

# Alocação Dinâmica de Memória

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Alocação Estática
- Alocação Dinâmica
  - *malloc*
  - *calloc*
  - *realloc*
  - *free*
  - Alocação em Funções
- Alocação Estática de Matrizes
- Alocação Dinâmica de Matrizes
- Erros Comuns em Alocação Dinâmica

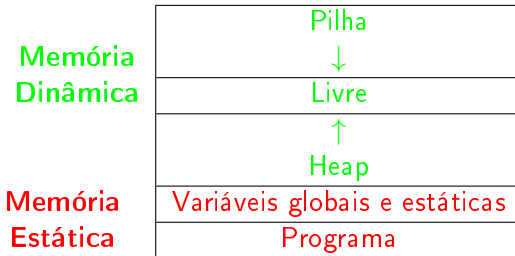
- O compilador reserva espaço na memória principal para todos os dados declarados explicitamente
  - Armazenamento em bytes

```
char c; // 1 byte
int i; // 4 bytes
char s[10]; // 10 bytes (1 * 10)
int v[20]; // 80 bytes (4 * 20)
float f; // 4 bytes
double s; // 8 bytes
```

- A subdivisão da memória ocorre para evitar conflitos durante a execução
  - Programas, variáveis globais e estáticas
  - Pilha
  - Heap

# Introdução

- Memória principal



## Alocação Estática

# Alocação Estática

- O espaço de memória para as variáveis é reservado no início da execução
- Variáveis locais alocadas estaticamente não podem ser alteradas após a execução de uma função
- A quantidade total de memória utilizada pelos dados é previamente conhecida, mas não pode ser alterada
  - Exemplo:  

```
int v[15];
```
- A forma mais natural de armazenar um conjunto de elementos é a alocação sequencial

- Variáveis globais (e estáticas):
  - Espaço reservado existe enquanto o programa estiver executando
- Variáveis locais:
  - Espaço reservado existe enquanto a função estiver sendo executada
  - Espaço liberado ao final da execução (Pilha)
- Arranjos alocados de forma estática

# Alocação Estática

- Vantagens da alocação estática
  - Simplicidade
  - Fácil organização
  - O acesso aos dados é imediato e simples
- Desvantagens:
  - Análise de memória necessária durante programação
  - Memória não pode ser redimensionada



## Alocação Dinâmica

- Espaço de memória requisitado em tempo de execução
- Utiliza funções de sistema para alocar e liberar espaços no espaço de memória *heap*
  - Espaços alocados e não liberados são desalocados apenas no final da execução do programa
- Vantagens:
  - Minimiza o desperdício de recursos
  - Possibilita o redimensionamento do espaço de memória alocado
  - Otimiza o gerenciamento de memória no sistema

- A memória alocada dinamicamente é acessada através de Apontadores
  - Variáveis que armazenam o endereço de uma área de memória

```
int *p, *q, x;
```

```
p = &x;
```

```
*q = 2; // não pode, pois não há um endereço associado  
ao ponteiro q
```

- No exemplo acima, os ponteiros são nulos (NULL) quando declarados
  - Como fazemos para vincular o ponteiro a um novo endereço de memória em vez de um já existente, como no caso da variável *x*?

- Funções da biblioteca padrão "stdlib.h":
  - *malloc*
  - *calloc*
  - *realloc*
  - *free*

- ***void\* malloc(size\_t x)***: responsável por alocar um tamanho *x* (em bytes) de memória, e retornar um ponteiro para o endereço base de memória
- Exemplo

```
int main(void){  
    int *p;  
    int amount = 15;  
  
    p = (int*) malloc(amount * sizeof(int));  
  
    return 0;  
}
```

- O *malloc* pode ser usado sem o *typecasting* (no exemplo acima é dado por *(int\*)*), mas pode acarretar em *warnings* durante a compilação

- Exemplo 2

```
int main(void){  
    int *p2;  
    p2 = (int*) malloc(sizeof(int));  
    *p2 = 5;  
  
    return 0;  
}
```

- E se a função *malloc* retornar NULL?
  - Como essa situação poderia ocorrer?

- Exemplo 2

```
int main(void){
    int *p2;
    p2 = (int*) malloc(sizeof(int));

    if (p2 == NULL){
        printf("Falha ao alocar. Espaço insuficiente!");
        exit(1);
    }

    *p2 = 5;

    return 0;
}
```

- ***void\* calloc(size\_t x, size\_t y)***: aloca *x* vezes o tamanho *y*, devolvendo um ponteiro para o endereço base da região alocada
- Inicializa o conteúdo da memória com valor zero
- Exemplo

```
int main(void){  
    int *p;  
    int amount = 15;  
  
    p = (int*) calloc(amount, sizeof(int));  
  
    return 0;  
}
```



- *Calloc* faz a mesma coisa que *malloc*
  - Diferença: *calloc* zera todo o espaço alocado
  - Isso significa que a função *calloc* é mais custosa em comparação à *malloc*
  - Por outro lado, após o uso do comando *malloc*, na memória alocada poderá ter lixo
  - Mesmo assim, *malloc* é mais utilizado em comparação com o *calloc*

