

Modularização

Programação de Computadores 1 - Ciência da Computação

Prof. Daniel Saad Nogueira Nunes



**INSTITUTO
FEDERAL**
Brasília

Campus
Taguatinga

Sumário

Introdução

Modularização

Compilação

Bibliotecas

Considerações Finais

Introdução

- ▶ Conforme sistemas vão ficando mais complexos e maiores, é interessante dividir este sistema em módulos, sendo cada módulo composto por diversas funções.
- ▶ Nesta divisão, cada módulo é responsável por uma determinada tarefa e pode conter diversos arquivos.
- ▶ Os módulos devem exportar uma espécie de “contrato” para os demais módulos, de modo que eles possam conversar entre si para prover a funcionalidade do sistema todo.

Introdução

- ▶ O “contrato” de um módulo define as funcionalidades providas por ele.
- ▶ Para que outro módulo utilize as funcionalidades deste primeiro módulo, ele não precisa saber como o contrato foi implementado, apenas a interface, isto é, como utilizar as funcionalidades providas.

Introdução

- ▶ Essa segmentação do sistema em módulos é conhecida como **modularização** e possui como uma das suas vantagens o aumento da capacidade de desenvolvimento.
- ▶ Equipes diferentes podem trabalhar em concomitância em módulos diferentes, desde que o contrato, a interface entre módulos, esteja bem definida.

Introdução

- ▶ Essa noção de utilizar um módulo como caixa preta para criar programas já é conhecida por vocês.
- ▶ Utilizamos desde o início a biblioteca `stdio.h`, que possibilita leitura e escrita formatada sem conhecer as implementações de fato das funções `scanf` e `printf`.
- ▶ Estudaremos agora como modularizar códigos em C.

Sumário

Introdução

Modularização

Compilação

Bibliotecas

Considerações Finais

Modularização

- ▶ Para iniciar o nosso estudo em modularização, partiremos de um exemplo simples.
- ▶ Uma aplicação que envolve vetores.
- ▶ Nesta aplicação deverá ser possível:
 - ▶ Leitura de vetores.
 - ▶ Escrita de vetores
 - ▶ Operações sobre vetores: soma, subtração e produto escalar.
- ▶ Cada um destes pontos será um módulo em nosso sistema.
- ▶ Para simplificar, vamos assumir que os vetores têm o tipo `int`.

Modularização

- ▶ Partindo desta divisão, teremos dois tipos de arquivos em cada módulo: os arquivos cabeçalho e os arquivos de implementação.
- ▶ Os arquivos cabeçalhos (.h) contém as assinaturas de funções e as definições importantes que devem ser exportados aos outros módulos. Eles compõem a interface dos módulos.
- ▶ Os arquivos de implementação (.c), como o nome sugere, implementam as funcionalidades dos módulos.

Sumário

Modularização

Arquivos cabeçalhos

Arquivos de implementação

Main

Arquivos Cabeçalhos

- ▶ Vamos aos arquivos cabeçalhos.
- ▶ Eles conterão as funcionalidades exportadas por cada módulo.
- ▶ Teremos os arquivos leitura.h, escrita.h e operacao.h.

Arquivos Cabeçalho: Leitura

```
1 #ifndef LEITURA_H
2 #define LEITURA_H
3
4 void le_vetor(int vetor[],int n);
5
6#endif
```

Arquivos Cabeçalho: Escrita

```
1 #ifndef ESCRITA_H
2 #define ESCRITA_H
3
4 void escreve_vetor(int vetor[], int n);
5
6#endif
```

Arquivos Cabeçalho: Operação

```
1 #ifndef OPERACAO_H
2 #define OPERACAO_H
3
4
5 void soma_vetores(int vetor_1[],int vetor_2[],int vetor_resultado[], int n);
6 void subtrai_vetores(int vetor_1[],int vetor_2[],int vetor_resultado[], int n);
7 int produto_escalar(int vetor_1[],int vetor_2[], int n);
8
9#endif
```

Arquivos Cabeçalhos

- ▶ Caso um módulo deseje utilizar as funcionalidades de outro módulo, basta incluir a interface desse para ter acesso às funcionalidades.
- ▶ `#include "nome_do_modulo.h"`
- ▶ Para evitar que as mesmas definições sejam incluídas múltiplas vezes por outros módulos, os arquivos cabeçalhos devem incluir uma guarda especial com a seguinte estrutura:

```
#ifndef NOME_DO_MODULO_H
#define NOME_DO_MODULO_H

/** definições e protótipos **/ 

#endif
```

Arquivos Cabeçalhos

- ▶ Esta guarda define uma macro da primeira vez que o arquivo incluído.
- ▶ Como a macro fica definida, a próxima vez que o arquivo for incluído, as definições não serão incluídas novamente, pois o `#ifndef NOME_DA_MACRO` irá avaliar como falso.

