

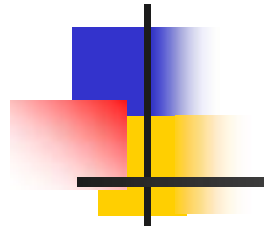
Universidade Estadual de Santa Cruz
Colegiado de Ciência da Computação



Linguagens de Programação II

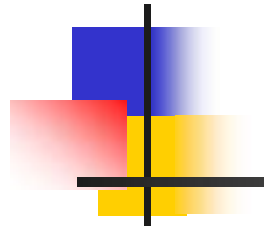
Modularização de Programas
Funções

Dany Sanchez Dominguez
dsdominguez@gmail.com
Sala 1 - NBCGIB



Introdução

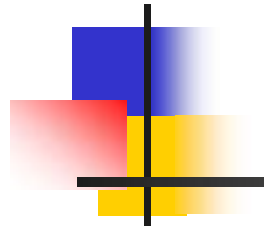
- A maioria dos programas de computador que resolvem problemas do mundo real são MUITO maiores que os programas vistos neste curso,
- O desenvolvimento e manutenção de grandes projetos é praticamente impossível se não usarmos o princípio de modularização.
- A melhor maneira de desenvolver e manter um programa é construí-lo a partir de pequenas partes ou módulos.



Introdução

- **Porque modularizar?**

- Em geral, problemas complexos exigem algoritmos complexos. Mas sempre é possível dividir um problema grande em problemas menores. Desta forma, cada parte menor tem um algoritmo mais simples, mais fácil de ser elaborado,
- Paradigma de dividir e conquistar,
- Esse trecho menor é chamado de módulo ou sub-rotina (ou função, ou procedimento).



Introdução

- **Sub-rotinas ou módulos:**

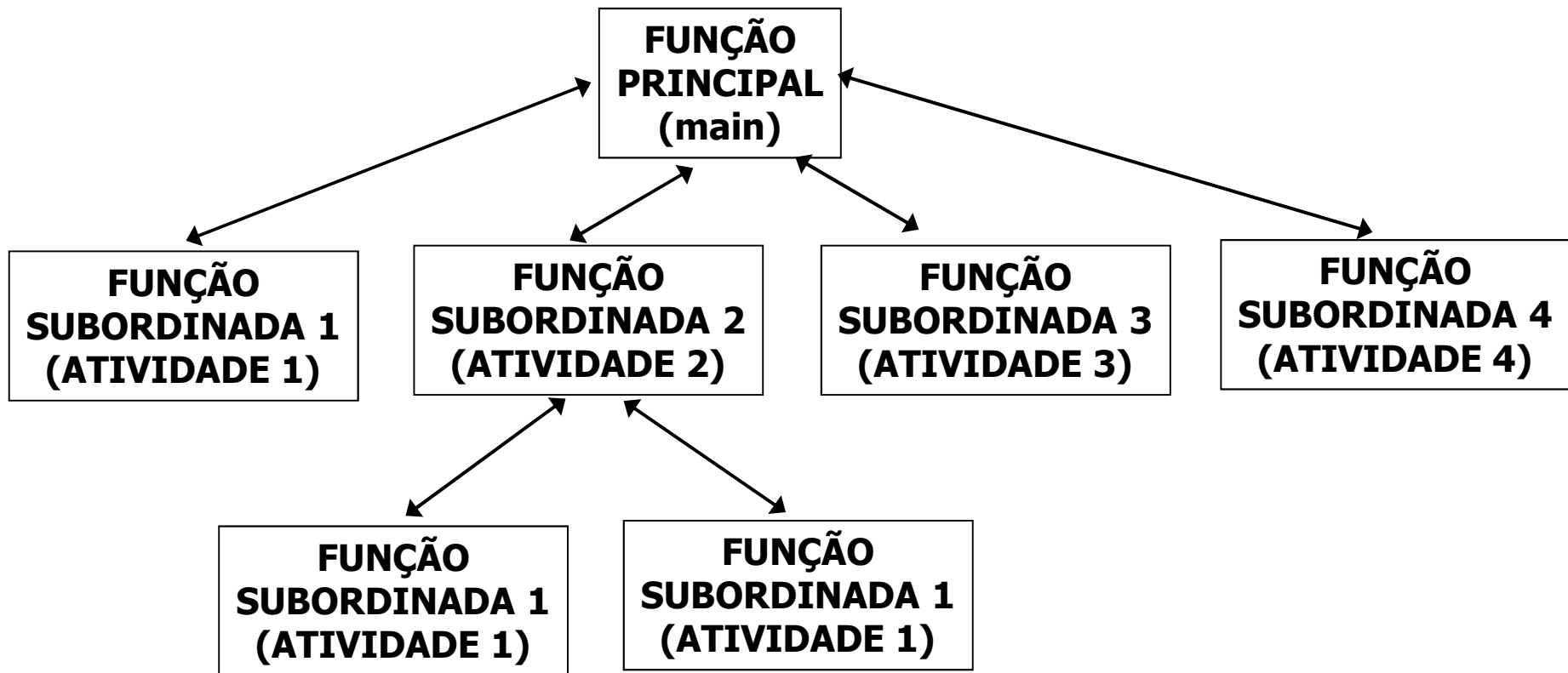
- Uma sub-rotina é na verdade um mini-programa e sendo um programa poderá efetuar diversas operações computacionais, funcionando no formato:

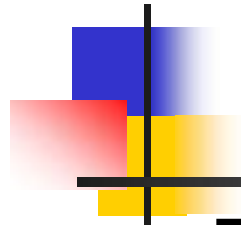


- Que precisamos conhecer para usar um módulo?
 - Nome do módulo
 - Dados de entrada
 - Saída

Introdução

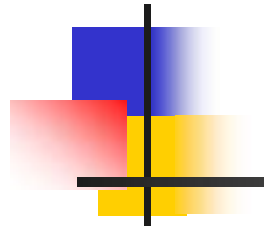
- Programas modularizados apresentam uma estrutura hierárquica





Introdução

- **Exemplo:** Cálculo do contracheque de um funcionário.
 - Quais são os módulos necessários:
 - Modulo 1 - Recebe o valor do salário e calcula o imposto de renda
 - Modulo 2 - Recebe o valor do salário e calcula o desconto para previdência
 - Modulo 3 - Recebe o valor do salário e o número de horas trabalhadas, calcula descontos por faltas.
 - etc...



