

ponteiros

& → endereço

* → conteúdo

- permite manipular o endereço de memória.

ENDEREÇO DE UMA VARIÁVEL: local onde a variável é armazenada em memória.

- ↳ os ponteiros são usados para acessar endereços em memória

OPERADOR DE ENDEREÇO &: usado para acessar endereço de memória de uma variável

- ↳ permite que o programador manipule os dados associados a ela.

- ↳ não é permitido modificar o endereço de uma variável por meio de atribuição

DECLARANDO PONTEIROS:

- ↳ não declarados usando o operador * seguido de dados específicos

↳ int* ponteiro;

ponteiro = & valor;

tipo apontado* nome_da_variável;

IMPRIMINDO O VALOR DE UM PONTEIRO: usa o %p

O PONTEIRO NULO NULL: um ponteiro nulo é um ponteiro que não aponta para nenhum objeto;

- ↳ inicializa um ponteiro;

- ↳ é usado para indicar ausência de um valor válido;

~ aritmética de ponteiros:

- ↳ manipulam espaços de memória e os dados.

108	P+2
104	P+1
100	P
96	{ descreve ou zebra as posições da memória}
92	P-1
88	P-2
84	P-3

EXERCÍCIO DO SLIDE

```
long * p1, * p2;  
int j;  
char * p3;
```

$p2 = p1 + 4;$ ✓ armazena no ponteiro $p2$ o endereço das Unidades de memória a frente de $p1$

$j = p2 - p1;$ ✓ → retorna um valor inteiro

$j = p1 - p2;$ ✓

$p1 = p2 - 2;$

$p3 = p1 - 1;$

$j = p1 - p3$

USANDO FUNÇÕES COM PONTEIROS:

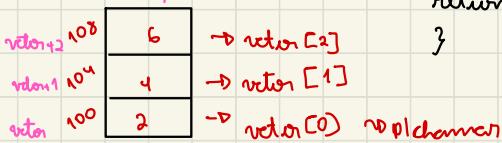
VETORES E MATRIZ

Want arrays → armazenam dados

→ um espaço é criado para armazenar esses valores

allocacão dinâmica → int main (void) {

Para alocar 10 unid.
novo de
memória.



```
int vetor [3] = {2, 4, 6};  
return 0;  
}
```

→ ou 5 vetores [0] → para alocar o conteúdo: *vetor → *(vetor+1); *(vetor+2)

→ era é uma alocação estática (não adiciona dados).

→ na alocação dinâmica, usamos um ponteiro

```
int main (void) {
```

```
int * vetor = malloc (3 * sizeof (int));
```

```
if (vetor == NULL) {
```

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

```
exit (1);
```

```
}
```

```
free (vetor);
```

```
return 0;
```

```
}
```

→ reserva de um espaço de memória

→ retorna um endereço de memória

→ O que é endereço

→ não usado, i.e. x=100.

→ reserva o espaço de memória para outras tarefas da
programação fazer uso da memória

→ a função calloc inicializa as posições com 0.

→ modifica o valor do vetor.

→ a função realloc é responsável realocar uma área de memória já alocada.

ponteiros de variáveis

- uma função pode retornar um tipo de valor para função que chama.
 - ↳ a possibilidade de retornar um valor não sempre é satisfatória
- muitas vezes precisaremos transferir mais do que um valor para a função que chama.
 - ↳ isso não pode ser feito com o retorno explícito de valores

• Para ilustrar isso, inicialmente, vamos considerar uma função simples que calcula a soma de dois valores inteiros

```
1 #include <stdio.h>
2
3 int soma (int a, int b){
4     int c;
5     c = a + b;
6     return c;
7 }
8
9 int main (void){
10    int s;
11    s = soma (3, 5);
12    printf("Soma = %d\n", s);
13
14    return 0;
15 }
```

↳ esse exemplo não apresenta nenhuma dificuldade, pois o resultado da soma pode ser retornado de forma explícita.

↳ o problema aparece quando definirmos que a função resulte em mais de um valor.

• Como exemplo, vamos considerar agora uma função para calcular a soma e o produto de dois números

```
1 #include <stdio.h>
2
3 void somaprod (int a, int b, int c, int d){
4     c = a + b;
5     d = a * b;
6 }
7
8 int main (void){
9     int s, p;
10    somaprod(3, 5, s, p);
11    printf("Soma = %d produto = %d\n", s, p);
12
13    return 0;
14 }
```

↳ essa seria uma forma INCORRETA de implementar essa função.

