

Estruturas de Dados

Alocação dinâmica de memória

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

paulo_regis@uvanet.br

Alocação dinâmica de memória

Definição

Funções para alocação de memória

Alocação de arrays multidimensionais

- Sempre que escrevemos um programa, é preciso reservar espaço para os dados que serão processados. Para isso usamos as variáveis.

Variável

Uma variável é uma posição de memória previamente reservada e que pode ser usada para armazenar algum dado.

- Por ser uma posição previamente reservada, uma variável deve ser declarada durante o desenvolvimento do programa.

Problema

Precisamos construir um programa que processe os valores dos salários dos funcionários de uma pequena empresa.

- Uma solução simples para resolver esse problema poderia ser declarar um array do tipo **float**, por exemplo, 1.000 posições:

Warning!

1. Se a empresa tiver menos de 1.000 funcionários: esse array será um exemplo de desperdício de memória.
2. Se a empresa tiver mais de 1.000 funcionários: esse array será insuficiente para lidar com os dados de todos os funcionários.

Ponteiro

Um ponteiro é uma variável que guarda o endereço de um dado na memória.

- É importante lembrar que arrays são agrupamentos sequenciais de dados de um mesmo tipo na memória.

O nome de um array

O nome do array é apenas um ponteiro que aponta para o primeiro elemento do array.

- A linguagem C permite alocar (reservar) dinamicamente (em tempo de execução) blocos de memórias utilizando ponteiros. A esse processo dá-se o nome **alocação dinâmica**.

Alocação dinâmica

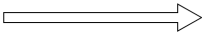
A alocação dinâmica consiste em requisitar um espaço de memória ao computador, em tempo de execução, o qual, usando um ponteiro, devolve para o programa o endereço do início desse espaço alocado.

Exemplo

1. Começando com um ponteiro `int *n` apontando para **NULL**, requisitamos para o computador cinco posições inteiras de memória.
2. O computador, por sua vez, nos devolve as posições de memória de `#123` até `#127` para armazenarmos nossos dados.
3. O ponteiro `n` passa então a se comportar como se fosse um array de tamanho 5, ou seja, `int n[5]`.

Memória		
#	Variável	Conteúdo
119		
120		
121	int *n	NULL
122		
123		
124		
125		
126		
127		
128		
129		

Alocando 5 posições de
memória em int *n



Memória		
#	Variável	Conteúdo
119		
120		
121	int *n	#123
122		
123	n[0]	11
124	n[1]	25
125	n[2]	32
126	n[3]	44
127	n[4]	52
128		
129		

- A linguagem C ANSI usa apenas quatro funções para o sistema de alocação dinâmica, disponíveis na biblioteca **stdlib.h**. São elas:
 - malloc
 - calloc
 - realloc
 - free
- Além destas, o operador **sizeof**, também é bastante usado para auxiliar as demais funções no processo de alocação de memória.

sizeof

O operador **sizeof** é usado para saber o número de *bytes* necessários para alocar um único elemento de determinado tipo de dado. Ele pode ser usado de duas formas:

sizeof nome_da_variável

sizeof (nome_do_tipo)


```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Ponto {
5      int x, y;
6  };
7
8  int main(){
9      int x;
10     double y;
11
12     printf("Tamanho char: %d\n", sizeof(char));
13     printf("Tamanho int: %d\n", sizeof(int));
14     printf("Tamanho float: %d\n", sizeof(float));
15     printf("Tamanho double: %d\n", sizeof(double));
16     printf("Tamanho struct ponto: %d\n", sizeof(struct Ponto));
17     printf("Tamanho da variavel x: %d\n", sizeof x);
18     printf("Tamanho da variavel y: %d\n", sizeof y);
19
20     return 0;
21 }
```

- A função **malloc()** serve para alocar memória durante a execução do programa.
- Ela possui o seguinte protótipo:

```
1 void *malloc (unsigned int num);
```

- A função **malloc()** recebe um parâmetro de entrada **num**: o tamanho do espaço de memória a ser alocado.
- e retorna um ponteiro para a primeira posição do array alocado. Ou **NULL** no caso de erro.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *p, i;
5
6      p = (int *) malloc(5*sizeof(int));
7
8      for (i=0; i<5; i++){
9          printf("Digite o valor da posicao %d: ",i);
10         scanf("%d", &p[i]);
11     }
12
13     return 0;
14 }
```

Tipo de retorno

A função **malloc()** retorna um ponteiro genérico (**void***). Esse ponteiro pode ser atribuído a qualquer tipo de ponteiro via *type cast* (linha 6).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *p, i;
5
6      p = (int *) malloc(5*sizeof(int));
7
8      for (i=0; i<5; i++){
9          printf("Digite o valor da posicao %d: ",i);
10         scanf("%d", &p[i]);
11     }
12
13     return 0;
14 }
```

Uso do sizeof

O operador `sizeof(int)` (linha 6) retorna 4 (número de bytes do tipo `int` na memória). Portanto, são alocados 20 bytes ($5 * 4$ bytes).