Introdução

- **Exemplo:** Lista 2. Ex. 8, subtração e soma de matrizes

```

int main(){
    float **A, **B, **Som, **Sub;
    int i, j, m, n;

    //Leitura do numero de linhas e colunas
    printf("Digite o numero de linhas da matriz: ");
    scanf("%d", &m);
    printf("Digite o numero de colunas da matriz: ");
    scanf("%d", &n);

    //Alocação dinâmica das matrizes
    A = (float **) malloc (m * sizeof(float *));
    B = (float **) malloc (m * sizeof(float *));
    Som = (float **) malloc (m * sizeof(float *));
    Sub = (float **) malloc (m * sizeof(float *));
    if((A==NULL) || (B==NULL) || (Som==NULL) || (Sub==NULL)){
        printf("Erro de alocaco!!!\n");
        system("PAUSE");
        return -1;
    }
    for(i=0;i<m;i++){
        A[i] = (float *) malloc (n * sizeof(float));
        B[i] = (float *) malloc (n * sizeof(float));
        Som[i] = (float *) malloc (n * sizeof(float));
        Sub[i] = (float *) malloc (n * sizeof(float));
        if((A[i]==NULL) || (B[i]==NULL) || (Som[i]==NULL) || (Sub[i]==NULL)){
            printf("Erro de alocaco!!!\n");
            system("PAUSE");
            return -1;
        }
    }
}

```

Módulo 1 – aloca uma matriz de floats

**Recebe: numero de linhas
numero de colunas**

**Retorna: ponteiro a memória
alocada**


```
//Leitura de dados e Calculos
```

```
for(i=0;i<m;i++)
```

```
    for(j=0;j<n;j++){
```

```
        printf("Digite o elemento [%d][%d] de A: ", i+1, j+1);
```

```
        scanf("%f", &A[i][j]);
```

```
        printf("Digite o elemento [%d][%d] de B: ", i+1, j+1);
```

```
        scanf("%f", &B[i][j]);
```

```
        Som[i][j] = A[i][j] + B[i][j];
```

```
        Sub[i][j] = A[i][j] - B[i][j];
```

```
    }
```

```
//Resultados
```

```
printf("\nSoma\n");
```

```
for(i=0;i<m;i++){
```

```
    for(j=0;j<n;j++) printf("%4.2f ", A[i][j]);
```

```
    if (i == (m/2)) printf(" + "); else printf(" ");
```

```
    for(j=0;j<n;j++) printf("%4.2f ", B[i][j]);
```

```
    if (i == (m/2)) printf(" = "); else printf(" ");
```

```
    for(j=0;j<n;j++) printf("%4.2f ", Som[i][j]);
```

```
    printf("\n");
```

```
}
```

```
printf("\nSubtracao\n");
```

```
for(i=0;i<m;i++){
```

```
    for(j=0;j<n;j++) printf("%4.2f ", A[i][j]);
```

```
    if (i == (m/2)) printf(" - "); else printf(" ");
```

```
    for(j=0;j<n;j++) printf("%4.2f ", B[i][j]);
```

```
    if (i == (m/2)) printf(" = "); else printf(" ");
```

```
    for(j=0;j<n;j++) printf("%4.2f ", Sub[i][j]);
```

```
    printf("\n");
```

```
}
```

Módulo 2 – imprime uma operação de matrizes

Recebe: Mensagem, três matrizes, operador

Retorna: nada (void)

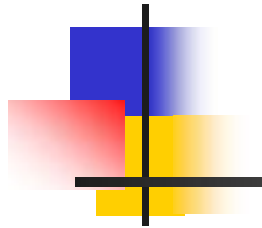
```
//liberando memória
for(i=0;i<m;i++){
    free(A[i]);
    free(B[i]);
    free(Som[i]);
    free(Sub[i]);
}
free(A);
free(B);
free(Som);
free(Sub);

system("PAUSE");
return 0;
}
```

Módulo 3 – libera memória de uma matriz

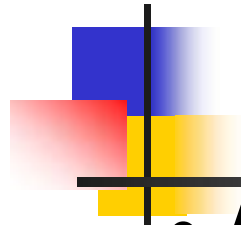
Recebe: numero de linhas, ponteiro

Retorna: nada (void)



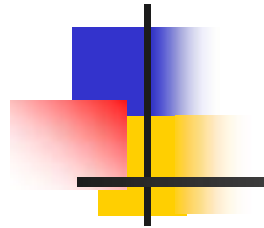
Introdução

- O conceito de modularização:
 - permite uma melhor reutilização do código,
 - implementa o conceito de abstração de processos (caixa preta),
 - evita a repetição de código,
 - permite utilizar códigos desenvolvidos por outros programadores,
 - implementa o principio de acesso mínimo.



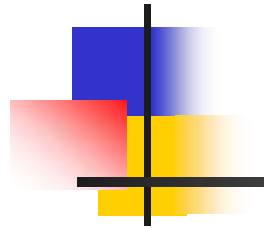
Módulos em C

- A modularização é implementada em C através de funções,
- Existem dois tipos de funções em C:
 1. as funções da biblioteca padrão:
 - Entrada/Saída (I/O) - `<stdio.h>`,
 - Matemáticas - `<math.h>`,
 - Manipulação de caracteres - `<ctype.h>`,
 - Manipulação de strings - `<string.h>`,
 - Muitas outras.
 2. funções definidas pelo programador.



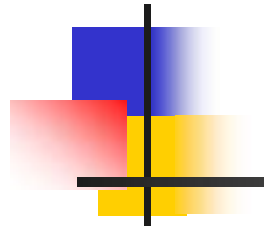
Módulos em C

- Todas as instruções em C devem estar incluídas em uma função,
- Todo programa deve conter uma função `main()` que será invocada pelo sistema operacional para começar a execução,
- A função `main()` poderá chamar outras funções da biblioteca padrão ou definidas pelo programador.



ESCOPO DE VARIÁVEIS

- Escopo = Área de atuação da variável
- As variáveis podem ter escopo local ou escopo global.
- Escopo local:
 - As variáveis são declaradas dentro de um modulo,
 - são criadas no inicio do módulo,
 - são destruídas no final do módulo,
 - apenas são acessíveis dentro do módulo
 - variáveis locais de um módulo não podem ser acessíveis por outro módulo.



