a todos os arquivos `.c` e compilá-los em arquivos `.o` correspondentes.
5. O alvo `clean` é usado para remover todos os arquivos de objeto gerados e o executável.

Para usar este Makefile, você normalmente o salvaria como um arquivo chamado `Makefile` em seu diretório de projeto e, em seguida, executaria o comando `make` no terminal para compilar seu programa. Por exemplo:

```
$ make
```

Isso compilará seus arquivos de origem e criará o executável `meu_programa` . Você também pode usar `make clean` para remover os arquivos de objeto gerados e o executável:

```
$ make clean
```

Makefiles podem se tornar muito mais complexos, dependendo das necessidades específicas do seu projeto, mas este exemplo deve lhe dar uma compreensão básica de como criar e usar um Makefile para compilar programas em C.