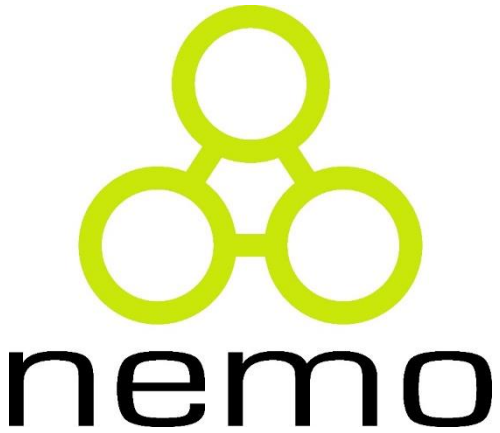


Programação II

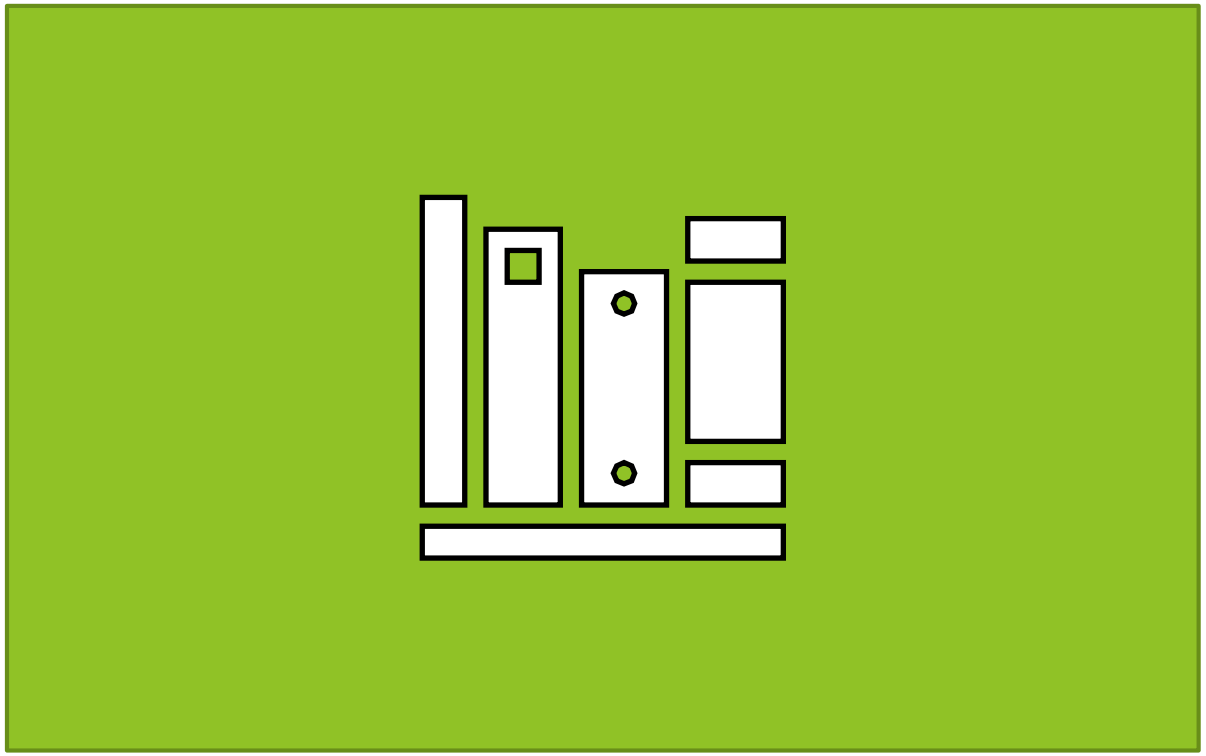
Jordana S. Salamon

jssalamon@inf.ufes.br

DEPARTAMENTO DE INFORMÁTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO



Bibliotecas

nemo

Introdução

- Todo programa em linguagem C é um conjunto de funções.
- A execução do programa começa sempre pela função main .
- Em um programa bem organizado, a função main tem caráter apenas gerencial: ela cuida da entrada de dados, da saída de resultados, e chama outras funções para fazer o resto do serviço .

