

Módulo 12 - Compilação de um projeto multifolha

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.

No programa, temos um conjunto de funções (chamaremos de `utils.c`), a função principal do programa (`main`) e um cabeçalho com as definições de nossas funções (chamaremos de `calculadora.h`). Vamos separar isto em três folhas:

- `calculadora.h`

```
#ifndef _CALC_
#define _CALC_

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

#endif
```

- `utils.c`

```
#include "calculadora.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

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);
}
```

- `main.c`

```
#include <stdio.h>
#include <string.h>
#include "calculadora.h"

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;
}
```

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. Os protótipos não fazem nada sozinhos.

Para que isso funcione, precisamos gerar o arquivo object de cada uma das folhas através do comando:

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

```
gcc -c utils.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:

```
Format-Hex main.o
```

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

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

ou a tradução de seu código fonte para assembly, que é de fato a compilação:

```
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 os arquivos objeto. Repare que o arquivo calculadora.h não é necessário na linkeditação. Isso porque ele já existe em utils.c e em main, ele serve apenas para referenciar as duas páginas. Se não existisse essas referências, main.c não compilaria, pois as funções não estariam definidas para ela.

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

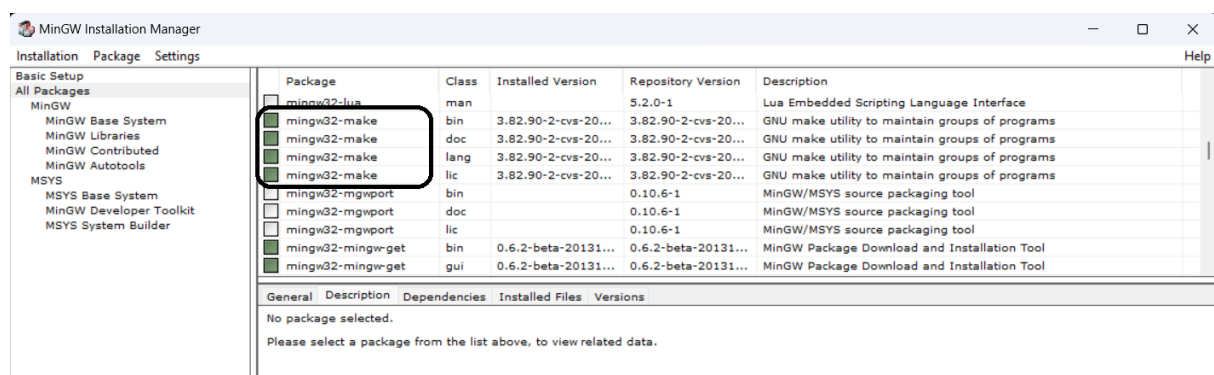
```
gcc main.o utils.o -o calculadora
```

Makefile


Instalação make

Caso o comando `make` não estiver funcionando em seu terminal, siga os passos a seguir:

No software minGW, certifique-se de ter instalado as opções de make:



No local onde foi instalado o minGW em sua máquina, dentro da pasta **bin**, deverá existir o seguinte arquivo:

 mingw32-make

01/09/2012 21:42

Aplicativo

215 KB

Se você veio utilizando o comando `gcc` até este momento, então basta rodar `mingw32-make` que funcionará. Utilize o comando `mingw32-make` no lugar de `make` para seguir as aulas.

Sobre o makefile

Um Makefile é um arquivo de configuração usado principalmente em sistemas Unix e sistemas operacionais baseados em Unix, como o Linux. Ele é usado para automatizar o processo de compilação e construção de projetos de software. Os Makefiles descrevem as dependências entre os diferentes componentes de um projeto e as regras para construir esses componentes.

A principal função de um Makefile é permitir que um sistema de compilação, geralmente uma ferramenta chamada "make," saiba como compilar, vincular e construir um programa a partir de um conjunto de arquivos-fonte e recursos. Isso é especialmente útil em projetos de desenvolvimento de software de médio a grande porte, onde há muitos arquivos e dependências.

Teste seu primeiro comando com o MakeFile:

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

Primeiramente o comando é escrito no terminal e depois executado. Para resolver isso, insira um `@` antes do `echo`, o que cria uma execução silenciosa.

Ao digitar `make` no terminal, o sistema operacional procura por um arquivo makefile e o executa.

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
```

- 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
```

Imagine um programa com todos estes arquivos para serem compilados:

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

Compilar um a um daria trabalho, não é mesmo? Por isso, criamos um makefile que automatiza esta tarefa para nós:

```
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
```