↳ s e p não foram inicializados na função main, e seus valores não serão alterados.
↳ vai retornar valores "livre"!

↳ alterados não os valores de c e d dentro da função somaprod mas eles não representam os variáveis da função main e não espelham os valores inicializados com os valores de c e d.

• Para que a função que chama tenha acesso aos dois valores calculados, é preciso entender antes o conceito de ponteiros para variáveis.

variáveis do tipo ponteiro

- a linguagem C permite o armazenamento e manipulação de valores e endereços de memória.
 - para cada tipo existente, há um tipo de ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.
- ↳ os ponteiros permitem a transferência de dados entre funções, permitindo aos programadores armazenar e alterar dados de outras partes do programa.

```
int a;
```

↳ declaramos uma variável de nome a que pode armazenar valores inteiros

↳ automaticamente, reservar-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes)

* assim como declaramos variáveis para armazenar inteiros, podemos declarar variáveis, os quais em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória em que há valores inteiros armazenados

```
int* p;
```

↳ usamos a mesma palavra de tipo com os mesmos das variáveis precedidos por asterisco (*)

↳ declaramos uma variável de nome p que pode armazenar endereços de memória em que existem inteiros armazenados

ENDERECO DE UMA VARIÁVEL: local onde a mesma é armazenada

&

↳ endereço de

- resulta no endereço da posição da memória reservada para a variável.
- é usado para acessar o endereço de memória de uma variável, permitindo que o programador manipule os dados associados a ela.
- não é permitido modificar o endereço de uma variável por meio de uma atribuição.

```
● ● ●
1 int numero = 2; // Cria uma variável inteira 'numero' com valor 2.
2 int *ponteiro; // Declara um ponteiro para inteiro chamado 'ponteiro'.
3
4 ponteiro = &numero; // Faz o ponteiro 'ponteiro' apontar para o endereço de memória da variável 'numero'.
```

*

↳ conteúdo de

- acessa o conteúdo do endereço de memória armazenado pela variável ponteira.
- usando o operador *, podemos acessar o conteúdo da variável referenciada pelo ponteiro, permitindo acesso direto ao conteúdo da variável.

```
● ● ●
1 float var = 3.14;
2 float *pointeiro; // Declara um ponteiro 'pointeiro'.
3 float pi = *pointeiro; // Atribui a 'pi' o valor de 'var' através do ponteiro.
4 *pointeiro = 3.1456; // Altera o valor de 'var' (é consequentemente 'pi') para 3.1456 através do ponteiro.
```

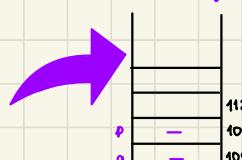
↳ quando alteramos o conteúdo, acontece uma desreferenciação

na prática

```
● ● ●
1 //variável inteira
2 int a;
3
4 //variável p/ ponteiro
5 int *p;
```

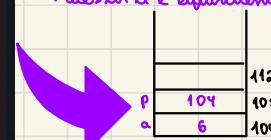
```
● ● ●
1 //a recebe o valor 5
2 a = 5;
3
4 /*p recebe o endereço de a
5 p = &a;
6
7 //conteúdo de p recebe o valor 6
8 *p = 6;
```

↳ as variáveis armazenam valores liso, pois ainda não foram inicializadas



↳ fazendo as atribuições, a variável a receber, indiretamente, os valores.

↳ acessar a é equivalente a acessar *p, pois p armazena o endereço de a



declarando ponteiros

- os ponteiros são declarados usando o operador '*' seguido do tipo de dados específico

```
1 /*declarando ponteiros:  
2 tipo_apontado * nome_da_variável_ponteiro;*/  
3 int * point;  
4  
5 //imprimindo o valor de um ponteiro:  
6 printf("O ponteiro: %p\n", point);  
7 //usa %p
```

o ponteiro nulo

↳ NULL

- um ponteiro nulo é um ponteiro que não aponta para nenhum objeto
- é usado para indicar ausência de um valor válido.

```
1 #include <stdlib.h> //Inclui a biblioteca padrão para alocação dinâmica de memória.  
2 char * point; //Declara um ponteiro para caractere chamado 'point'.  
3 point = NULL; //Indica que não aponta para nenhum endereço de memória específico.
```