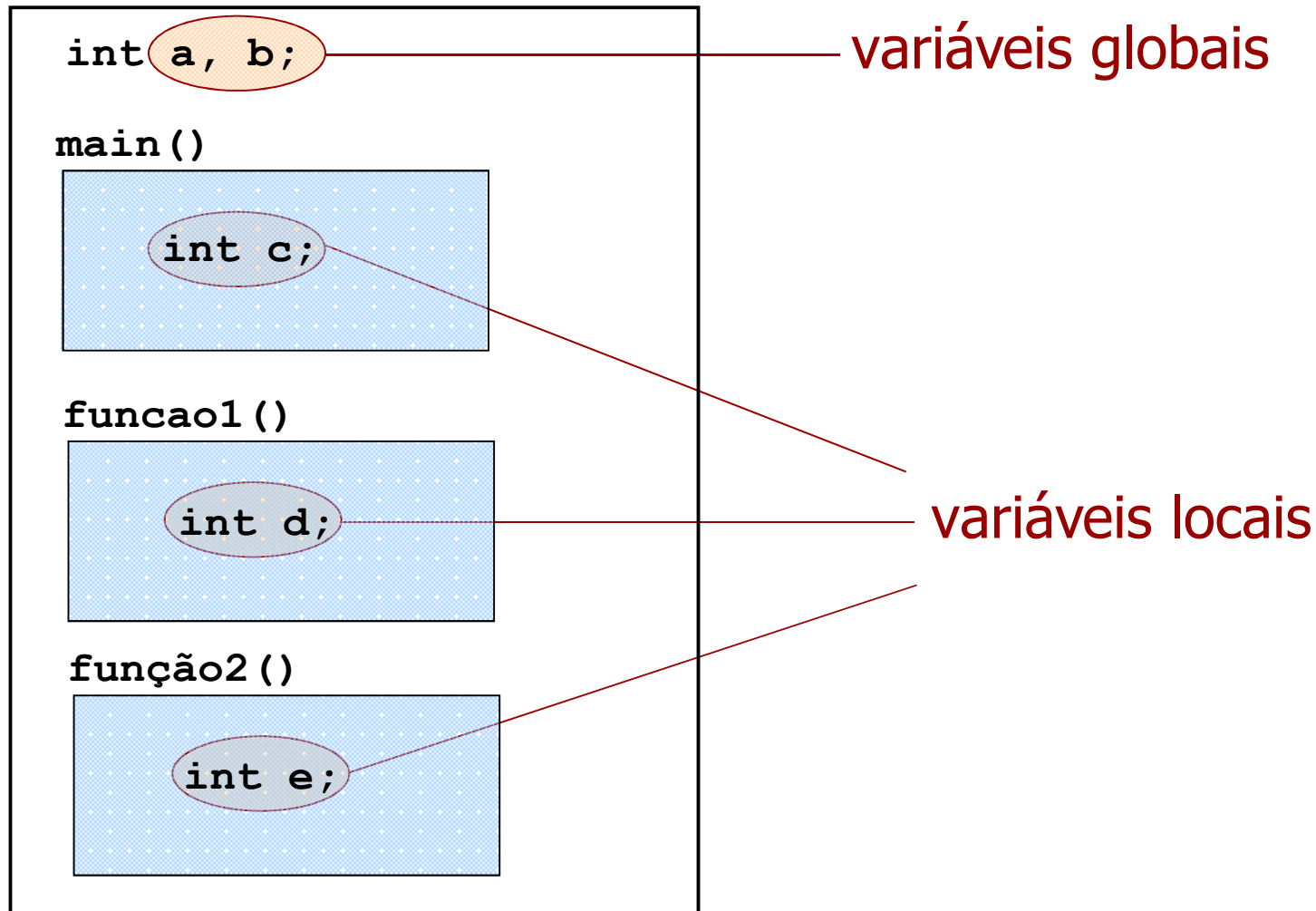
ESCOPO DE VARIÁVEIS

Escopo global:

- As variáveis são declaradas fora de qualquer modulo,
- são criadas no inicio do programa,
- são destruídas no final do programa,
- são acessíveis dentro de qualquer módulo,
- podem ser utilizadas (modificadas) em qualquer módulo,
- as variáveis globais violam o princípio de acesso mínimo,
- as praticas de engenharia de software recomendam não utilizar variáveis globais.



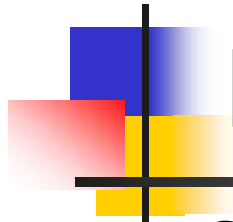
ESCOPO DE VARIÁVEIS





FUNÇÕES EM C

- Ao criar nossas próprias funções devemos considerar:
 1. o protótipo da função,
 2. a definição da função,
 3. os parâmetros da função,
 4. a chamada a função,
 5. o valor de retorno da função.



FUNÇÕES EM C

- Sintaxe do protótipo de função:

```
tipo_de_retorno nome_função(lista_parâmetros) ;
```

- O protótipo (ou cabeçalho) da função diz ao compilador:
 - o tipo de dado retornado pela função
 - o número de parâmetros que a função espera receber, os tipos dos parâmetros e a ordem na qual os parâmetros são esperados.
- O compilador utiliza os protótipos para validar as chamadas a funções.



FUNÇÕES EM C

- Ao incluirmos uma biblioteca de cabeçalho:

```
#include <biblioteca.h>
```

- Fornecemos ao compilador os protótipos de todas as funções da biblioteca,
- Permitindo verificar se a quantidade de parâmetros, e o tipo de cada um é correto,
- Não incluir a biblioteca de cabeçalho nos leva a erros de sintaxe.



FUNÇÕES EM C

- Definição de função:

```
tipo_de_retorno nome_função(lista_parâmetros) {  
    declarações  
    instruções  
    return value;  
}
```

- Funções não podem ser definidas dentro do corpo de outras funções.
- Os parâmetros de uma função são utilizados para enviar informação (dados) a função.



FUNÇÕES EM C

- Os parâmetros enviados a uma função são variáveis locais da função.
- A lista de parâmetros de uma função é formada por duplas de tipo e nome de variáveis separadas por virgulas:

```
tipo_retorno nome_função(tipo1 nome1,  
                          tipo2 nome2,  
                          ...,  
                          tipon nomen) ;
```



FUNÇÕES EM C

- Funções que não recebem nenhum parâmetro devem ter uma lista de parâmetros “vazia”:

```
tipo_retorno nome_função(void) ;
```

- Uma declaração do tipo:

```
tipo_retorno nome_função() ;
```

indica ao compilador de C para não verificar a lista de parâmetros desta função, esta pratica não é recomendavel.

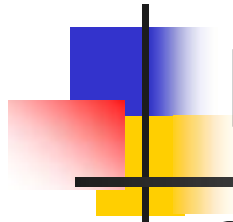


FUNÇÕES EM C

- Uma chamada a função tem a seguinte sintaxe:

`nome_função(lista_parâmetros)`

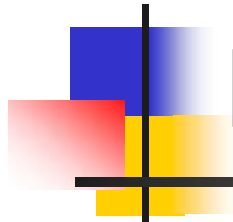
- A chamada a função é utilizada para invocar uma função,
- Uma chamada a função transfere o controle do programa a primeira instrução da função,
- A quantidade de parâmetros, o tipo dos parâmetros e a ordem na chamada devem coincidir com o protótipo da função.



FUNÇÕES EM C

- O controle de execução é retornado ao ponto de chamada (função chamadora) ao finalizar a execução da função chamada,
- A execução de uma função finaliza se o comando return é encontrado ou se o bloco da função é finalizado,
- As funções em C podem retornar um valor,
- O tipo do valor de retorno é especificado no protótipo da função:

tipo_de_retorno nome_função(lista_parâmetros) ;



FUNÇÕES EM C

- Uma função que não retorna nenhum valor deve ter um tipo de retorno "vazio":

```
void nome_função(lista_parâmetros) ;
```

- Em C por defeito todas as funções retornam um valor inteiro,
- Como será interpretada a declaração:

```
nome_função(lista_parâmetros) ;
```

- O valor retornado por uma função pode ser atribuído a uma variável, ou utilizado em qualquer expressão válida.