- Assim como a função **malloc()**, a função **calloc()** também serve para alocar memória durante a execução do programa.
- Ela possui o seguinte protótipo:

```
1 void *calloc (unsigned int num, unsigned int size);
```

- A função **calloc()** recebe dois parâmetros de entrada **num**: o número de elementos no array a ser alocado e **size**: o tamanho de cada elemento do array.
- e retorna um ponteiro para a primeira posição do array alocado. Ou **NULL** no caso de erro.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *p, i;
5
6      p = (int *) calloc(5, sizeof(int));
7      if(p == NULL){
8          printf("Erro: Memoria Insuficiente!\n");
9      }
10
11     for (i=0; i<5; i++){
12         printf("Digite o valor da posicao %d: ",i);
13         scanf("%d", &p[i]);
14     }
15
16     return 0;
17 }
```

malloc vs. calloc

Uma diferença entre a função `calloc()` e a função `malloc()`: ambas servem para alocar memória, mas a função `calloc()` inicializa todos os **BITS** do espaço alocado com 0s.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int i;
5      int *p1, *p2;
6      p1 = (int *) malloc(20 * sizeof(int));
7      p2 = (int *) calloc(20, sizeof(int));
8
9      printf("calloc \t\t malloc\n");
10     for (i=0; i<20; i++)
11         printf("p1[%d]=%d \t p2[%d] = %d\n", i, p1[i], i, p2[i]);
12
13     return 0;
14 }
```

- A função **realloc()** serve para alocar memória ou realocar blocos de memória previamente alocados.
- Ela possui o seguinte protótipo:

```
1 void *realloc (void *ptr, unsigned int num);
```

- A função **realloc()** recebe dois parâmetros de entrada **ptr**: Um ponteiro para um bloco de memória previamente alocado e **num**: o tamanho em bytes do espaço de memória a ser alocado.
- e retorna um ponteiro para a primeira posição do array alocado. Ou **NULL** no caso de erro.


```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int i;
6      int *p = malloc(5 * sizeof(int));
7
8      for (i = 0; i < 5; i++) {
9          p[i] = i+1;
10         printf("%d ", p[i]);
11     }
12     printf("\n");
13
14     p = realloc(p, 10*sizeof(int)); //Aumenta o tamanho do array
15     for (i = 0; i < 10; i++)
16         printf("%d ", p[i]);
17     printf("\n");
18
19     return 0;
20 }
```

Warning!

Sempre que alocamos memória de forma dinâmica (`malloc()`, `calloc()` ou `realloc()`), é necessário liberar essa memória quando ela não for mais necessária.

- Para liberar um bloco de memória previamente alocado utilizamos a função **free()**
- Ela possui o seguinte protótipo:

```
1 void free (void *p);
```