- ***void\* realloc(void\* ptr, size\_t x)***: modifica o tamanho de memória já alocada
  - Altera o tamanho da memória apontada pelo ponteiro *ptr* para *x* bytes
  - Não há perda do conteúdo da faixa de memória inicial
  - O conteúdo do endereço extra é indefinido
  - Caso o compilador não encontre memória suficiente para fazer realocação, os endereços permanecem inalterados e a função retorna um ponteiro NULL

- Exemplo

```
int main(void){
    char *s;

    //armazenar 12 caracteres e \0
    s = (char*) malloc(13 * sizeof(char));

    //colocando uma frase na string
    strcpy(s, "kame hame ha");

    //realocar espaço para mais um caractere
    s = (char*) realloc(s, 14 * sizeof(char));

    //adicionando o ponto de exclamação à string
    strcat(s, "!");

    return 0;
}
```

- ***void free(void\* ptr)***: devolve ao *heap* a memória apontada por *ptr*
  - Aceita apenas ponteiros alocados dinamicamente
  - Deve ser usada para liberar blocos de memória inteiros e não partes do bloco

```
int main(void) {  
    int *p;  
    int amount = 15;  
  
    p = (int*) malloc(amount * sizeof(int));  
  
    free(p);  
  
    return 0;  
}
```

# Alocação Dinâmica

## Alocação em Funções

- Declaração estática: perde referência ao fim da execução
- Declaração dinâmica: disponível para acesso após fim da execução

- Exemplo de implementação com problemas:

```
int* func(int *v){  
    int p[2];  
  
    p[0] = v[0] * v[0];  
    p[1] = v[1] * v[1];  
  
    return p;  
}
```

- Ao compilar essa função, aparecerá um alerta informando que o endereço de uma variável local é retornado
- Em outras palavras, o vetor resultante não pode ser acessado após a execução da função

- Correção da implementação anterior:

```
int* func(int *v){  
    int *p = (int*) malloc(2 * sizeof(int));  
  
    p[0] = v[0] * v[0];  
    p[1] = v[1] * v[1];  
  
    return p;  
}
```

- Exemplo 2:

```
typedef struct{
    int item[100];
    int quantidade;
}Vetor;

Vetor* iniciar_vetor(){
    int i;
    Vetor* v = (Vetor*) malloc(sizeof(Vetor)); // e se
    usarmos calloc?

    v->quantidade = 0;

    for (i = 0; i < 100; i++)
        v->item[i] = 0;

    return v;
}
```



- Exemplo 3:

```
typedef struct{
    int n;
    int *item;
}Vetor;

Vetor* iniciar_vetor(int n){
    Vetor* v = (Vetor*) malloc(sizeof(Vetor));

    v->n = n;
    v->item = (int*) malloc(sizeof(int) * n);

    return v;
}
```

## Alocação Estática de Matrizes

# Alocação Estática de Matrizes

- Conjunto de vários vetores
- Organizadas por linhas e colunas
- Todos os dados estão dispostos sequencialmente na memória principal

```
int rows = 5;  
int cols = 5;
```

```
int mat[rows][cols];
```

# Alocação Estática de Matrizes

```
int rows = 4;  
int cols = 4;  
  
int mat[rows][cols];
```

Memória

mat	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0x300																

# Alocação Estática de Matrizes

```
int rows = 4;  
int cols = 4;  
  
int mat[rows][cols];
```

Memória

<b>mat</b>	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
<b>0x300</b>																



<b>mat[0]</b>
<b>0x300</b>

# Alocação Estática de Matrizes

```
int rows = 4;  
int cols = 4;  
  
int mat[rows][cols];
```

Memória

mat	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0x300																



mat[1]
0x316

# Alocação Estática de Matrizes

```
int rows = 4;  
int cols = 4;  
  
int mat[rows][cols];
```

Memória

<b>mat</b>	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
<b>0x300</b>																



<b>mat[2]</b>
<b>0x332</b>

# Alocação Estática de Matrizes

```
int rows = 4;  
int cols = 4;  
  
int mat[rows][cols];
```

Memória

<b>mat</b>	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
<b>0x300</b>																



<b>mat[3]</b>
<b>0x348</b>



## Alocação Dinâmica de Matrizes

- Função para alocar uma matriz quadrada  $n \times n$

```
int** create_matrix(int n){
    int i;
    int **mat;

    mat = (int**) malloc(n * sizeof(int*));

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

    return mat;
}
```

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

- Uso de ponteiro de ponteiro: tipo de variável que armazena o endereço de outro ponteiro

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

- Um vetor de ponteiros é alocado
  - Cada ponteiro representará uma linha

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

- Um vetor para cada linha da matriz é alocado
  - Cada ponteiro recebe um vetor alocado dinamicamente

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){
    int i;
    int **mat;

    mat = (int**) malloc(n * sizeof(int*));

