Algoritmos e estruturas de Dados A06

Alocação Dinâmica de Memória

Prof. Maiquel de Brito

Gestão de memória

Em C a memória pode ser gerida de forma:

- Estática – variáveis são criadas e inicializadas somente uma vez. **Persistem** para todo o ciclo de vida do programa

```
static int i = 0;
```

- Automática – variáveis são criadas e eliminadas quando as funções são chamadas; são variáveis válidas num contexto (ou escopo) também designadas de **variáveis locais**

```
int i = 0;
```

- Dinâmica – variáveis são geridas explicitamente (através de alocação dinâmica)

Exemplo Variáveis Automáticas

```c
#include <stdio.h>

int inc(){
  int cont=0;
  cont++;
  return  cont;
}

int main(void){
  int i;
  inc();
  inc();
  i = inc();
  printf("%d\n", i);
}
```

Exemplo Variáveis Automáticas

```c
#include <stdio.h>

int inc(){
  int cont=0;
  cont++;
  return  cont;
}

int main(void){
  int i;
  inc();
  inc();
  i = inc();
  printf("%d\n", i);
}
```

O valor impresso de `i` é 1.

A variável `cont` tem seu valor ajustado para zero a cada chamada da função `inc`

Exemplo Variável Estática (cont)

```c
#include <stdio.h>

int inc(){
  static int cont=0;
  cont++;
  return  cont;
}

int main(void){
  int i;
  inc();
  inc();
  i= inc();
  printf("%d\n", i);
}
```

Exemplo Variável Estática (cont)

```c
#include <stdio.h>

int inc(){
  static int cont=0;
  cont++;
  return  cont;
}

int main(void){
  int i;
  inc();
  inc();
  i= inc();
  printf("%d\n", i);
}
```

O valor impresso de `i` é 3.

A variável `cont` tem seu valor ajustado para zero na primeira chamada à função `inc`, mantendo seu valor nas execuções seguintes.

Variáveis Locais

- Dimensão fixa
- Ocupam uma área de memória invariável

Então, como criar e destruir variáveis, aumentar e diminuir a sua dimensão, durante a execução do programa?

As linguagens de programação permitem definir **variáveis dinâmicas** à custa de rotinas que reservam espaço para variáveis durante a execução de um programa e o liberam quando as variáveis já não são necessárias.

Variáveis dinâmicas

Gestão de memória dinâmica em C é feita através de um conjunto de funções na biblioteca stdlib.h:

- `malloc/calloc`
- `free`
- `realloc`

Atribuição do espaço de memória durante a alocação de variáveis é feita por um algoritmo específico

Diferentes tipos de dados – diferentes tamanhos

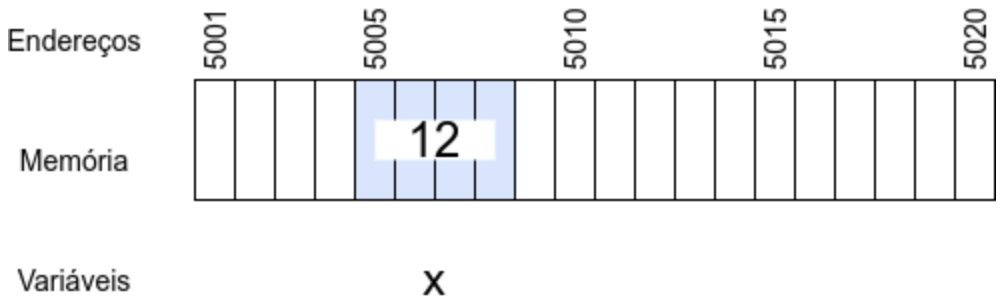
Diferentes tipos de dados – diferentes tamanhos

int – 4 bytes

```
...
int   x = 12;



...
```

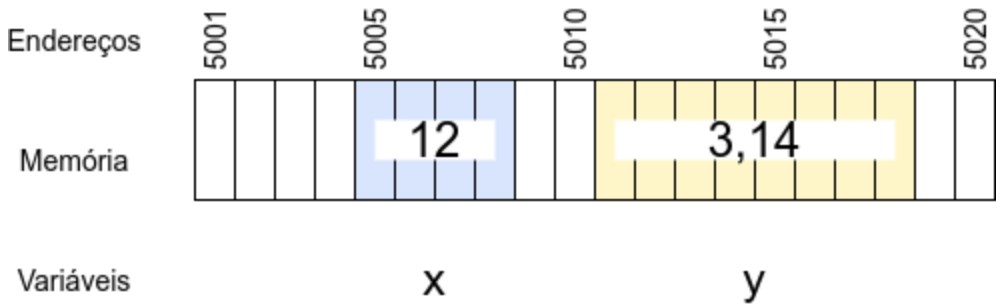


Diferentes tipos de dados – diferentes tamanhos

int – 4 bytes
double – 8 bytes

```
...
int    x = 12;
double y = 3.14;


...
```