- A função **free()** recebe apenas um parâmetro de entrada: o ponteiro para o início do bloco de memória alocado.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int *p, i;
6      p = (int *) malloc(50 * sizeof(int));
7      if(p == NULL){
8          printf("Erro: Memoria Insuficiente!\n");
9          exit(1);
10     }
11     for (i = 0; i < 50; i++)
12         p[i] = i+1;
13     for (i = 0; i < 50; i++)
14         printf("%d\n", p[i]);
15     printf("\n");
16
17     free(p); //libera a memória alocada
18
19     return 0;
20 }
```

Exercício 1

Crie uma estrutura representando um aluno de uma disciplina. Essa estrutura deve conter o número de matrícula do aluno, seu nome e as notas de três provas. Escreva um programa que mostre o tamanho em bytes dessa estrutura.

Exercício 2

Crie uma estrutura chamada Cadastro. Essa estrutura deve conter o nome, a idade e o endereço de uma pessoa. Agora, escreva uma função que receba um inteiro positivo N e retorne o ponteiro para um vetor de tamanho N, alocado dinamicamente, dessa estrutura. Solicite também que o usuário digite os dados desse vetor dentro da função.

Exercício 3

Elabore um programa que leia do usuário o tamanho de um vetor a ser lido. Em seguida, faça a alocação dinâmica desse vetor. Por fim, leia o vetor do usuário e o imprima.

Exercício 4

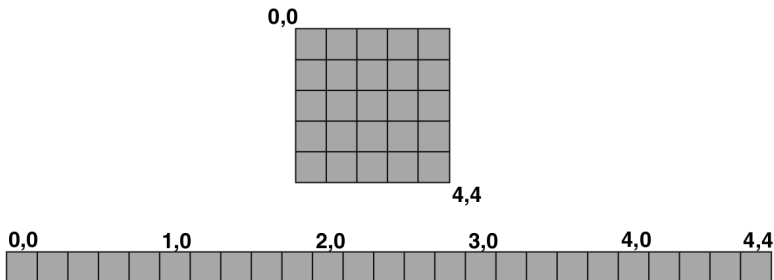
Faça um programa que leia um valor inteiro N não negativo. Se o valor de N for inválido, o usuário deverá digitar outro até que ele seja válido (ou seja, positivo). Em seguida, leia um vetor V contendo N posições de inteiros, em que cada valor deverá ser maior ou igual a 2. Esse vetor deverá ser alocado dinamicamente.

- Na linguagem C existem diferentes soluções de alocação para um array com mais de uma dimensão. Algumas dessas soluções são:
 1. usando array unidimensional
 2. usando ponteiro para ponteiro
 3. usando ponteiro para ponteiro para array.

Solução 1: usando array unidimensional

- Arrays multidimensionais podem ser armazenados linearmente na memória.
- O uso dos colchetes para acessar os índices cria a impressão de estarmos trabalhando com mais de uma dimensão.

```
int mat[5][5];
```



Solução 1: usando array unidimensional

- Uma solução trivial é **simular um array bidimensional** (ou com mais dimensões) utilizando um único array unidimensional alocado dinamicamente.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int i, j, Nlinhas = 2, Ncolunas = 2;
5      int *p = (int *) malloc(Nlinhas * Ncolunas * sizeof(int));
6
7      for (i = 0; i < Nlinhas; i++){
8          for (j = 0; j < Ncolunas; j++){
9              p[i * Ncolunas + j] = i+j;
10         }
11     for (i = 0; i < Nlinhas; i++){
12         for (j = 0; j < Ncolunas; j++){
13             printf("%d ", p[i * Ncolunas + j]);
14             printf("\n");
15         }
16     free(p);
17     return 0;
18 }
```


- O maior inconveniente dessa abordagem é que temos que abandonar a notação de colchetes para indicar a segunda dimensão da matriz.
- É preciso calcular o deslocamento no array para simular a segunda dimensão.
- Isso é feito somando o índice da coluna que se quer acessar ao produto do índice da linha que se quer acessar pelo número total de colunas da "matriz":

$$m[i * N_{colunas} + j]$$


- Se quisermos alocar um array com mais de uma dimensão e manter a notação de colchetes para cada dimensão, precisamos utilizar o conceito de “ponteiro para ponteiro”.

char ****p*;

- Em um ponteiro para ponteiro, cada nível do ponteiro permite criar uma nova dimensão no array.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      char c = 'a';
6      char *p1;
7      char **p2;
8      char ***p3;
9
10     p1 = &c;
11     p2 = &p1;
12     p3 = &p2;
13
14     return 0;
15 }
```

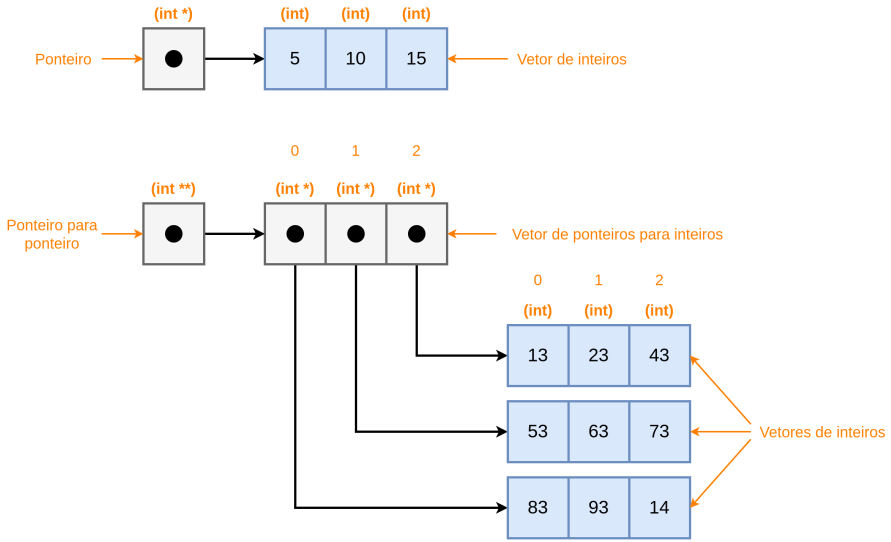
Memória		
#	Variável	Conteúdo
119		
120		
121	char ***p3	#123
122		
123	char **p2	#125
124		
125	char *p1	#127
126		
127	char c	'a'
128		
129		



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int **p; // 2 dimensões
5      int i, j, N = 2;
6
7      p = (int**) malloc(N*sizeof(int*));
8
9      for (i = 0; i < N; i++){
10         p[i] = (int*)malloc(N*sizeof(int));
11
12         for (j = 0; j < N; j++)
13             scanf("%d", &p[i][j]);
14     }
15
16     return 0;
17 }
```