Introdução

- Programas grandes e complexos em um único arquivo fonte pode dificultar sua compreensão e manutenção

- Os compiladores da linguagem C permitem a separação do código em arquivos diferentes, chamados **módulos.**

- Modularização:
  - facilita a manutenção e aumenta a legibilidade de um programa.

Introdução

- diretiva #include
- Exemplo:
- #include <stdio.h>

- main() { /* ... */ }

- As bibliotecas padrão do C encontram-se em um diretório específico que o compilador conhece. Ao especificar o nome do arquivo entre < e >, o compilador procura naquele diretório a biblioteca desejada e a inclui durante a compilação do programa.

nemo

Introdução

- É possível, porém, especificar uma inclusão a partir do diretório atual, usando “ " ao invés de < >.

```
// Arquivo prog.c
#include <stdio.h>
#include "minhabib.c"

main() {
        minhaFunc();
}
```

- O compilador trata a inclusão de minhabib.c em prog.c como se todo o código estivesse num arquivo só (no caso, prog.c). Para compilar, portanto, basta chamar o compilador como de costume:
- gcc prog.c -o prog

Introdução

- Algumas desvantagens:

- Qualquer modificação nos arquivos envolvidos requer uma recompilação de todo o código;

- Não há flexibilidade para substituir uma implementação de um conjunto de funções por outra diferente.

Como organizar um programa em C

- O texto de um programa (código) reside em um ou mais arquivos-fonte (*source files*), também conhecidos como módulos. O nome de cada módulo termina com extensão ".c".

- Cada módulo contém algumas das funções do programa. O módulo principal é o que contém a função main.

nemo

Como organizar um programa em C

- Para cada módulo, exceto o principal, é importante escrever um arquivo-cabeçalho (*header file*), contendo as constantes, as declarações dos tipos definidos pelo usuário, as variáveis globais (se houver) e os protótipos das funções a que o módulo principal pode ter acesso.

- O nome do arquivo-cabeçalho acompanha o nome do correspondente módulo.

Como organizar um programa em C

- Arquivo de declarações ou arquivo cabeçalho (header) – extensão .h:
  - declara as funções existentes naquela biblioteca, porém não provê sua implementação
- Arquivo de definições/implementações ou arquivo de código – extensão .c:
  - define a implementação de todas as funções declaradas no arquivo cabeçalho
- Caso a biblioteca de funções utilize tipos de dados definidos pelo usuário, estes usualmente são definidos no arquivo cabeçalho.

nemo

Exemplo

```
//Arquivo minhabib.h
void minhaFunc();
```

```
//Arquivo minhabib.c
#include <stdio.h>
#include "minhabib.h"

void minhaFunc() {
        printf("minhaFunc() foi chamada!\n");
}
```

```
// Arquivo prog.c
    #include "minhabib.h"

    int main() {
        minhaFunc();
        return 0;
    }
```

Compilação de um programa em módulos

- A compilação de um programa separado em módulos como no exemplo anterior, se dá em dois passos:

- gcc -c minhabib.c
- gcc -o prog prog.c minhabib.o

- A compilação com a opção -c diz ao compilador que ali não existe um programa completo (e, portanto, ele não procura pela função main()), mas apenas uma biblioteca de funções.
- O compilador gera um arquivo .o (chamado de arquivo objeto) com o mesmo nome da biblioteca

Compilação de um programa em módulos

- Outra forma de compilar seria:
  - gcc –c *.c
  - gcc –o prog *.o
  - Porém essa forma de compilar compila absolutamente todos os arquivos .c e linka todos os arquivos .o
  - Assim, caso uma das bibliotecas utilizadas seja modificada, todas as bibliotecas e o programa principal serial recompilados, o que não acontece na primeira opção.