Função Malloc

```
void *malloc(int size)
```

- Aloca um bloco de bytes consecutivos na memória do computador
- O conteúdo da memória NÃO é inicializado
- Argumento: n° de bytes
- Retorno: endereço desse bloco de memória alocado

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.    p = malloc(sizeof(int));
7.    *p = 10;
8.    printf("%d\n", *p);
9. }
```

Na linha 5, a variável *p* não “sabe” que vai apontar para um *int*. Só sabe-se que vai apontar para um espaço de memória com o tamanho de um *int*.

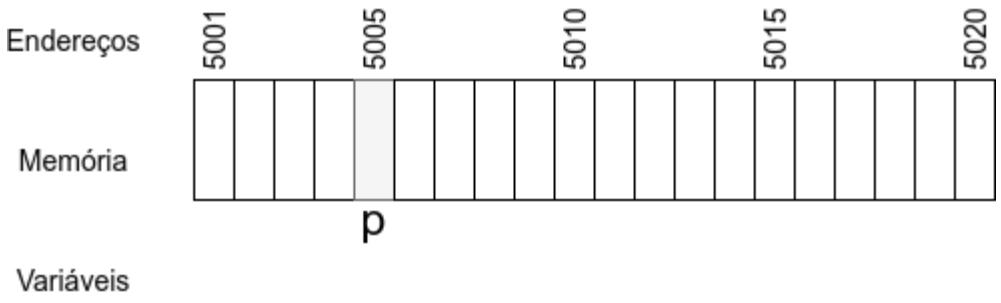
Função Malloc – exemplo (slide 1/5)

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.
6.
7.
8.
9. }
```

Função Malloc – exemplo (slide 2/5)

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.
7.
8.
9. }
```

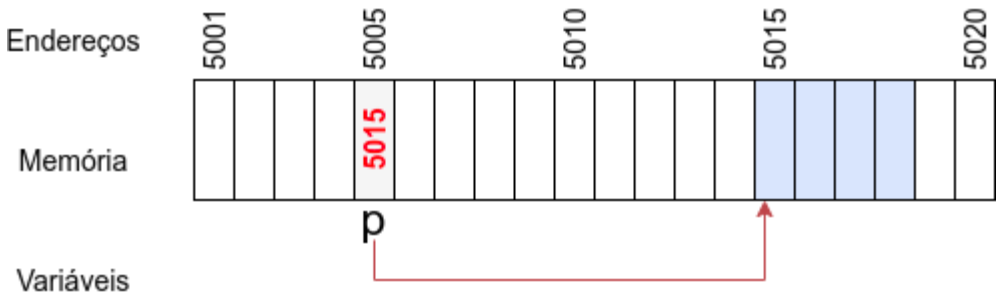
cria um ponteiro p sem atribuir-lhe valor algum



Função Malloc – exemplo (slide 3/5)

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.    p = malloc(sizeof(int));
7.
8.
9. }
```

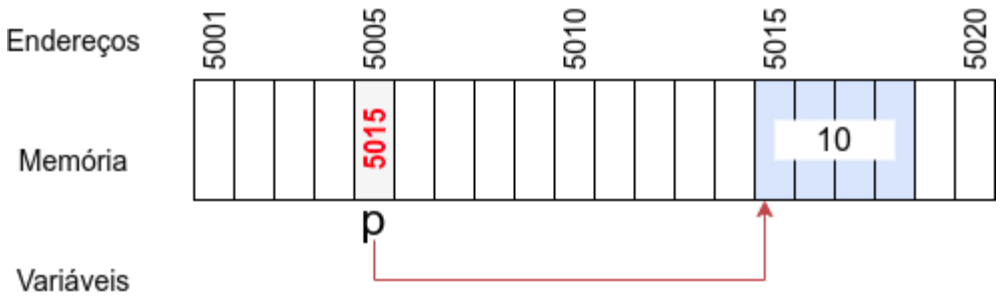
aloca um bloco de 4 bytes (4 = sizeof(int))
atribui primeiro endereço do bloco a `p`