FUNÇÕES EM C - EXEMPLO

```
int quadrado(int);    /* protótipo da função */

int main()
{
    int i;

    for(i=1; i<=10; i++)
        printf("%d", quadrado(i));    /* chamada a função */
    printf("\n");

    return 0;
}

/* definição da função */
int quadrado(int num){
    return num*num
}
```



FUNÇÕES - EXEMPLOS

- **Exemplo:** *Escreva um programa que receba um número inteiro e imprima o seguinte padrão (n=4)*

```
*****
*****
*****
*****

*****
*   *
*   *
*   *
*****

*
**
***
****
```

- Crie uma função para imprimir o quadrado.*
- Crie uma função para imprimir o quadrado vazado.*
- Crie uma função para imprimir o triangulo.*
- Crie outras funções se forem necessárias.*

```

void prn_quadrado(int);
void prn_quadrado_vaz(int);
void prn_triangu(int);
void prn_linha(int);
void prn_linha_vaz(int);

int main() {
    int n;
    printf("Digite n:");
    scanf("%d", &n);

    prn_quadrado(n);
    prn_quadrado_vaz(n);
    prn_triangu(n);

    system("PAUSE");
    return 0;
}

void prn_linha(int n) {
    int j;
    for(j=0;j<n;j++)
        printf("*");
    printf("\n");
}

```

```

void prn_linha_vaz(int n) {
    int j;
    printf("*");
    for(j=1;j<n-1;j++)
        printf(" ");
    printf("*\n");
}

void prn_quadrado(int n) {
    int i;
    for(i=0;i<n;i++)
        prn_linha(n);
    printf("\n");
}

void prn_quadrado_vaz(int n) {
    int i;
    prn_linha(n);
    for(i=1;i<n-1;i++)
        prn_linha_vaz(n);
    prn_linha(n);
    printf("\n");
}

void prn_triangu(int n) {
    int i;
    for(i=0;i<n;i++)
        prn_linha(i+1);
    printf("\n");
}

```



PASSAGEM DE PARÂMETROS

- Existem duas formas de enviar parâmetros para uma função:
 1. chamada por valor,
 2. chamada por referência.
- Chamada por valor:
 - é feita uma **cópia** do valor da variável original para a variável que representa o parâmetro,
 - a cópia (parâmetro) esta disponível na função chamada,



PASSAGEM DE PARÂMETROS

- Chamada por valor ...
 - as modificações na cópia (parâmetro) não afetam o valor original da variável na função que realizou a chamada,
 - a passagem por valor deve ser usada sempre que a função chamada não precisa modificar o valor da variável original,
 - Em C, todas as chamadas são por valor.
 - Evita efeitos “colaterais”, como modificar acidentalmente o valor de uma variável.



PASSAGEM DE PARÂMETROS

- Chamada por referencia:
 - é enviado a função uma **referência** (endereço) a variável original,
 - é possível modificar a variável original na função chamada,
 - Adequado quando o “tamanho” do parâmetro é grande, evitando a sobrecarga da chamada por valor,
 - ou quando é necessário modificar o valor de um parâmetro dentro da função.



PASSAGEM DE PARÂMETROS

- Chamada por referencia:
 - Em C, o uso de ponteiros nos permite simular chamadas por referência.
 - Ao chamar uma função com argumentos que devem ser modificados, são passados os endereços (&) dos argumentos.
- Ilustramos as diferencias entre chamada por valor e chamada por referência com um exemplo.


```

/* Eleva uma variável ao cubo usando chamada por valor */
#include <stdio.h>
#include <stdlib.h>

int cubPorValor(int);

main() {
→ int num = 5;

→ printf("Valor original %d\n", num);
→ num = cubPorValor(num);
→ printf("Novo valor %d\n", num);
→ system("PAUSE");
→ return 0;
}

int cubPorValor(int n) {
→ return n * n * n;
}

```

Valor original 5
Novo valor 125
Pressione qualquer tecla para continuar. . .

Endereço de memória	Células de memória
------------------------	-----------------------

1024

1056

1088

1120

1152

⋮

⋮

```

/* Eleva uma variável ao cubo usando chamada por referencia */
#include <stdio.h>
#include <stdlib.h>

void cubPorReferencia(int *);

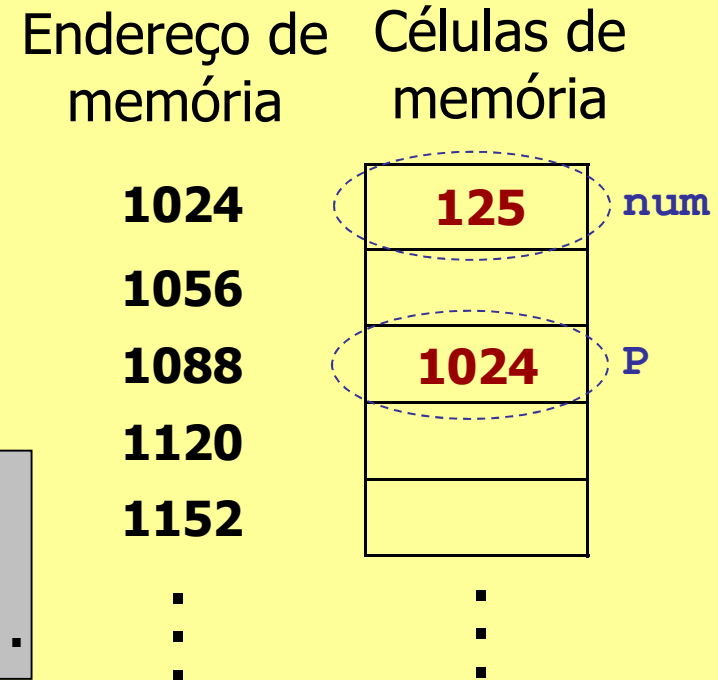
main() {
→ int num = 5;

→ printf("Valor original %d\n", num);
→ cubPorReferencia(&num);
→ printf("Novo valor %d\n", num);
→ system("PAUSE");
→ return 0;
}

void cubPorReferencia(int *P) {
→ *P = *P * *P * *P;
}

```

Valor original 5
Novo valor 125
Pressione qualquer tecla para continuar...





