Alocação de Memória

PE-07 – v1.0

Prof. Paulo Joia Filho

Objetivos

Os principais objetivos desta aula são:

- Explorar a utilização de ponteiros.
- Conhecer o funcionamento da alocação dinâmica de memória.
- Compreender a alocação dinâmica de arranjos multidimensionais.

Por que estudar ponteiros?

Entre outras funcionalidades, ponteiros permitem...

- Passar parâmetros para funções **"por referência"**.
- **Alocar e liberar memória dinamicamente** (em tempo de execução).
- Implementação eficiente de certas **estruturas de dados**.

Alocação estática × dinâmica de memória

- Estruturas de dados estáticas têm seu espaço pré-determinado em tempo de compilação.

Pergunta

Então, como criar um arranjo que saiba o tamanho durante a execução do programa?

Alocação estática × dinâmica de memória

- Estruturas de dados dinâmicas têm seu tamanho determinado em tempo de execução.
- Para isso, existem mecanismos que permitem fazer alocação dinâmica de memória em tempo de execução.
- Memória pode ser alocada de uma região chamada **heap**.
- Memória alocada no **heap deve ser liberada** quando não for mais utilizada.

Alocação dinâmica de memória

Funções para manipulação da *heap*

- Encontram-se na biblioteca **`stdlib.h`**.
  - **`malloc`**
  - **`calloc`**
  - **`realloc`**
  - **`free`**

Alocação dinâmica de memória

Função `malloc`

- Aloca um espaço de memória.
- Sintaxe:

```
void* malloc (size_t size);
```

`void*` → ponteiro de tipo não especificado – necessita de `typecast`.

`size` → quantidade de *bytes* a serem alocados.

- Retorno de **`malloc`** é um ponteiro do tipo **`void`** (sem tipo) para o início do espaço de memória alocado.
- Exemplo usando *cast*:

```
int *n;
n = (int *) malloc(sizeof(int));
...
free(n);
```

Alocação dinâmica de memória

Função `free`

- Libera o espaço de memória alocado.
- Sintaxe:

```
void free (void* ptr);
```

`ptr` → um ponteiro que aponta para uma área da memória previamente alocada com **`malloc`**.

- Exemplo:

```
int *n;
n = malloc(sizeof(int));
...
free(n);
```

Alocação dinâmica de memória

Função `calloc`

- Aloca um espaço de memória e inicializa o bloco com zeros.

- Sintaxe:

```
void* calloc (size_t num, size_t size);
```

num → número de elementos a serem alocados.
size → tamanho de cada elemento.

☞ **malloc** apenas aloca um bloco de memória, não inicializa a memória.

Alocação dinâmica de memória

Função `realloc`

- Realoca o bloco de memória.

- Sintaxe:

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

`ptr` → ponteiro para o bloco de memória previamente alocado com **`malloc`**, **`calloc`** ou **`realloc`**.

`size` → novo tamanho para o bloco de memória, em *bytes*.

Alocação dinâmica de memória [aloc_din_char.c]

Exemplo 1 (Alocação de 100 caracteres – Observe o *typecast*)

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5     char *pc;
      pc = (char*) malloc(100 * sizeof(char));
7
      fgets(pc, 100, stdin);
9     printf("%s", pc);

11    free(pc);
      return 0;
13 }
```

Pergunta:

- Sempre conseguirá alocar a memória desejada?

Alocação dinâmica de memória [aloc_din_int.c]

Exemplo 2 (Alocação de x inteiros – Observe o *typecast*)

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5     int x;
      int *pi;
7     scanf("%d", &x);
      pi = (int*) malloc(x * sizeof(int));
9
      if (pi == NULL) {
11       printf("Memória insuficiente. Programa encerrado.\n");
         return -1;
13    }

15    pi[5] = 44; /* Considere x > 5*/

17    /* Várias outras operações*/

19    free(pi);
      return 0;
21 }
```

Alocação dinâmica de memória [aloc_din_int.c]

Exemplo 2 (Alocação de x inteiros – Observe o *typecast*)

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5     int x;
      int *pi;
7     scanf("%d", &x);
      pi = (int*) malloc(x * sizeof(int));
9
      if (pi == NULL) {
11       printf("Memória insuficiente. Programa encerrado.\n");
         return -1;
13    }