compatibilidade e conversões entre ponteiros

- ponteiros são versáteis, permitindo aos programadores criar algoritmos complexos e modernos
- a conversão entre diferentes tipos de ponteiros pode ser usada para acessar memória e dados de forma segura

```
1 float number; //Declara uma variável float chamada 'number'.  
2 int * pointer; //Declara um ponteiro para inteiro chamado 'pointer'.  
3 pointer = &number; //Faz o ponteiro 'pointer' apontar para o endereço de memória da variável 'number', mas isso pode causar problemas de tipo.  
4 pointer = (int*) &number; //Corrigi o problema de tipo, convertendo o ponteiro para apontar corretamente para o endereço de memória de 'number'.
```

↳ o (int*) força que o número seja do tipo int*

aritmética de ponteiros

- a aritmética de ponteiros permite que os programadores manipulem memória e dados usando operações matemáticas simples
- o cálculo de deslocamento de ponteiros pode ser usado para acessar dados específicos em estruturas de dados complexas.
- considerando que p é um ponteiro:
 - soma de um inteiro a um ponteiro: p+2
 - subtração de um inteiro a um ponteiro: p-3
 - incremento de ponteiro: ++p ou p++
 - decremento de ponteiro: --p ou p--
 - subtração entre dois ponteiros do mesmo tipo: p-p2

```
1 int * pointer;  
2 pointer + 3;  
3 /*representa o endereço da posição de  
4 memória que está três objetos do tipo  
5 int adiante do endereço do objeto para  
6 o qual pointer aponta*/
```

...	...
112	P+3
108	P+2
104	P+1
100	P
96	P-1
92	P-2
88	P-3
...	...



SOBE E DESCE AS POSIÇÕES DA MEMÓRIA

usando funções com ponteiros

- já vimos que as funções não podem alterar diretamente valores de variáveis da função que faz a chamada.
 - mas se passarmos para uma função os valores dos endereços de memória em que suas variáveis estão armazenadas, essa função pode alterar, indiretamente, os valores das variáveis da função que a chamou.
- Numa das maneiras de usar ponteiros na linguagem C é passar ponteiros como argumentos para uma função. Isso permite que a função acesse e modifique o valor de uma variável fora do seu próprio escopo.
- retomando o exemplo de uma função para calcular a soma e o produto de dois números inteiros: a solução para esse problema é fazer com que a função somaprod receba os endereços das variáveis da função main e, assim, alterar os valores indiretamente.

```
● ● ●
1 #include <stdio.h>
2
3 //função chamada somaprod que recebe dois inteiros (a e b) e dois ponteiros para inteiros (p e q).
4 void somaprod(int a, int b, int *p, int *q){
5     *p = a + b;
6     *q = a * b;
7     //calcula a soma de a e b, armazenando o resultado em *p, e o produto de a e b, armazenando o resultado em *q.
8 }
9
10 int main (void){
11     int s, p;
12
13     somaprod(3, 5, &s, &p); //Chama a função somaprod com os argumentos 3, 5, e os endereços de s e p.
14     printf("Soma = %d. Produto = %d\n", s, p); //Imprime na tela a soma e o produto calculados pela função somaprod
15
16     return 0;
17 }
18 }
```

EXEMPLOS DOS SLIDES:

```
● ● ●
1 // função chamada incrementa que recebe um ponteiro para inteiro como argumento.
2 void incrementa (int * ponteiro){
3     (*ponteiro)++; //incrementa o conteúdo do ponteiro
4 }
5
6 int main (){
7     int variavel = 5;
8     incrementa (&variavel); //passa o endereço da variável como argumento
9     printf("Valor da variável: %d", variavel);
10    return 0;
11 }
```

- a função somaprod recebe o endereço de duas variáveis, armazena os valores calculados nesses endereços de memória e altera os valores das variáveis originais

```
1 // função chamada cria_variavel que retorna um ponteiro para inteiro.
2 int *cria_variavel () {
3     int variavel = 5;
4     return &variavel;
5     //cria uma variável intira local chamada variavel com valor 5 e retorna o endereço dessa variável.
6 }
7
8 int main () {
9     int *ponteiro = cria_variavel(); //declara um ponteiro para inteiro chamado ponteiro e inicialização com o valor retornado pela função cria_variavel.
10    printf("Valor da variável: %d", *ponteiro);
11
12    return 0;
13 }
```