Arquivos Cabeçalhos

- ▶ Como visto, os arquivos cabeçalhos possuem as definições exportadas por cada módulo.
- ▶ É missão dos arquivos de implementação de cada módulo, implementar estas definições.

Sumário

Modularização

Arquivos cabeçalhos

Arquivos de implementação

Main

Arquivos de implementação

- ▶ Os arquivos de implementação serão responsáveis por implementar as funcionalidades descritas pela interface.
- ▶ Teremos os arquivos `leitura.c`, `escrita.c` e `operacao.c`.
- ▶ Eles devem incluir as definições dos arquivos cabeçalhos e implementá-las.

Arquivos Implementação: Leitura

```
1 #include <stdio.h>
2 #include "leitura.h"
3
4 void le_vetor(int vetor[],int n){
5     int i;
6     for(i=0;i<n;i++){
7         printf("vetor[%d] = ",i);
8         scanf("%d",&vetor[i]);
9     }
10 }
```

Arquivos Implementação: Escrita

```
1 #include <stdio.h>
2 #include "escrita.h"
3
4 void escreve_vetor(int vetor[], int n){
5     int i;
6     for(i=0;i<n;i++){
7         printf("V[%d] = %d\n",i,vetor[i]);
8     }
9 }
```

Arquivos Implementação: Operação

```
1 #include "operacao.h"
2
3 void soma_vetores(int vetor_1[], int vetor_2[], int vetor_resultado[], int n){
4     int i=0;
5     for(i=0;i<n;i++){
6         vetor_resultado[i] = vetor_1[i] + vetor_2[i];
7     }
8 }
9 void subtrai_vetores(int vetor_1[], int vetor_2[], int vetor_resultado[], int n){
10    int i=0;
11    for(i=0;i<n;i++){
12        vetor_resultado[i] = vetor_1[i] - vetor_2[i];
13    }
14 }
15 int produto_escalar(int vetor_1[], int vetor_2[], int n){
16     int i=0;
17     int soma=0;
18     for(i=0;i<n;i++){
19         soma += vetor_1[i] * vetor_2[i];
20     }
21     return soma;
22 }
```

Sumário

Modularização

Arquivos cabeçalhos

Arquivos de implementação

Main

Função Main

- ▶ Qualquer sistema em C precisa ter um arquivo com a função `main`, caso contrário, não será possível criar o arquivo executável.

Função Main

```
1 #include <stdio.h>
2 #include "leitura.h"
3 #include "escrita.h"
4 #include "operacao.h"
5
6 #define MAX 100
7
8 int main(void){
9     int vetor_1[MAX];
10    int vetor_2[MAX];
11    int vetor_soma[MAX];
12    int vetor_subtracao[MAX];
13    int prod_escalar;
14    int n;
15    printf("Digite o tamanho dos vetores (<=100): ");
16    scanf("%d",&n);
17
18    printf("Leitura do vetor 1.\n");
19    le_vetor(vetor_1,n);
20    printf("Leitura do vetor 2.\n");
21    le_vetor(vetor_2,n);
```

Função Main

```
22
23     /* Calcula soma de vetores */
24     soma_vetores(vetor_1,vetor_2,vetor_soma,n);
25
26     /* Calcula subtração de vetores*/
27     subtrai_vetores(vetor_1,vetor_2,vetor_subtracao,n);
28
29     /* Calcula o produto escalar*/
30     prod_escalar = produto_escalar(vetor_1,vetor_2,n);
31
32     printf("Imprimindo resultado da soma dos vetores.\n");
33     escreve_vetor(vetor_soma,n);
34
35     printf("Imprimindo resultado da subtração dos vetores.\n");
36     escreve_vetor(vetor_subtracao,n);
37
38     printf("Produto escalar = %d.\n",prod_escalar);
39
40     return 0;
41
42 }
```