PASSAGEM DE PARÂMETROS

- O qualificador `const` permite ao programador informar ao compilador que uma determinada variável não pode ser modificada.

```
int const a = 1;
```

- O qualificador `const` é usado freqüentemente em passagem de parâmetros para indicar ao compilador que o parâmetro não deve ser modificado no corpo da função.
- Exemplo: Imprimindo um vetor.

```
/* Qualificador const, funcao para imprimir um vetor */
#include <stdio.h>
#include <stdlib.h>

void prn_vet(int *, const int);

main() {
    int vet[5] = {1,2,3,4,5}, n;

    n=5;
    prn_vet(vet, n);

    system("PAUSE");
    return 0;
}

void prn_vet(int *v, const int m){
    int i;

    for(i=0;i<m;i++)
        printf("%d\t", v[i]);
    printf("\n");
}
```



PASSAGEM DE PARÂMETROS

- O qualificador `const` é combinado com chamadas por referências para conseguirmos a eficiência de uma chamada por referência com a segurança de uma chamada por valor.



PONTEIROS COMO PARÂMETROS

- Há quatro maneiras de passar um ponteiro para uma função:
 1. um ponteiro não-constante para um dado não-constante
 2. um ponteiro constante para um dado não constante
 3. um ponteiro não-constante para um dado constante
 4. um ponteiro constante para um dado constante
- cada uma das quatro combinações fornece um nível de acesso diferente.



PONTEIROS COMO PARÂMETROS

- Ponteiro não-constante para um dado não constante:
 - é o maior nível de acesso,
 - o dado pode ser modificado desreferenciando o ponteiro,
 - o ponteiro pode ser modificado para apontar para outro endereço,
 - esta declaração não inclui `const.`
 - Ilustramos com um exemplo.

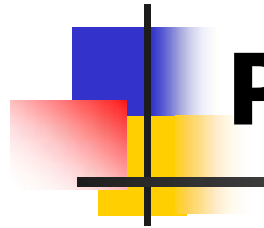
```
/* Converte letras minusculas para maiusculas */
/* usando um ponteiro não-constante para um dado não-constante*/
#include <stdio.h>
#include <stdlib.h>

void converteParaMaiusculas(char *);

main() {
    char string[] = "caracteres";

    printf("A string antes da conversao: %s\n", string);
    converteParaMaiusculas(string);
    printf("A string depois da conversao: %s\n", string);
    system("PAUSE");
    return 0;
}

void converteParaMaiusculas(char *s){
    while (*s != '\0') {
        if (*s >= 'a' && *s <= 'z')
            *s -= 32; /* converte para a letra maiuscula ASCII */
        ++s; /* incrementa s para apontar o proximo caractere */
    }
}
```

PONTEIROS COMO PARÂMETROS

- Ponteiro constante para um dado não-constante:
 - Um ponteiro constante sempre aponta para o mesmo local de memória,
 - Os dados de aquele local podem ser modificados usando o ponteiro,
 - Este é o caso default quando enviamos um vetor para uma função,
 - Apenas os elementos do vetor podem ser modificados.

```
/* Enviando um vetor para uma função */
/* usando um ponteiro constante para um dado não-constante*/
#include <stdio.h>
#include <stdlib.h>

void inc_vet(int [], int);

main() {
    int vet[5] = {1,2,3,4,5};

    inc_vet(vet, 5);

    system("PAUSE");
    return 0;
}

void inc_vet(int v[], int m) {
    int i;

    for(i=0; i<m; i++)
        v[i]++;
}
```

```
/* Enviando um vetor para uma função */
/* usando um ponteiro constante para um dado não-constante*/
#include <stdio.h>
#include <stdlib.h>

void inc_vet(int * const, int);

main() {
    int vet[5] = {1,2,3,4,5};

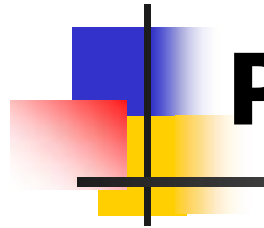
    inc_vet(vet, 5);

    system("PAUSE");
    return 0;
}

void inc_vet(int * const v, int m) {
    int i;

    for(i=0; i<m; i++)
        v[i]++;
}
```

A declaração é interpretada da esquerda para a direita.



PONTEIROS COMO PARÂMETROS

- Ponteiro não-constante para um dado constante:
 - Um ponteiro não-constante pode ser modificado para apontar qualquer item de dado do tipo apropriado,
 - O dado ao qual ele aponta não pode ser modificado,
 - Neste caso a função não pode modificar os dados,
 - Exemplo: a função imprime string.

```
/* Imprime uma string, caracter por caracter */
/* usando um ponteiro não-constante para um dado constante */
#include <stdio.h>
#include <stdlib.h>

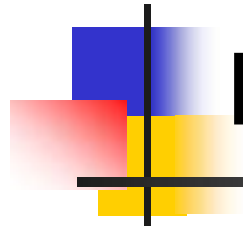
void imprimeCaracteres(const char *);

main() {
    char string[] = "imprime caracteres de uma string";

    printf("A string e:\n");
    imprimeCaracteres(string);
    putchar('\n');
    system("PAUSE");
    return 0;
}

void imprimeCaracteres(const char *s){
    for ( ; *s != '\0'; s++)
        putchar(*s);
}
```

A declaração é interpretada da esquerda para a direita.



PONTEIROS COMO PARÂMETROS

- Ponteiro constante para um dado constante:
 - Garante o princípio de acesso mínimo,
 - O ponteiro sempre aponta para o mesmo local de memória,
 - E os dados nesse local de memória não podem ser modificados,
 - Este caso tem uso pouco freqüente,
 - Exemplo: Imprimir um vetor.

```
/* imprime um vetor, tamanho do vetor enviado como
   ponteiro constante para dado constante */
#include <stdio.h>
#include <stdlib.h>

void prn_vet(int *, const int * const);

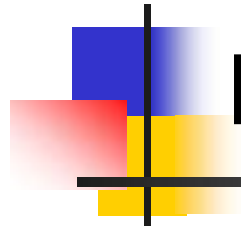
main() {
    int vet[5] = {1,2,3,4,5}, n;

    n=5;
    prn_vet(vet, &n);

    system("PAUSE");
    return 0;
}

void prn_vet(int *v, const int * const m){
    int i;

    for(i=0;i<*m;i++)
        printf("%d\t", v[i]);
    printf("\n");
}
```