PONTEIROS COMO RETORNO DE FUNÇÃO: permite que uma função retorne um endereço de memória para uma variável.

ponteiros de funções

- os ponteiros de função não apontam para uma função específica, permitindo que a função possa ser chamada diretamente
- também permitem que a função seja passada como parâmetro para outras funções
- Para declarar um ponteiro de função, você deve especificar o tipo de retorno da função seguido do nome do ponteiro e o tipo de parâmetros entre parenteses:

```
1 int (*ponteiro)(int, int);
```

Neste exemplo, estou declarando um ponteiro chamado "ponteiro" que é um ponteiro para uma função que tem um tipo de retorno inteiro e dois parâmetros inteiros.

EXEMPLO DOS SLIDES:

```
1 //função chamada soma que recebe dois inteiros como argumentos e retorna a soma deles.
2 int soma (int a, int b){
3     return a + b;
4 }
5
6 // função chamada calcula que recebe dois inteiros (x e y) e um ponteiro para uma função que recebe dois inteiros e retorna um inteiro (operacao).
7 int calcula (int x, int y, int (*operacao)(int, int)){
8     return (operacao)(x,y);
9     //chama a operacao passando os argumentos x e y e retorna o resultado.
10 }
11
12 int main (){
13     int resultado = calcula (5, 3, soma); //Chama a função calcula passando os valores 5, 3 e a função soma como operacao. O resultado é armazenado na variável resultado.
14     printf("Resultado: %d", resultado);
15
16     return 0;
17 }
```

passando ponteiros para funções

EXEMPLO DO LIVRO

- alocação dinâmica

↳ malloc, calloc, realloc e free

malloc → aloca a memória

calloc → também aloca e cada espaço é inicializado com 0

realloc → aumentar ou diminuir uma memória previamente alocada

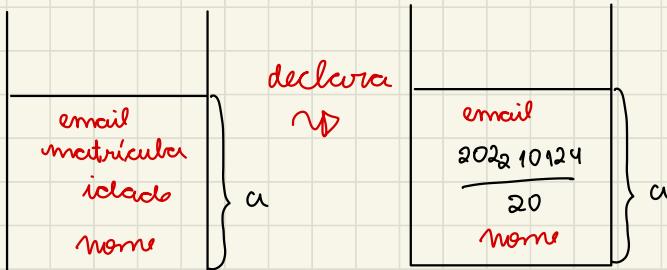
free → liberar memória

Struct → permite agrupar dados distintos em um único espaço de memória

↳ Struct nome {

variáveis ...

};



a · o que quer = dado atribuído

alocação dinâmica: vetores e matrizes

importância dos vetores e matrizes

- vetores e matrizes são estruturas de dados fundamentais para a programação em C
- permitem o armazenamento de dados em estruturas de dados de forma eficiente
- permitem aos programadores criar algoritmos para resolver problemas complexos
- permitem aos programadores armazenar dados de forma organizada e acessar esses dados de forma rápida e eficiente

aplicabilidade

- são utilizados em jogos para armazenar informações sobre os jogadores e as ações que desempenham
- sistemas de inteligência artificial, para processar e armazenar informações sobre o ambiente
- aplicativos de negócios, para armazenar e processar dados financeiros, sendo usados para criar modelos de previsão de mercado, para ajudar os investidores a tomar decisões informadas
- as matrizes são usadas na criação e manipulação de imagens digitais, permitindo que aja aplicativos filtros e transformações que possibilitam a edição de fotos
- também são usadas em sistemas de detecção de objetos e reconhecimento de imagens, permitindo que os computadores possam identificar e interpretar imagens

vantagens dos vetores e matrizes

- vetores e matrizes oferecem várias vantagens para os programadores:
 - armazenamento e acesso de dados de forma eficiente
 - criação de algoritmos para resolver problemas complexos de forma rápida e eficiente
 - códigos mais limpos e reutilizáveis
 - códigos mais compactos, já que os programadores podem testar seus algoritmos com mais facilidade

vetores e alocação dinâmica

- alguns códigos não precisam armazenar o conjunto de valores, mas, em muitas aplicações, necessita-se armazenar o conjunto de valores na memória da computador para depois efetuar os cálculos em cima destes valores
- a forma mais simples de estruturar um conjunto de dados é por meio dos vetores