Sumário

Introdução

Modularização

Compilação

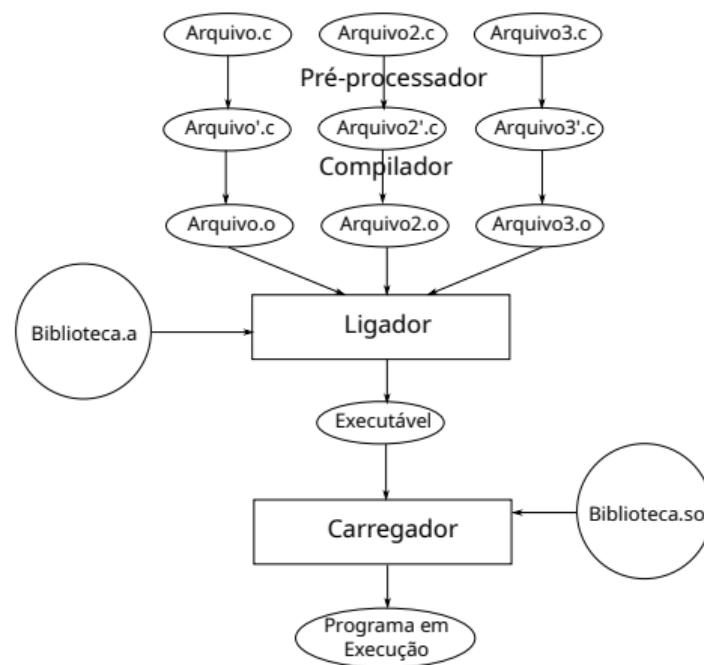
Bibliotecas

Considerações Finais

Compilação

- ▶ Para entender como combinar todos os arquivos, é necessário entender que existem diversas fases em C.
- ▶ Pré-processador: expande as macros e processa as diretivas de `#include`, `#define`, `#ifdef...`
- ▶ Compilador (*compiler*): traduz o código-fonte para código objeto.
- ▶ Ligador (*linker*): combina diversos arquivos objetos e bibliotecas estáticas para gerar um executável.
- ▶ Carregador (*loader*): carrega o programa para memória primária e inicializa os espaços de endereçamento bem como carrega as bibliotecas dinâmicas.

Compilação



Compilação

- ▶ Para compilar um arquivo .c em seu respectivo arquivo objetivo, basta realizar:
`gcc -c fonte.c`
- ▶ Com isso o arquivo `fonte.o` será criado.

Compilação

- ▶ Para unir todos os objetos e criar um executável segundo o nosso exemplo, seria necessário apenas os seguintes comandos:

```
gcc -c leitura.c escrita.c operacao.c main.c  
gcc leitura.o escrita.o operacao.o main.o -o executavel
```

Sumário

Introdução

Modularização

Compilação

Bibliotecas

Considerações Finais

Bibliotecas

- ▶ Bibliotecas são uma coleção de rotinas, funções e variáveis que fornecem alguma funcionalidade ao sistema.
- ▶ Elas facilitam a reutilização de código em outros programas.
- ▶ Podem ser:
 - ▶ Estáticas.
 - ▶ Dinâmicas.

Bibliotecas

- ▶ Bibliotecas estáticas: são ligadas juntamente com outros arquivos objetos para geração do executável.
 - ▶ No Linux: arquivos com extensão **.a**.
- ▶ Bibliotecas dinâmicas (shared): são carregadas em tempo de execução devido ao carregador (loader).
 - ▶ No Linux: arquivos com extensão **.so**.

Sumário

Bibliotecas

Bibliotecas Estáticas

Bibliotecas Dinâmicas

Reuso de Código

Bibliotecas Estáticas

- ▶ Para criar bibliotecas estáticas a partir dos arquivos objetos .o, basta usar o utilitário ar: ar -crs biblioteca.a escrita.o leitura.o operacao.o.
- ▶ Uma vez criada a biblioteca, podemos ligá-la com o nosso arquivo que contém a função main: gcc main.c biblioteca.a -o executavel

Sumário

Bibliotecas

Bibliotecas Estáticas

Bibliotecas Dinâmicas

Reuso de Código

Bibliotecas Dinâmicas

- ▶ Para gerar biblioteca dinâmica com nome libdinamica.so dos objetos leitura.o escrita.o e operacao.o:
gcc -shared -o libdinamica.so leitura.o escrita.o operacao.o
- ▶ Para ligar os objetos e formar o executável propriamente dito, precisamos informar o caminho de onde está a biblioteca e indicar o nome da mesma (abreviado):
gcc -L<caminho_da_biblioteca> -Wl,-rpath,<caminho_da_biblioteca> \
main.c -o main -ldinamica

Sumário

Bibliotecas

Bibliotecas Estáticas

Bibliotecas Dinâmicas

Reuso de Código

- ▶ As bibliotecas possuem uma utilidade muito grande, na prática.
- ▶ Você pode integrar bibliotecas de terceiros sem ter que implementar tudo do zero.
- ▶ Só é necessário conhecer a interface da biblioteca (os arquivos .h).

Sumário

Introdução

Modularização

Compilação

Bibliotecas

Considerações Finais

Considerações

- ▶ Verificamos a importância da modularização para aumentar a capacidade de desenvolvimento e legibilidade em projetos grandes.
- ▶ É essencial utilizá-la e documentá-la.
- ▶ É possível criar bibliotecas para uso posterior em outras aplicações através da modularização ou então utilizar bibliotecas já existentes em seus programas.
- ▶ Verificamos mais de perto como é o processo de criação de executáveis no sistema operacional.