ESTRUTURAS COMO PARÂMETROS

- Variáveis de tipo estrutura podem ser utilizadas como parâmetros de funções e como valores de retorno de uma função,
- Ao igual que as outras variáveis por default as estruturas são passadas por valor,
- **Exemplo:** *Crie um programa que lê e imprime os dados de um aluno (nome, idade, sexo, CR). Utilize funções para fazer a leitura e a impressão.*


```
/* Estruturas como parâmetros de funções,  
   passagem por valor  
#include <stdio.h>  
#include <stdlib.h>
```

```
typedef struct{  
    char nome[50];  
    int idade;  
    char sexo;  
    float CR;  
}Taluno;  
  
Taluno le_Aluno(void);  
void prn_Aluno(Taluno);
```

```
int main(){  
    Taluno dado;  
  
    dado = le_Aluno();  
    prn_Aluno(dado);  
  
    system("PAUSE");  
    return 0;  
}
```

```
Taluno le_Aluno(void){  
    Taluno a;  
  
    printf("Digite o nome: ");  
    gets(a.nome);  
    printf("Digite a idade: ");  
    scanf("%d", &a.idade);  
    fflush(stdin);  
    printf("Digite o sexo (M ou F):");  
    scanf("%c", &a.sexo);  
    printf("Digite o CR: ");  
    scanf("%f", &a.CR);  
  
    return a;  
}
```

```
void prn_Aluno(Taluno b){  
    printf("\n--Dados do Aluno--\n");  
    printf("Nome: %s\n", b.nome);  
    printf("Idade: %d\n", b.idade);  
    printf("Sexo: %c\n", b.sexo);  
    printf("CR: %.2f\n", b.CR);  
}
```



ESTRUTURAS COMO PARÂMETROS

- Variáveis de tipo estrutura ocupam grandes quantidades de memória,
- Ao utilizarmos estruturas como parâmetros de funções é recomendável utilizar passagem por referência,
- a passagem por referência evita a sobrecarga associada à criação das cópias da passagem por valor.
- **Exemplo:** *Modifique o exemplo anterior para utilizar passagem por referência.*

```
/* Estruturas como parâmetros de funções,  
   passagem por referencia
```

```
#include <stdio.h>  
#include <stdlib.h>
```

```
typedef struct{  
    char nome[50];  
    int idade;  
    char sexo;  
    float CR;  
}Taluno;
```

```
void le_Aluno(Taluno *);  
void prn_Aluno(const Taluno *);
```

```
int main(){  
    Taluno dado;  
  
    le_Aluno(&dado);  
    prn_Aluno(&dado);  
  
    system("PAUSE");  
    return 0;  
}
```

```
void le_Aluno(Taluno *Ptr){  
    printf("Digite o nome: ");  
    gets(Ptr->nome);  
    printf("Digite a idade: ");  
    scanf("%d", &(Ptr->idade));  
    fflush(stdin);  
    printf("Digite o sexo (M ou F):");  
    scanf("%c", &(Ptr->sexo));  
    printf("Digite o CR: ");  
    scanf("%f", &(Ptr->CR));  
}
```

```
void prn_Aluno(const Taluno *Ptr){  
    printf("\n--Dados do Aluno--\n");  
    printf("Nome: %s\n", Ptr->nome);  
    printf("Idade: %d\n", Ptr->idade);  
    printf("Sexo: %c\n", Ptr->sexo);  
    printf("CR: %.2f\n", Ptr->CR);  
}
```



EXEMPLOS

- *Funções para alocar dinamicamente um vetor de inteiros.*

alocar memória

- recebe a quantidade de elementos
- retorna um ponteiro ao inicio do vetor

liberar memória

- recebe um ponteiro
- não retorna nada

Protótipos

```
int * aloca_vetor(const int);
```

```
void libera_vetor(int *);
```



EXEMPLOS

- *Alocação dinâmica de um vetor . . .*

Definição

```
int * aloca_vetor(const int n){
    int *v;

    v = (int *) malloc(n * sizeof(int));
    if (v==NULL){
        printf("Erro, estouro de memoria!!!\n");
        exit(1);
    }
    return v;
}
```



EXEMPLOS

- *Alocação dinâmica de um vetor . . .*

Definição ...

```
void libera_vetor(int *v) {  
    free(v);  
}
```

Chamadas

```
main() {  
    int *vet, n = 5;  
  
    vet = aloca_vetor(n);  
    ...  
    libera_vetor(vet);  
  
    return 0;  
}
```



EXEMPLOS

- *Funções para alocar dinamicamente uma matriz de floats.*

alocar memória

- recebe a quantidade de linhas
- recebe a quantidade de colunas
- retorna um ponteiro

liberar memória

- recebe um ponteiro
- recebe a quantidade de linhas
- não retorna nada

Protótipos

```
float ** aloca_matriz(const int, const int);  
  
void libera_matriz(float **, const int);
```



EXEMPLOS

- *Alocação dinâmica de matriz . . .*

Definição

```
float ** aloca_matriz(const int l, const int c){
    float **m;
    int i;

    m = (float **) malloc(l * sizeof(float *));
    if (m==NULL){
        printf("Erro, estouro de memoria!!!\n");
        exit(1);
    }
    for(i=0; i<l; i++){
        m[i] = (float *) malloc(c * sizeof(float));
        if (m[i]==NULL){
            printf("Erro, estouro de memoria!!!\n");
            exit(1);
        }
    }
    return m;
}
```




EXEMPLOS

- *Alocação dinâmica de matriz . . .*

Definição ...

```
void libera_matriz(float **m, const int l){  
    int i;  
    for(i=0;i<l;i++)  
        free(m[i]);  
    free(m);  
}
```



EXEMPLOS

- *Alocação dinâmica de matriz . . .*

Chamadas

```
main() {  
    float **mat;  
    int m = 4, n = 5, i, j;  
  
    mat = aloca_matriz(m, n);  
  
    ...  
  
    libera_matriz(mat, m);  
  
    system("PAUSE");  
    return 0;  
}
```