Função Malloc – exemplo (slide 4/5)

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.    p = malloc(sizeof(int));
7.    *p = 10;
8.
9. }
```

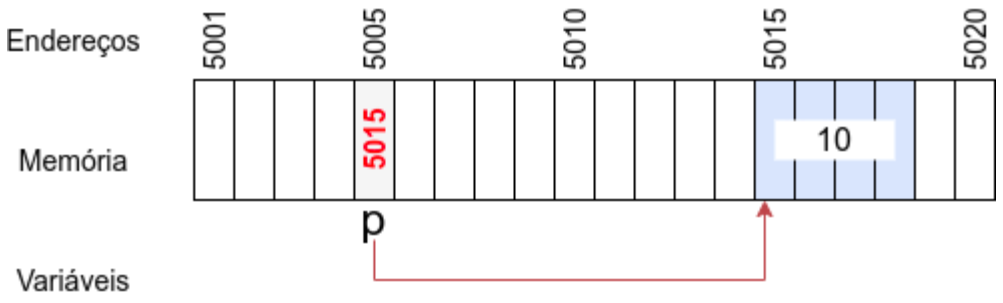
atribui `10` à área de memória para onde `p` aponta



Função Malloc – exemplo (slide 5/5)

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.    p = malloc(sizeof(int));
7.    *p = 10;
8.    printf("%d\n", *p);
9. }
```

o ponteiro p é manipulado normalmente (nesse caso, imprime-se o conteúdo do endereço para onde ele aponta.



Função Malloc

Na linha 5, a variável *p* não "sabe" que vai apontar para um *int*. Só sabe-se que vai apontar para um espaço de memória com o tamanho de um *int*.

Exercício: Executar o código ao lado e analisar o que acontece.

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.    //p = malloc(sizeof(int));
7.    *p = 10;
8.    printf("%d\n", *p);
9. }
```

Na linha 6, *p* passa a apontar para um determinado endereço, que é retornado pela função *malloc*. A função *malloc* executa duas tarefas: (i) faz a alocação de memória com o tamanho indicado e (ii) retorna um ponteiro para o espaço de memória alocado.

Exercício: Executar o código ao lado e comparar com o que aconteceu no exercício anterior.

```
1 #include <stdio.h>
2.#include <stdlib.h>
3.
4. int main(){
5.    int *p;
6.    p = malloc(sizeof(int));
7.    *p = 10;
8.    printf("%d\n", *p);
9. }
```

Função Malloc

Exercício:

Executar o programa abaixo e analisar o que é impresso.

OBS: `printf("%p",x)` imprime o endereço do ponteiro x.

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

int main(){
    int *p;
    p = malloc(sizeof(int));
    printf("%p - %d\n", p, *p);
    *p = 10;
    printf("%p - %d\n", p, *p);
}
```

Vetores, ponteiros e Alocação Dinâmica

- Na linguagem C, o nome de um vetor corresponde ao endereço do seu primeiro elemento, isto é, se v é um vetor v == &v[0]

- Existem 2 formas de colocar um ponteiro apontando para o primeiro elemento de um vetor:

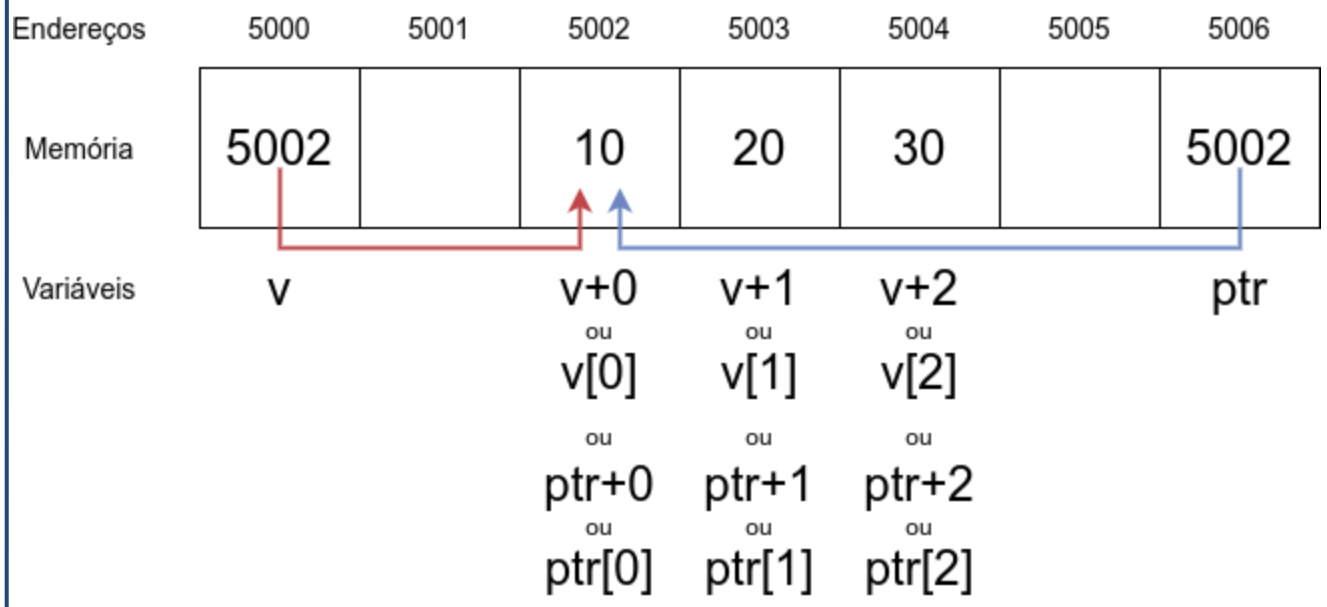
```
int v[3] = {10,20,30};
int *ptr;

ptr = &v[0]; // primeira forma
ptr = v;     // segunda forma
```

Vetores, ponteiros e Alocação Dinâmica

```
int v[3] = {10,20,30};
int *ptr;

ptr = &v[0]; // primeira forma
ptr = v;     // segunda forma
```



Exemplo

Uso de um vetor:
- Alocação automática
- Forma tradicional de acesso aos elementos

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

int main(){
  int i, p[3];
  for(i=0;i<3;i++){
    p[i] = i;
  }
  for(i=0;i<3;i++){
    printf("%d ", p[i]);
  }
}
```

Uso de um vetor:
- Alocação dinâmica
- Forma tradicional de acesso aos elementos

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

int main(){
  int i, *p;
  p = malloc(sizeof(int)*3);
  for(i=0;i<3;i++){
    p[i] = i;
  }
  for(i=0;i<3;i++){
    printf("%d ", p[i]);
  }
}
```

Exemplo

Uso de um vetor:
- Alocação automática
- Forma tradicional de acesso aos elementos

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

int main(){
  int i, p[3];
  for(i=0;i<3;i++){
    p[i] = i;
  }
  for(i=0;i<3;i++){
    printf("%d ", p[i]);
  }
}
```

Uso de um vetor:
- Alocação dinâmica
- Forma tradicional de acesso aos elementos

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

int main(){
  int i, *p;
  p = malloc(sizeof(int)*3);
  for(i=0;i<3;i++){
    p[i] = i;
  }
  for(i=0;i<3;i++){
    printf("%d ", p[i]);
  }
}
```

Exemplo

Uso de um vetor:
- Alocação automática
- Acesso aos elementos usando aritmética de ponteiros

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

int main(){
  int i, p[3];
  for(i=0;i<3;i++){
    *(p+i) = i;
  }
  for(i=0;i<3;i++){
    printf("%d ", *(p+i));
  }
}
```

Uso de um vetor:
- Alocação dinâmica
- Acesso aos elementos usando aritmética de ponteiros

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

int main(){
  int i, *p;
  p = malloc(sizeof(int)*3);
  for(i=0;i<3;i++){
    *(p+i) = i;
  }
  for(i=0;i<3;i++){
    printf("%d ", *(p+i));
  }
}
```

Função Calloc

- Aloca um bloco de bytes consecutivos na memória do computador
- Inicializa cada elemento com o valor '0'

- Argumentos: n° de elem., tamanho de cada elem.
- Retorno: endereço desse bloco de memória alocado

```
void *calloc(int nelements, int size)
```