Memória		
#	Variável	Conteúdo
119	int **p	#120
120	p[0]	#123
121	p[1]	#126
122		
123	p[0][0]	69
124	p[0][1]	74
125		
126	p[1][0]	14
127	p[1][1]	31
128		
129		

Solução 2: usando ponteiro para ponteiro



Warning

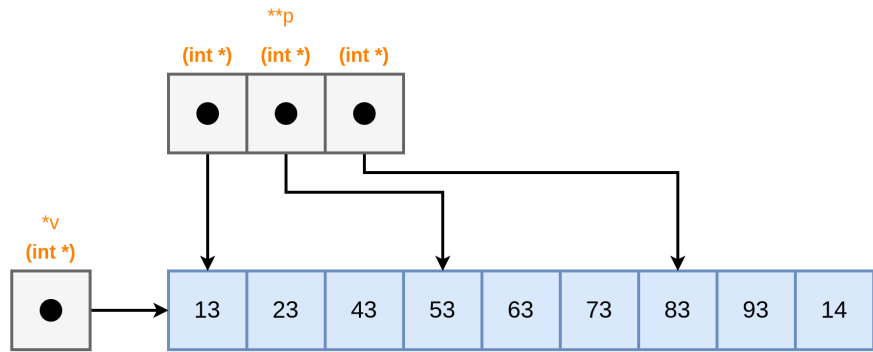
Para liberar da memória um array com mais de uma dimensão é preciso liberar a memória alocada em cada uma de suas dimensões, na ordem inversa da que foi alocada.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int **p, i, j, N = 2;
5      p = (int **) malloc(N*sizeof(int *));
6      for (i = 0; i < N; i++){
7          p[i] = (int *) malloc(N*sizeof(int));
8          for (j = 0; j < N; j++)
9              scanf("%d",&p[i][j]);
10     }
11     for (i = 0; i < N; i++)
12         free(p[i]);
13     free(p);
14     return 0;
15 }
```

- Simulamos um array bidimensional (ou com mais dimensões) utilizando:
 - Um array unidimensional alocado dinamicamente e contendo as posições de todos os elementos.
 - Um array de ponteiros unidimensional que simulará as dimensões e, assim, manterá a notação de colchetes.

Solução 3: ponteiro para ponteiro para array

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(){
4      int *v; //1 dimensão
5      int **p; //2 dimensões
6      int i, j, Nlinhas = 2, Ncolunas = 2;
7      v = (int*) malloc(Nlinhas * Ncolunas * sizeof(int));
8      p = (int **) malloc(Nlinhas * sizeof(int *));
9      for (i = 0; i < Nlinhas; i++){
10         p[i] = v + i * Ncolunas;
11         for (j = 0; j < Ncolunas; j++)
12             scanf("%d",&p[i][j]);
13     }
14     for (i = 0; i < Nlinhas; i++){
15         for (j = 0; j < Ncolunas; j++)
16             printf("%d ",p[i][j]);
17         printf("\n");
18     }
19     free(v);
20     free(p);
21     return 0;
22 }
```

Exercício 5

Escreva uma função que receba como parâmetro um valor N e retorne o ponteiro para uma matriz alocada dinamicamente contendo N linhas e N colunas. Essa matriz deve conter o valor 1 na diagonal principal e 0 nas demais posições.

Exercício 6

Aloque dinamicamente uma matriz $N \times N$ e um vetor de tamanho N , inicialize ambos com números aleatórios entre 0 e 99. Em seguida, escreva uma função que receba como parâmetro uma matriz A contendo N linhas e N colunas, e um vetor B de tamanho N . A função deve retornar o ponteiro para um vetor C de tamanho N alocado dinamicamente, em que C é o produto da matriz A pelo vetor B .