DEFINIÇÃO DE UM VETOR: int v[10];

- ↳ uma declaração diz que v é um vetor de inteiros dimensionado com 10 elementos, ou seja, reservamos um espaço de memória contínuo para armazenar 10 valores inteiros
- ↳ se cada int ocupa 4 bytes, a declaração reserva um espaço de memória de 40 bytes.
- ↳ o acesso a cada elemento do vetor é feito por meio de uma indexação da variável v.
- ↳ a indexação de um vetor varia de 0 a n-1, onde n representa a dimensão do vetor.

V[0] → acessa o primeiro elemento de v
V[1] → acessa o segundo elemento de v
...
V[9] → acessa o último elemento de v
V[10] → está errado (inversão de memória)

- Para exemplificar o uso de vetores, vamos usar o problema de cálculo da média e variância de um conjunto de valores, vamos considerar que desejamos calcular esses valores para um conjunto de dez números reais.
- no exemplo a seguir, os valores são lidos e armazenados no vetor, depois efetuamos os cálculos da média e da variância sobre o conjunto de valores armazenado

```
● ○ ●
1 /*Cálculo da média e da variância de 10 números*/
2
3 #include <stdio.h>
4
5 int main (void){
6     float v[10]; //declara vetor com 10 elementos
7     float med, var; //variáveis para a média e a variância
8     int i; //variável usada como índice do vetor
9
10 }
11
12 //Leitura dos valores
13 for(i = 0; i < 10; i++) //faz índice variar de 0 a 9
14     scanf("%f", &v[i]); //lê cada elemento do vetor
15
16 //cálculo da média
17 med = 0.0f; //inicializa média com zero
18 for(i = 0; i < 10; i++)
19     med = med + v[i]; //acumula soma de elementos
20 med = med / 10; //calcula a média
21
22 //cálculo da variância
23 var = 0.0f; //inicializa com zero
24 for (i = 0; i < 10; i++)
25     var = var+(v[i]-med)*(v[i]-med); //acumula
26 var = var / 10; //calcula a variância
27
28 //exibição do resultado
29 printf("Media = %f Variância = %f \n", med, var);
30 return 0;
```

- 4 devemos observar que passarmos para a função só tem o endereço de cada elemento do vetor ($\&v[i]$), não desejamos o armazenamento dos valores capturados nos elementos do vetor
- 5 existe uma associação entre os vetores e ponteiros, pois existe a declaração: $\text{int } v[10];$
- 6 o símbolo $\&(vetor)$, é uma constante que representa seu endereço inicial; ou seja: o seu indexação aponta para o primeiro elemento do vetor

- 7 podemos somar e subtrair ponteiros, desde que o valor de ponteiro resultante aponte para dentro da área reservada para o vetor

$v + 0 \rightarrow$ aponta para o primeiro elemento do vetor
$v + 1 \rightarrow$ aponta para o segundo elemento do vetor
$v + 2 \rightarrow$ aponta para o terceiro elemento do vetor
...
$v + 9 \rightarrow$ aponta para o décimo elemento do vetor

Portanto, escrever $\&v[i]$ é equivalente a escrever $(v + i)$. E $v[i]$ é equivalente a escrever $*(v + i)$

↳ a forma indexada é mais clara e adequada.

- 8 os vetores também podem ser inicializados na declaração: $\text{int } v[5] = \{5, 10, 15, 20, 25\};$
- 9 ou simplesmente: $\text{int } v[] = \{5, 10, 15, 20, 25\};$