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

    return mat;
}
```

- Um ponteiro de ponteiro é retornado
  - Ao ser alocado na *heap*, a matriz continuará existindo após finalizar a função
  - A matriz só será liberada quando o programa finalizar ou ser dado um comando *free(mat)*

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

Memória

0x001	n	
-------	---	--

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

Memória

0x001	n	4
-------	---	---



# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	
0x125	mat	

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	
0x125	mat	0x318

0x318	
0	
1	
2	
3	

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	0
0x125	mat	0x318

0x318	
0	0x224
1	
2	
3	

0x224	0	1	2	3

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	1
0x125	mat	0x318

0x318	
0	0x224
1	0x232
2	
3	

0x232	0	1	2	3

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	2
0x125	mat	0x318

0x318	
0	0x224
1	0x232
2	0x240
3	

0x240	0	1	2	3

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```

## Memória

0x001	n	4
0x005	i	3
0x125	mat	0x318

0x318	
0	0x224
1	0x232
2	0x240
3	0x248

0x248	0	1	2	3

# Alocação Dinâmica de Matrizes

```
int** create_matrix(int n){  
    int i;  
    int **mat;  
  
    mat = (int**) malloc(n * sizeof(int*));  
  
    for (i = 0; i < n; i++)  
        mat[i] = (int*) malloc(n * sizeof(int));  
  
    return mat;  
}
```



# Alocação Dinâmica de Matrizes

- Vetores são estruturas alocadas sempre sequencialmente na memória
- Com a alocação dinâmica de matrizes é possível:
  - Alocar matrizes maiores
  - Alocar matrizes onde cada linha possui um tamanho diferente, otimizando espaço durante a execução

## Erros Comuns em Alocação Dinâmica

# Erros Comuns em Alocação Dinâmica

```
int main(){  
    int v = malloc(sizeof(int));  
    free(v);  
    return 0;  
}
```

- As funções de alocação dinâmica devem ser utilizadas apenas para ponteiros
- Solução:

```
int main(){  
    int *v = malloc(sizeof(int));  
    free(v);  
    return 0;  
}
```

# Erros Comuns em Alocação Dinâmica

```
int main(){  
    int n = 10;  
    int *v = malloc(sizeof(int) * n);  
    *v = malloc(sizeof(int) * n);  
    free(v);  
    return 0;  
}
```

- O ponteiro `v` é alocado duas vezes, mas a primeira alocação é perdida
- Solução: usar um ponteiro para cada alocação ou liberar o espaço referente à primeira alocação

```
int main(){  
    int n = 10;  
    int *v = malloc(sizeof(int) * n);  
    int *v2 = malloc(sizeof(int) * n);  
    free(v);  
    free(v2);  
    return 0;  
}
```

# Erros Comuns em Alocação Dinâmica

```
int main(){  
    int *v = malloc(2);  
    free(v);  
    return 0;  
}
```

- Alocação de espaço insuficiente para o tipo de dado
- Solução: usar *sizeof* (caso seja um vetor, multiplicar o resultado da função pela quantidade de itens) ou colocar o tamanho exato do espaço necessário

```
int main(){  
    int *v = malloc(sizeof(int));  
    free(v);  
    return 0;  
}
```

# Erros Comuns em Alocação Dinâmica

```
int main(){  
    int n = 10;  
    int *v = malloc(sizeof(int) * n);  
    free(*v);  
    return 0;  
}
```

- Utilizar um valor em vez de um ponteiro alocado dinamicamente
- Solução: utilizar ponteiro alocado dinamicamente

```
int main(){  
    int n = 10;  
    int *v = malloc(sizeof(int) * n);  
    free(v);  
    return 0;  
}
```

# Erros Comuns em Alocação Dinâmica

```
int main(){  
    int n = 10;  
    int *p = &n;  
    free(p);  
    return 0;  
}
```

- Liberar ponteiro não alocado dinamicamente na função *free*
- Solução: Não usar a função *free* ou usar ponteiro alocado dinamicamente

```
int main(){  
    int n = 10;  
    int *p = malloc(sizeof(n));  
    *p = n;  
    free(p);  
    return 0;  
}
```

# Erros Comuns em Alocação Dinâmica

```
void imprimir_e_liberar(int *v, int n){  
    int i;  
    free(v);  
    for (i = 0; i < n; i++)  
        printf("%d\n", v[i]);  
}
```

- Liberar ponteiro antes do seu uso
- Solução: liberar o ponteiro após o seu uso

```
void imprimir_e_liberar(int *v, int n){  
    int i;  
    for (i = 0; i < n; i++)  
        printf("%d\n", v[i]);  
    free(v);  
}
```





Arakaki, R.; Arakaki, J.; Angerami, P. M.; Aoki, O. L.; Salles, D. S.

*Fundamentos de programação C: técnicas e aplicações.*  
LTC, 1990.



Deitel, H. M.; Deitel, P. J.

*Como programar em C.*  
LTC, 1999.



Pereira, S. L.

*Estrutura de Dados e em C: uma abordagem didática.*  
Saraiva, 2016.



Tenenbaum, A.; Langsam, Y.

*Estruturas de Dados usando C.*  
Pearson, 1995.