O que acontece ao compilar esse código?

```
// Arquivo outrabib.h
typedef struct TMeuTipo {
int inteiro;
float real;
char string[100];
} MeuTipo;

void outraFunc(MeuTipo param);


// Arquivo outrabib.c
#include <stdio.h>
#include "outrabib.h"

void outraFunc(MeuTipo param) {
printf("outraFuncao() foi
          chamada!\n");
}
```

```
// Arquivo minhabib.h
#include "outrabib.h"
void minhaFunc(MeuTipo param);

// Arquivo minhabib.c
#include <stdio.h>
#include "minhabib.h"
#include "outrabib.h"

void minhaFunc(MeuTipo param) {
printf("minhaFunc() foi
          chamada!\n");
}

// Arquivo prog.c
#include "minhabib.h"
#include "outrabib.h"

main() {
MeuTipo var;
minhaFunc(var);
outraFunc(var);
}
```

nemo

O que acontece ao compilar esse código?

- o programa principal inclui o cabeçalho de outrabib.h duas vezes, uma diretamente e outra indiretamente
- ao tentar compilar o programa principal, o seguinte erro é apresentado:

error: redefinition of ‘struct TMeuTipo’
error: redefinition of typedef ‘MeuTipo’

Diretiva #ifndef

- Arquivos cabeçalho só precisam/devem ser incluídos uma vez em um código.
- Muitas vezes incluímos indiretamente um arquivo várias vezes, pois muitos cabeçalhos dependem de outros cabeçalhos.
- Para evitar problemas, costuma-se envolver o arquivo cabeçalho inteiro com um bloco condicional que só será compilado se o arquivo já não tiver incluído.

Diretiva #ifndef

- Para isso normalmente usa-se um símbolo baseado no nome do arquivo. Por exemplo, se o arquivo se chama "cabecalho.h", é comum usar um símbolo com o nome __CABECALHO_H

- Se o arquivo ainda não tiver sido incluído, ao chegar na primeira linha do arquivo, o compilador não encontrará o símbolo CABECALHO_H, e continuará a ler o arquivo, o que lhe fará definir o símbolo.

- Se tentarmos incluir novamente o arquivo, o compilador pulará todo o conteúdo pois o símbolo já foi definido.

O que acontece ao compilar esse código?

```
// Arquivo outrabib.h
#ifndef __OUTRABIB_H
#define __OUTRABIB_H

typedef struct TMeuTipo {
int inteiro;
float real;
char string[100];
} *MeuTipo;

void outraFunc(MeuTipo param);

#endif
```

nemo

Diretiva #ifndef

- O compilador inclui o arquivo cabeçalho no ponto onde ele foi chamado somente se a constante __OUTRABIB_H não estiver definida
- Se a constante não está definida, o cabeçalho a define sem valor algum
- Da próxima vez que o cabeçalho for incluído, #ifndef retornará falso e o código do cabeçalho não será adicionado no ponto de inclusão.

Makefile

```
all: prog

prog: prog.c minhabib.h outrabib.h minhabib.o
outrabib.o
   gcc -o prog prog.c minhabib.o outrabib.o

minhabib.o: minhabib.c minhabib.h outrabib.h
        gcc -c minhabib.c

outrabib.o: outrabib.c outrabib.h
         gcc -c outrabib.c

clean:
rm -rf *.o prog
```

Makefile

- Os termos antes de “:” são os alvos.
- Quando digitamos “make alvo” ele procura o alvo na lista e executa os comandos.
- Os termos após o “:” são as dependências.
- Ao tentar produzir um alvo, o comando make tenta primeiro satisfazer as dependências, que podem ser outros alvos .
- Abaixo da declaração dos alvos e dependências estão os comandos.
- É essencial que cada linha de comando comece com uma tabulação.

Makefile

- O primeiro alvo deve sempre ser all, pois é o alvo padrão. Ao digitar o comando make, ele começará do alvo all.
- O alvo clean também é muito comum, ele apaga arquivos que são gerados pelo processo de construção.
- Para os demais alvos, utilizamos os nomes dos arquivos produzidos a cada passo do processo de compilação.

Makefile

- Na lista de dependências devemos listar os arquivos que são incluídos com #include no código-fonte que está sendo compilado.
- Se um destes arquivos sofrer alteração em relação à última compilação, o make sabe que deve repetir os comandos daquele alvo.
- Também devemos incluir os arquivos usados no comando de compilação (códigos fonte e arquivos objeto).

That’s all Folks!