↳ dimensiona o vetor pelo número de elementos inicializados

passagem de vetores

- 1 passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor
- 2 se passarmos um valor de endereço, a função chamada deve ter um parâmetro do tipo ponteiro para armazenar esse valor
- 3 se passarmos para uma função um vetor int, devemos ter um parâmetro do tipo int^* , capaz de armazenar endereços de inteiros
- 4 os elementos do vetor não são copiados para a função, o argumento aponta é apenas o endereço do primeiro elemento
- 5 para exemplificar, vamos modificar o código anterior, usando funções separadas para o cálculo da média e variância

```

1 //cálculo da média e variância de 10 reais (segunda versão)
2
3 #include <stdio.h>
4
5 //função para cálculo da média que recebe um inteiro n e um ponteiro v real
6 float media(int n, float *v){
7     int i;
8     float s = 0.0f;
9     for(i = 0; i < n; i++)
10         s += v[i];
11     return s/n; //Retorna a média calculada dividindo o valor acumulado s pelo número de elementos n.
12 }
13
14 //Função para cálculo da variância que recebe um variável do tipo inteiro, um ponteiro real e uma variável real
15 float variancia(int n, float *v, float real){
16     int i;
17     float s = 0.0f;
18     for(i = 0; i < n; i++)
19         s += (v[i] - real) * (v[i] - real);
20 }
21
22 int main(void){
23     float v[10]; //Declaração de um vetor com 10 caracteres
24     float med, var;
25     int i;
26     //Entrada dos dados
27     for(i = 0; i < 10; i++)
28         scanf("%f", &v[i]);
29
30     med = media(10, v); //Chama a função media e passa seus atributos
31     var = variancia(10, v, med); //Chama a função variância e passa seus atributos
32     printf("Média = %f\n", med);
33     printf("Variância = %f\n", var);
34     return 0;
35 }
```

- 6 usamos também os operadores de atribuição $+=$ para acumular os valores
- 7 como é passado para a função o endereço do primeiro elemento do vetor, podemos alterar os valores dos elementos do vetor dentro da função

```

1 //Incrementa elementos de um vetor
2
3 #include <stdio.h>
4
5 //declaração da função incr_vetor que recebe os seguintes atributos: um inteiro n e um ponteiro do tipo inteiro chamada v
6 void incr_vetor(int n, int *v){
7     int i;
8     for (i = 0; i < n; i++)
9         v[i]++;
10 }
11
12 int main (void){
13     int a[3] = {1, 3, 5}; //declara e inicializa um vetor
14     incr_vetor(3, a);
15     printf("%d %d %d\n", a[0], a[1], a[2]); //imprime as posições e a saída é: 2 4 6
16
17     return 0;
18 }

```

alocação de memória

- dimensionar um vetor nos obriga a reservar quanto espaço necessário (tínhamos que prever um número máximo de elementos).
- ↳ essa tri-dimensionamento é um fator limitante
- ↳ se dimensionar um vetor muito alto, ocorre um desperdício de memória
- a linguagem C oferece meios de requisitar espaços de memória em tempo de execução

uso da memória

- existem três maneiras de reservar espaço de memória para o armazenamento de informações:

VARIÁVEIS GLOBAIS (E ESTÁTICAS): espaço reservado existe enquanto o programa estiver sendo executado.

VARIÁVEIS LOCAIS: o espaço só existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina.

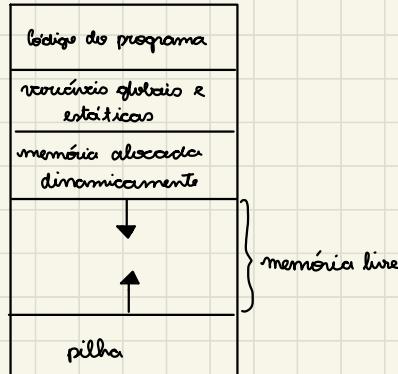
↳ as variáveis globais e locais podem ser simples ou vetores

ALOCAÇÕES DINÂMICAS: requisitar ao sistema, em tempo de execução, um espaço de determinado tamanho. Esse espaço alocado dinamicamente permanece reservado até que seja explicitamente liberado pelo programa.

↳ por isso, podemos alocar dinamicamente um espaço de memória em uma função e acioná-la em outra.

↳ se o programa não liberar um espaço alocado, ele será automaticamente liberado quando a execução do programa terminar.