EXEMPLOS

- *Re-escreva o programa de soma e subtração de matrizes, usando funções.*
- Programa original: uma função, 88 linhas
- Funções identificadas:
 - alocar matriz de floats
 - imprime operação aritmética de matrizes
 - liberar matriz

```
#include <stdio.h>
#include <stdlib.h>

//Prototipos de funções
float ** aloca_matriz(const int, const int);

void prn_oper(float **, float **, float **,
              const int, const int, const char);

void libera_matriz(float **, const int);
```

```
int main(){
    float **A, **B, **Som, **Sub;
    int i, j, m, n;

    //Leitura do numero de linhas e colunas
    printf("Digite o numero de linhas da matriz: ");
    scanf("%d", &m);
    printf("Digite o numero de colunas da matriz: ");
    scanf("%d", &n);

    //Alocação dinâmica das matrizes
    A = aloca_matriz(m, n);
    B = aloca_matriz(m, n);
    Som = aloca_matriz(m, n);
    Sub = aloca_matriz(m, n);
```

```
//Leitura de dados e Calculos
```

```
for(i=0;i<m;i++)  
    for(j=0;j<n;j++){  
        printf("Digite o elemento [%d][%d] de A: ", i+1, j+1);  
        scanf("%f", &A[i][j]);  
        printf("Digite o elemento [%d][%d] de B: ", i+1, j+1);  
        scanf("%f", &B[i][j]);  
        Som[i][j] = A[i][j] + B[i][j];  
        Sub[i][j] = A[i][j] - B[i][j];  
    }
```

```
//Resultados
```

```
printf("\nSoma\n");  
prn_oper(A, B, Som, m, n, '+');  
printf("\nSubtracao\n");  
prn_oper(A, B, Sub, m, n, '-');
```

```
//liberando memória
```

```
libera_matriz(A, m);  
libera_matriz(B, m);  
libera_matriz(Som, m);  
libera_matriz(Sub, m);
```

```
system("PAUSE");  
return 0;
```

```
}
```

```
float ** aloca_matriz(const int l, const int c){
    float **m;
    int i;

    m = (float **) malloc(l * sizeof(float *));
    if (m==NULL){
        printf("Erro, estouro de memoria!!!\n");
        exit(1);
    }
    for(i=0; i<l; i++){
        m[i] = (float *) malloc(c * sizeof(float));
        if (m[i]==NULL){
            printf("Erro, estouro de memoria!!!\n");
            exit(1);
        }
    }

    return m;
}
```

```
void libera_matriz(float **m, const int l){
    int i;
    for(i=0; i<l; i++)
        free(m[i]);
    free(m);
}
```

```

void prn_oper(float **m1 , float **m2, float **m3,
              const int l, const int c, const char oper){
    int i, j;

    for(i=0;i<l;i++){
        for(j=0;j<c;j++){
            printf("%4.2f ", m1[i][j]);
            if (i == (l/2)) printf(" %c ", oper); else printf(" ");
            for(j=0;j<c;j++){
                printf("%4.2f ", m2[i][j]);
                if (i == (l/2)) printf(" = "); else printf(" ");
                for(j=0;j<c;j++){
                    printf("%4.2f ", m3[i][j]);
                }
            }
        }
    }
}

```

- Programa modularizado: quatro funções, 90 linhas



BIBLIOTECAS EM C

- Arquivos-cabeçalhos são aqueles que temos mandado o compilador incluir no início de nossos programas,
- Os arquivos de cabeçalho tem extensão (.h),
- Os arquivos de cabeçalho não possuem o código completo das funções, eles só contêm os protótipos das funções,
- Se você criar funções que deseje aproveitar em vários programas é recomendável criar uma biblioteca com essas funções,



BIBLIOTECAS EM C

- Uma biblioteca é formada por:
 - arquivo de cabeçalho que contêm os protótipos de todas as funções da biblioteca,
 - arquivo de funções que contêm as definições de todas as funções da biblioteca.

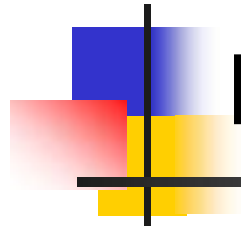
- Exemplo:

Biblioteca matriz → `matriz.h` (protótipos)
 → `matriz.c` (definições)



BIBLIOTECAS EM C

- Para utilizar funções de uma biblioteca em um programa o arquivo de cabeçalho da biblioteca deve ser incluído antes de utilizar a função,
- E recomendável incluir todos os arquivos de cabeçalho no início do programa usando a diretiva de compilação `#include`,
- Para funções da biblioteca padrão:
`#include <nome_arquivo.h>`
- Para bibliotecas criadas pelo usuário:
`#include "nome_arquivo.h"`

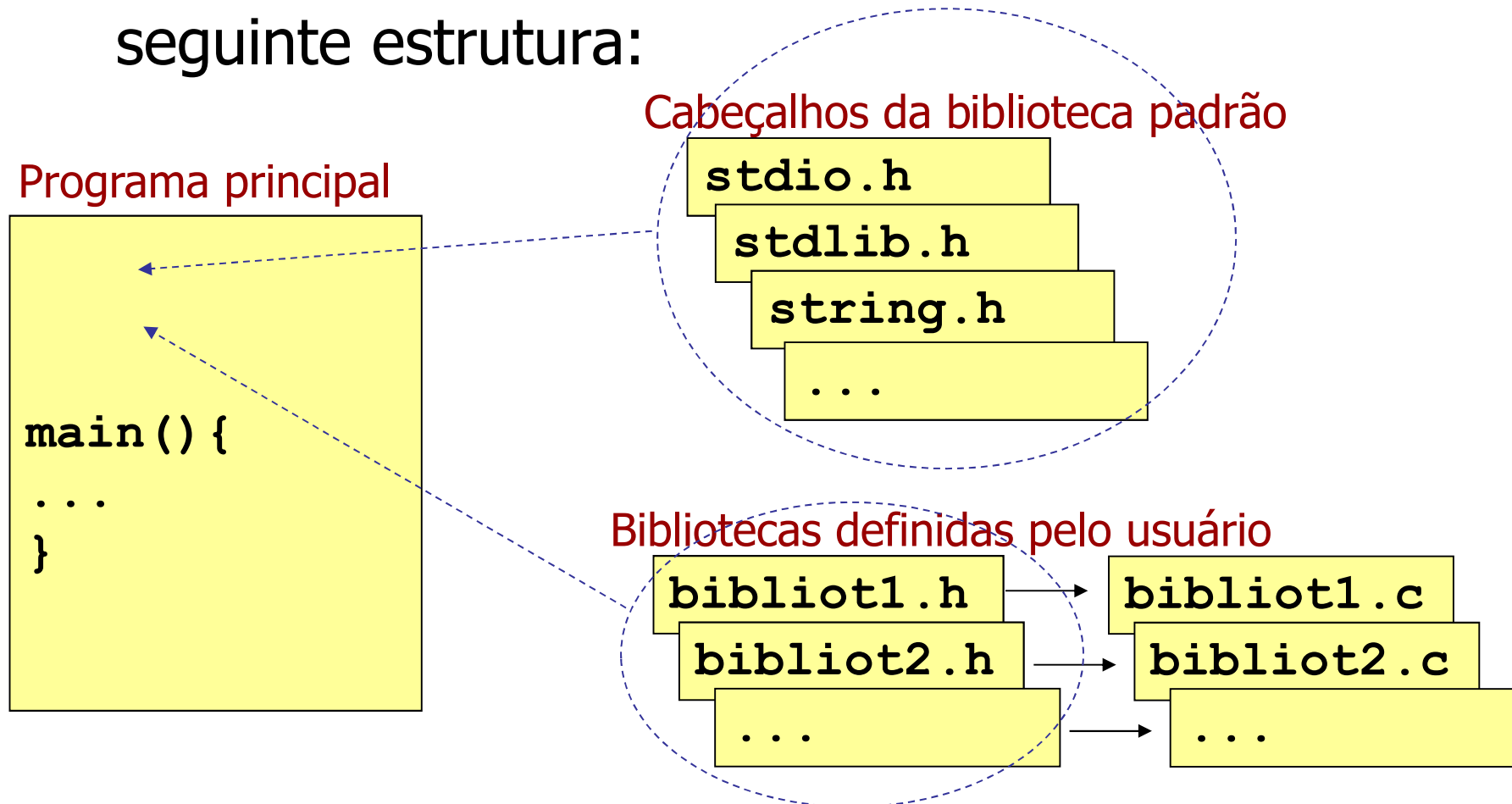


BIBLIOTECAS EM C

- Utilizarmos bibliotecas tem várias vantagens:
 - Permite uma melhor organização e independência em projetos grandes,
 - Possibilita a reutilização e distribuição de código,
 - Vários programadores podem trabalhar simultaneamente em um projeto,
 - Utiliza o modelo de compilação separada do C, para diminuir o tempo de compilação.