15    pi[5] = 44; /* Considere x > 5*/

17    /* Várias outras operações*/

19    free(pi);
      return 0;
21 }
```

Importante:

Em caso de erro, isto é, não conseguir memória disponível para efetuar a alocação, a função **malloc** retorna **NULL**.

Alocação dinâmica de memória [aloc_din_float.c]

Exemplo 3 (Alocação de um arranjo de ponto flutuante)

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5     float *v;
      int tam, i;
7     scanf("%d", &tam);
      v = (float*) malloc(tam * sizeof(float));
9
      if (!v) { /*Verifica se conseguiu alocar */
11       printf("Memória insuficiente. Programa encerrado.\n");
         return -1;
13    }

15    for (i = 0; i < tam; i++) { /* Inicializa o arranjo */
         v[i] = 0;
17    }
      /* Várias outras operações*/
19
      free(v); /* Libera a área de memória alocada */
21    return 0;
   }
```

Alocação dinâmica de arranjos multidimensionais

Ponteiro para ponteiro

- Como sabemos, um arranjo de inteiros equivale a um ponteiro para inteiros:

```
int A[] ↔ int *A;
```

- Logo, um arranjo bidimensional de inteiros equivale a um arranjo de ponteiros para inteiros:

```
int A[][] ↔ int *A[] ↔ **A;
```

- É por isso que a assinatura do método **main** pode ser:

```
int main (int argc, char argv[][]);
int main (int argc, char *argv[]);
int main (int argc, char **argv);
```

Arranjos multidimensionais

[aloc_din_multid.c]

Exemplo 4

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5     float f1 = 27, f2 = 13, *pf1, *pf2, **ppf1, **ppf2;

7     pf1 = &f1;
      pf2 = &f2;
9     printf("%.2f %2.f\n", *pf1, *pf2);

11    ppf1 = &pf1;
      ppf2 = &pf2;
13
      **ppf1 = *pf1 - 1;
15    **ppf2 = *pf2 + 1;

17    printf("%.2f %2.f\n", **ppf1, **ppf2);

19    return 0;
   }
```

O que será impresso?

Arranjos multidimensionais

[aloc_din_multid.c]

Exemplo 4

```
1  #include<stdio.h>
   #include<stdlib.h>
3
   int main(int argc, char *argv[]) {
5     float f1 = 27, f2 = 13, *pf1, *pf2, **ppf1, **ppf2;

7     pf1 = &f1;
      pf2 = &f2;
9     printf("%.2f %2.f\n", *pf1, *pf2);

11    ppf1 = &pf1;
      ppf2 = &pf2;
13
      **ppf1 = *pf1 - 1;
15    **ppf2 = *pf2 + 1;

17    printf("%.2f %2.f\n", **ppf1, **ppf2);

19    return 0;
   }
```

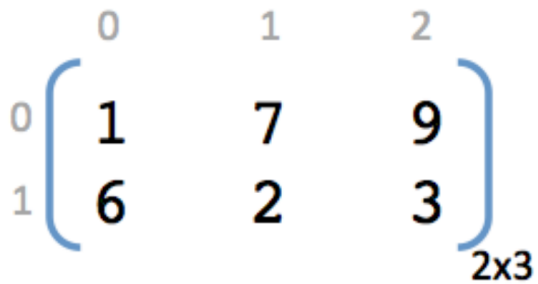
O que será impresso?

```
27.00 13.00
26.00 14.00
```

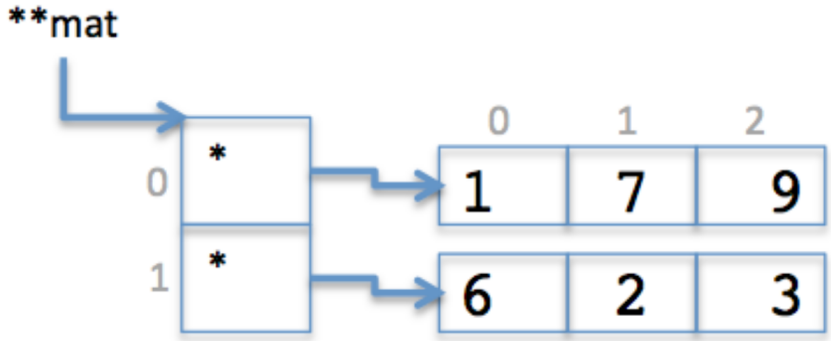
Alocação dinâmica de arranjos multidimensionais

Matriz

Matriz é um arranjo bidimensional. Na verdade, pode ser pensado como um arranjo de arranjos.



(a) Visualização como matriz.



(b) Visualização como ponteiro.

Arranjos multidimensionais

[matriz_din.c]

Exemplo 5 (Alocação de matriz x × y)

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

/* Protótipo */
int soma(int **mat, int x, int y);

int main(int argc, char *argv[]) {
   int **mat;
   int x, y, i, j;
   scanf("%d", &x);
   scanf("%d", &y);

   mat = (int**) malloc(x * sizeof(int*));
   if (!mat) {
      printf("Memória insuficiente.\n");
      return -1;
   }

   for (i = 0; i < x; i++) {
      mat[i] = (int*) malloc(y * sizeof(int));

      if (!mat[i]) {
         printf("Memória insuficiente.\n");
         return -1;
      }
   }
   for (i = 0; i < x; i++) {
      for(j = 0; j < y; j++) {
         scanf("%d", &mat[i][j]);
      }
   }
   printf("Soma dos Elementos: %d\n",
       soma(mat, x, y));

   for (i = 0; i < x; i++) {
      free(mat[i]);
   }
   free(mat);
   return 0;
}

int soma(int **mat, int x, int y) {
   int i, j, soma = 0;
   for (i = 0; i < x; i++){
      for (j = 0; j < y; j++) {
         soma += mat[i][j];
      }
   }
   return soma;
}
```