A DISTRIBUIÇÃO SERIA:



funções da biblioteca padrão

- existem funções da biblioteca stdlib que permitem alocar e liberar memória dinamicamente

FUNÇÃO MALLOC: recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

- int * v;

v = malloc(10*4); → alocação dinâmica de um vetor de inteiros com 10 elementos

para ficarmos independentes de compiladores e máquinas, usamos o operador sizeof()

- v = malloc(10 * sizeof(int));

malloc retorna um ponteiro genérico, para um tipo qualquer, representado por void*, que pode ser convertido automaticamente para o tipo apropriado na atribuição

mas é comum fazer a conversão explicitamente:

- v = (int*) malloc(10 * sizeof(int));

se não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (NULL)

```
1 v = (int*) malloc(10*sizeof(int));
2 if(v==NULL){
3     printf("Memória Insuficiente.\n");
4     exit(1); //aborda o programa e retorna 1 para o sistema operacional
5 }
```

para liberar um espaço de memória alocado dinamicamente, usamos a função free, que recebe como parâmetro o ponteiro da memória a ser liberada:

free(v);

```
1 //cálculo da média e da variância de n reais
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main (void){
7     int i, n;
8     float *v;
9     float med, var;
10
11    //Leitura do número de valores
12    scanf("%d", &n);
13    //alocação dinâmica
14    v = (float*) malloc(n*sizeof(float));
15
16    if(v==NULL){
17        printf("Memória insuficiente.\n");
18        return 1;
19    }
20
21    //Leitura dos valores
22    for (i = 0; i < n; i++)
23        scanf("%f", &v[i]);
24    med = media(n, v);
25    var = variância(n, v, med);
26    printf("Media = %f Variancia = %f \n", med, var);
27
28    free(v);
29    return 0;
30}
```

- a função `malloc()` aloca um bloco contíguo de memória do tamanho especificado, enquanto a função `calloc()` aloca um bloco contíguo de memória e inicializa todos os bytes para 0.

FUNÇÃO REALLOC: é usada para redimensionar uma área de memória já alocada dinamicamente

- `void * realloc (void * ptr, size_t size);`

↳ é útil quando precisamos alocar mais memória para um vetor ou matriz dinâmica, mas não quer perder os dados armazenados nela ou para liberar espaço desnecessário, se o novo tamanho for menor que o antigo.

matrizes

vetores bidimensionais - matrizes

- a linguagem C permite a criação de vetores bidimensionais, declarados estáticamente
- para declarar uma matriz de valores reais com 4 linhas x 3 colunas: float mat [4][3];
- essa declaração reserva um espaço de memória necessário para armazenar os 12 elementos da matriz, que são organizados de maneira contínua, organizados linha a linha
- os elementos da matriz são acessados com indexação dupla: mat[i][j]
 - o elemento da primeira linha e primeira coluna é acessado por mat[0][0]
- as matrizes também podem ser inicializadas na declaração:
- float mat [4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}}
- ou podemos iniciar sequencialmente:
 - float mat [4][3] = {1,2,3,4,5,6,7,8,9,10,11,12}
- o número de elementos por linha pode ser omitido numa inicialização, mas o número de colunas deve ser sempre fornecido.
- float mat [] [3] = {1,2,3,4,5,6,7,8,9,10,11,12}

passagem de matrizes para funções

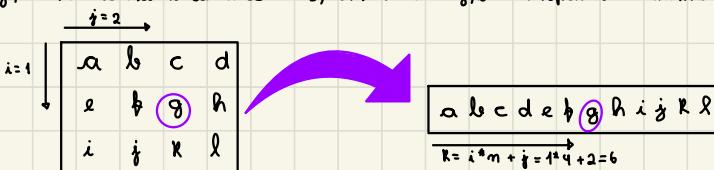
- void f(..., float (*mat)[3], ...); ou void f(..., float mat [] [3], ...);
- de qualquer modo, o acesso aos elementos da matriz dentro da função é feito da forma usual, com indexação dupla

matrizes dinâmicas

- para trabalhar com matrizes alocações dinamicamente, temos que criar abstrações concorrentes com vetores para representar conjuntos bidimensionais

MATRIZ REPRESENTADA POR UM VETOR SIMPLES: reservaremos as primeiras posições do vetor para armazenar os elementos da primeira linha, seguidas dos elementos da segunda linha e assim por diante.