BIBLIOTECAS EM C

- Utilizando bibliotecas um programa em C tem a seguinte estrutura:





EXEMPLO

- *Crie uma biblioteca com as funções*


```
float ** aloca_matriz(const int, const int);
```

```
void prn_oper(float **, float **, float **,  
              const int, const int, const char);
```

```
void libera_matriz(float **, const int);
```

*do programa de soma e subtração de matrizes.
Modifique o programa para um projeto com
vários arquivos que utilize a biblioteca.*

Programa principal (prg.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "matriz.h" 

int main(){
    float **A, **B, **Som, **Sub;
    int i, j, m, n;

    //Leitura do numero de linhas e colunas
    ...

    //Alocação dinâmica das matrizes
    ...

    //Leitura de dados e Calculos
    ...

    //Resultados
    ...

    //liberando memória
    ...

    system("PAUSE");
    return 0;
}
```

Cabeçalho da Biblioteca - prototipos (matriz.h)

```
//Prototipos das funções  
float ** aloca_matriz(const int, const int);  
  
void prn_oper(float **, float **, float **,  
              const int, const int, const char);  
  
void libera_matriz(float **, const int);
```

Definição da Biblioteca - definições (matriz.c)

```
//Definição de funções
#include <stdio.h>
#include <stdlib.h>

float ** aloca_matriz(const int l, const int c){
    float **m;
    int i;


    ...
    return m;
}

void libera_matriz(float **m, const int l){
    int i;

    ...
}

void prn_oper(float **m1 , float **m2, float **m3,
              const int l, const int c, const char oper){
    int i, j;

    ...
}
```





EXEMPLO ...

- Crie um projeto no DevC++
 - Arquivo -> Novo -> Projeto
 - Opções: Aplicação Console, Projeto C
 - Salvar o projeto
- Adicione arquivos ao projeto
 - Click Direito -> Adicionar Arquivos
- Compile e Execute o projeto



NUMEROS ALEATÓRIOS

- Uma aplicação divertida e popular da programação é sua utilização, em criar jogos e simulações.
- Na maioria dos jogos de azar, o *fator sorte* atrai a maioria das pessoas, o mesmo pode ser introduzido em aplicações em C utilizando a função `rand()` para gerar números aleatórios.
- `i = rand()`
- a função `rand()` gera um inteiro entre 0 e `RAND_MAX`,



NUMEROS ALEATÓRIOS

- `RAND_MAX` é uma constante simbólica definida no arquivo de cabeçalho `<stdlib.h>`
- na maioria dos compiladores `RAND_MAX = 32767`,
- em muitas aplicações o conjunto de valores gerados com `rand()`, é diferente do necessário para uma determinada aplicação
- lançamento de uma moeda (0 ou 1),
- jogo de dados (1 a 6),



NUMEROS ALEATÓRIOS

- em tais casos é necessário fazer um ajuste de escala ou deslocamento da escala,
- para gerar números no intervalo $[a, b]$ utilize a expressão:
 - $i = a + \text{rand}() \% (b - a + 1)$
 - a primeiro número do intervalo desejado,
 - b último número do intervalo



NUMEROS ALEATÓRIOS

- Exemplo: *Crie um programa que simule o lançamento de um dado 20 vezes e imprima o valor de cada lançamento.*

```
int main()
{
    int i;

    for(i=1; i<=20; i++){
        printf("%8d", 1 + (rand() % 6));

        if (i%5==0) printf("\n");
    }

    return 0;
}
```



NUMEROS ALEATÓRIOS

- na realidade a função `rand()` gera números pseudo-aleatórios,
- ao chamar `rand()` repetidamente produz números aparentemente aleatórios, a mesma seqüência se repete cada vez que o programa for executado.
- para gerar números realmente aleatórios a função `srand()` deve ser utilizada,



NUMEROS ALEATÓRIOS

- a função `srand()`, utiliza um argumento inteiro sem sinal para ser a *semente* da função `rand()`, de forma que seja produzida uma seqüência diferente de números aleatórios cada vez que o programa for executado.
- o protótipo da função `srand()` encontrasse em `<stdlib.h>`



NUMEROS ALEATÓRIOS

- Exemplo: *Modifique o exemplo anterior para gerar números verdadeiramente aleatórios.*

```
int main()
{
    int i, semente;

    printf("Entre com a semente:");
    scanf("%d", &semente);
    srand(semente);

    for(i=1; i<=20; i++){
        printf("%8d", 1 + (rand() % 6));

        if (i%5==0) printf("\n");
    }
    return 0;
}
```




NUMEROS ALEATÓRIOS

- se desejássemos randomizar sem necessidade de introduzir uma semente cada vez, devemos procurar uma semente dentro do programa.
- Geralmente é utilizado:

```
srand (time (NULL) ) ;
```

- a função `time()` retorna o valor do relógio do computador em segundos,
- o protótipo da função `time()` se encontra em `<time.h>`
- **Exercício:** *Modifique o programa do exemplo anterior para randomizar lendo o relógio do sistema.*



EXEMPLOS

- *Crie uma função para gerar números aleatórios num intervalo $[a, b]$. Use a função para:*
 - a) Imprimir três números entre 1 e 3.*
 - b) Imprimir um número entre 1 e 6.*
 - c) Imprimir 10 números entre 3 e 10.*

```
#Tres numeros entre 1 e 5
```

```
4 5 2
```

```
#Um numero entre 1 e 6
```

```
4
```

```
#Dez numeros entre 3 e 10
```

```
7 7 9 8 3 6 6 6 4 10
```

```
int gera_num(const int, const int);

int main(){
    int a, b, i;

    srand(time(NULL));

    printf("#Tres numeros entre 1 e 5\n");
    for(i=0;i<3;i++)
        printf("%d\t", gera_num(1, 5));
    a = 1; b = 10;
    printf("\n#Um numero entre 1 e 6\n");
    printf("%d", gera_num(a, 6));
    a = 3;
    printf("\n#Dez numeros entre 3 e 10\n");
    for(i=0;i<10;i++)
        printf("%d\t", gera_num(a, b));
    return 0;
}

int gera_num(const int ei, const int ed){
    return (ei + rand() % (ed-ei+1));
}
```