Alocação dinâmica de arranjos multidimensionais

- Observe o que faz o seguinte código (suponha **`x = 2`**):

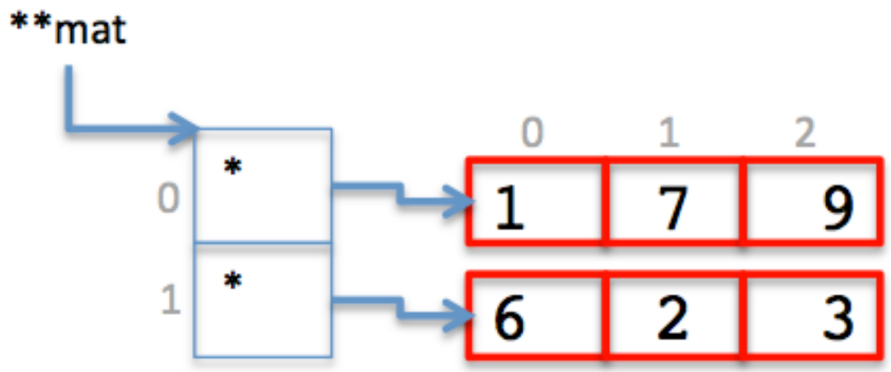
```
mat = (int**) malloc(x * sizeof(int*);
```

Alocação dinâmica de arranjos multidimensionais

- E o que faz o seguinte código de repetição (suponha `y = 3`):

```
for (i = 0; i < x; i++) {
    mat[i] = (int*) malloc(y * sizeof(int));

    if (!mat[i]) {
        printf("Memória insuficiente.\n");
        return -1;
    }
};
```

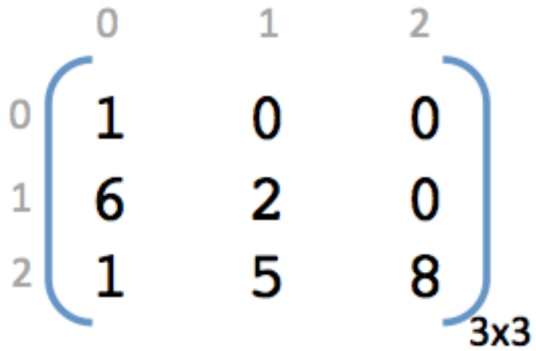


Alocação dinâmica de arranjos multidimensionais

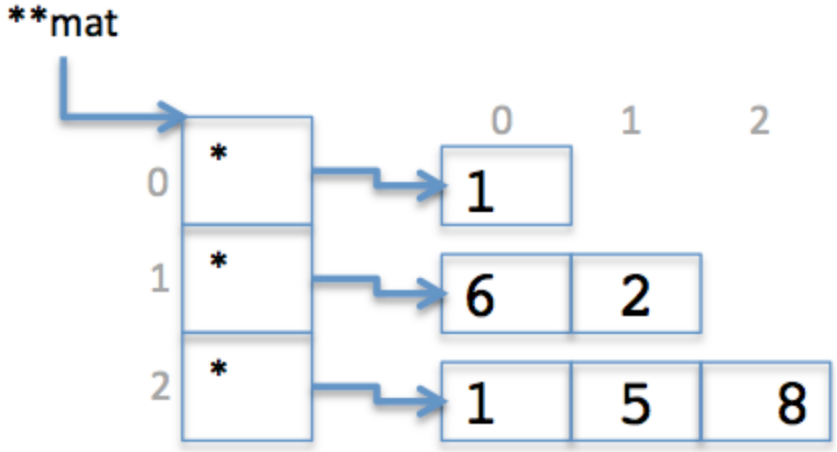
Matriz triangular

Definição (Matriz Triangular)