- conceitualmente, trabalharemos com um conjunto bidimensional mas, de fato, temos um vetor unidimensional
- a estratégia de endereçamento para acessar os elementos é a seguinte: se quisermos acessar o que seria o elemento mat[i][j] de uma matriz, devemos acessar o elemento $\rightarrow [k]$, com $k = i \cdot n + j$, onde n representa o número de colunas da matriz



- ↳ se quisermos acessar elementos da terceira linha ($i=2$) linha da matriz, temos de pular duas linhas de elementos ($i \neq m$) e depois indicar o elemento da linha com j .

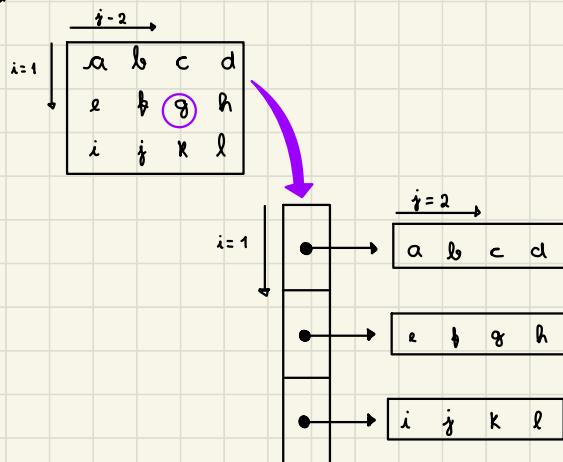
↳ com essa estratégia, a alocação da "matrix" recai em uma alocação de vetor com $m \times n$ elementos, onde $m \times n$ representam as dimensões da matriz.

```
1 float *mat; //matrix representada por um vetor
2 ...
3 mat = (float*) malloc(m*n*sizeof(float));
4 ...
```

- 4) mas somos obrigados a usar uma matr \times cula descomportável, $\{i^k m + j\}$, para acenar os elementos, o que pode deixar o código pouco legível.

MATRIZ REPRESENTADA POR UM VETOR DE PONTEIROS: cada linha da matriz é representada por um vetor independente

- 6) A matriz é representada por um vetor de vetores, ou vetor de ponteiros, no qual cada elemento armazena o endereço do primeiro elemento de cada linha.



- A alocação agora é mais elaborada
 - primeiros termos da alocação de vetores de ponteiros, em seguidos alocamos cada uma das linhas da matriz, atribuindo seus endereços aos elementos do vetor de ponteiros criado

```
1 int i;
2 float **mat; //matrix representada por um vetor de ponteiros
3 ...
4 mat = (float**) malloc(m*sizeof(float));
5 for(i=0; i<m; i++)
6     mat[i] = (float**) malloc(n*sizeof(float));
```

- a grande vantagem dessa estratégia é o acesso aos elementos ser feito da mesma forma que quando temos uma matriz circular estaticamente
 - a liberação do espaço de memória ocupado pelas matrizes também exige a construção de um laço, pois temos de liberar cada linha antes

de liberar os vetores de ponteiros

```
● ● ●  
1 ...  
2 for (i=0; i<m; i++)  
3     free(mat[i]);  
4 free(mat);
```

Operações com matrizes

MATRIZ COM VETOR SIMPLES: float* transposta (int m, int n, float* mat);
n linhas
up colunas

```
● ● ●  
1 float* transposta (int m, int n, float* mat){  
2     int i, j;  
3     float* trp;  
4  
5     //aloca matriz transportada:  
6     trp = (float*) malloc(m*n*sizeof(float));  
7  
8     //preenche matriz  
9     for(i=0; i<n; i++)  
10        for(j=0; i<n; j++)  
11            trp[j*m+i] = mat[i*n+j];  
12    return trp;  
13 }
```

MATRIZ COM VETOR DE PONTEIROS: float** transposta (int m, int n, float** mat);

```
● ● ●  
1 float** transposta(int m, int n, float** mat){  
2     int i, j;  
3     float** trp;  
4  
5     //aloca matriz transposta: n linhas, m colunas  
6     trp = (float**) malloc(n*sizeof(float*));  
7     for (i=0; i<n; i++)  
8         trp[i] = (float*) malloc(m*sizeof(float));  
9  
10    //preenche matriz  
11    for (i=0; i<m; i++)  
12        for (j=0; j<n; j++)  
13            trp[j][i] = mat [i][j];  
14  
15    return trp;  
16 }
```

structs