Matrizes triangulares são matrizes quadradas, onde os elementos acima (ou abaixo) da diagonal principal são nulos. Podem ser classificadas em triangular superior ou inferior, de acordo com a posição desses elementos em relação à diagonal principal.



(a) Visualização como matriz.



(b) Visualização como ponteiro.

Arranjos multidimensionais

[matriz_triang_inf.c]

Exemplo 6 (Alocação de matriz triangular inferior)

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

/*Protótipo*/
int soma(int **mat, int tam);

int main(int argc, char *argv[]) {
   int **mat;
   int tam, i, j;
   scanf("%d", &tam);

   mat = (int**) malloc(tam * sizeof(int*));
   if (!mat) {
      printf("Memória insuficiente.\n");
      return -1;
   }

   for (i = 0; i < tam; i++) {
      mat[i] = (int*) malloc((i+1) * sizeof(int));

      if (!mat[i]) {
         printf("Memória insuficiente.\n");
         return -1;
      }
   }

   for (i = 0; i < tam; i++) {
      for(j = 0; j < (i + 1); j++) {
         scanf("%d", &mat[i][j]);
      }
   }
   printf("Soma dos Elementos: %d\n",
      soma(mat, tam));

   for (i = 0; i < tam; i++) {
      free(mat[i]);
   }
   free(mat);
   return 0;
}

int soma(int **mat, int tam) {
   int i, j, soma = 0;
   for (i = 0; i < tam; i++){
      for (j = 0; j < (i + 1); j++) {
         soma += mat[i][j];
      }
   }
   return soma;
}
```

Exercícios de Aprendizagem

Exercício 1

Crie uma agenda de contatos.
Solicite o número de contatos a serem informados.
Para cada contato solicite: Nome, End. e Fone.
Baixe o template do site.

Exemplo de Funcionamento

```
Informe o nr de contatos: n = 1

Contato Nr 1
Nome: Paulo Joia
End.: Rua Oratório, s/n
Fone: 11 3370-7070

Lista de contatos:
  1: Paulo Joia;Rua Oratório, s/n;11 3370-7070
```

Considerações Finais

- Nesta aula foram apresentados os principais conceitos relacionados a:
  - Alocação dinâmica de memória
  - Alocação dinâmica de arranjos multidimensionais

✓ *É importante rever os conceitos apresentados na aula e consultar a bibliografia sugerida sobre o assunto.*

Referências Bibliográficas I

Aguilar, L. J. (2008).

Programação em C++: Algoritmos, Estruturas de Dados e Objetos.
McGraw-Hill, São Paulo.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2002).

Algoritmos: Teoria e Prática.
Elsevier, Rio de Janeiro.

Drozdek, A. (2009).

Estrutura de Dados e Algoritmos em C++.
Cengage Learning, São Paulo.

Forbellone, A. L. V. e Eberspacher, H. F. (2005).

Lógica de Programação: A Construção de Algoritmos e Estrutura de Dados.
Pearson Prentice Hall, São Paulo, 3 edition.

Knuth, D. E. (2005).

The Art of Computer Programming.
Addison-Wesley, Upper Saddle River, NJ, USA.

Pinheiro, F. d. A. C. (2012).

Elementos de Programação em C.
Bookman, Porto Alegre.

Referências Bibliográficas II

Sedgewick, R. (1998).

Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, Searching.
Addison-Wesley, Boston, 3rd edition.

Szwarcfiter, J. L. e Markenzon, L. (1994).

Estruturas de Dados e Seus Algoritmos.
LTC, Rio de Janeiro.

Tenenbaum, A. A., Langsam, Y., e Augenstein, M. J. (1995).

Estruturas de Dados Usando C.
Makron Books, São Paulo.

Terra, R. (2014).

Linguagem C - Notas de Aula.
Universidade Federal de Lavras (UFLA).
Disponível em:
<http://professores.dcc.ufla.br/~terra/public_files/2014_apostila_c_ansi.pdf>. Acesso em: 2018-09-16.