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

int main(){
  int i, *p;
  p = calloc(3, sizeof(int));
  p[1] = 5;
  for(i=0;i<3;i++){
    printf("%d ", p[i]);
  }
}
```

Função realloc

- Altera o tamanho do bloco de memória apontado
- Não altera o valor da memória
- Argumentos:
  - ponteiro para o bloco de memória
  - tamanho do novo bloco (em bytes), que pode ser maior ou menor que o original
- Retorno: endereço desse bloco de memória alocado

```
void *realloc(void *p, int new_size)
```

```
int main() {
  char *s;
  s = malloc(80);
  printf("Escreva uma frase:");
  gets(s);
  printf("%s\n", s);
  s = realloc(s, 20);
  strcpy(s, "Bom dia! ");
  printf("%s\n", s);
}
```

A memória é finita

- Se a memória do computador já estiver toda ocupada, malloc e calloc não conseguem alocar mais espaço e devolvem NULL
- Convém verificar essa possibilidade antes de prosseguir

```
/* aloca espaco para array de 10 elementos int */
int *ip = malloc(100 * sizeof(int));

if(ip == NULL) {
  printf("Sem memoria\n");
  exit(EXIT_FAILURE);
}
```

```
/* aloca espaco para array de 10 elementos float*/
float *ptr = calloc(10,sizeof (float));

if (ptr == NULL) {
  printf("Sem memoria\n");
  exit(EXIT_FAILURE);
}
/* alocacao bem sucedida */
```

Função Free

Diferente de variáveis alocadas automaticamente, as variáveis alocadas dinamicamente não são liberadas quando a função que a criou é encerrada.

Variáveis alocadas dinamicamente são liberadas apenas quando o programa é encerrado
(e não quando a função que as criou é finalizada).

Por isso, é importante liberar explicitamente o espaço de memória para o qual os ponteiros apontam usando *free:*

```
void free(void *p)
```

```
...
char *p = (char *) malloc(15);
strcpy(p, "Hello, world!");
free(p);
...
```

Dangling pointers

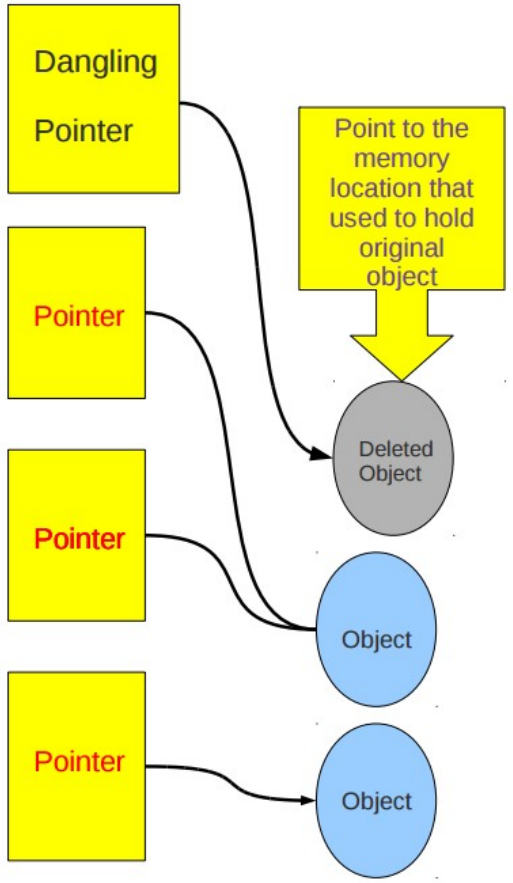
A instrução *free(p)* libera o espaço de memória pra onde *p* aponta para posterior utilização

No entanto, *p* continua apontando para o mesmo espaço de memória após *free(p)*

Novos dados escritos naquele espaço poderão ser acessados por *p.* Diz-se que *p* torna-se um ponteiro "solto" (ou *dangling pointer)*

Por segurança recomenda-se atribuir o valor NULL ao ponteiro após o comando free:

```
free(p);
p = NULL;
```



Vazamento de Memória

- Não é obrigatório liberar a memória quando já não é necessária.
- No fim do programa, a memória pedida ao sistema operativo é automaticamente devolvida.
- Porém pode levar a falhas no sistema se a memória for completamente consumida

Vazamento de Memória

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

int func(void){
    void *s;
    s = malloc(50);
    return;
}

int main(){

  while (1){
    func();
  }
  return 0;
}
```

Este laço infinito chama a função func(), definida acima.
Cedo ou tarde a chamada à função irá falhar, devido a um erro de alocação na função malloc, quando a memória terminar.

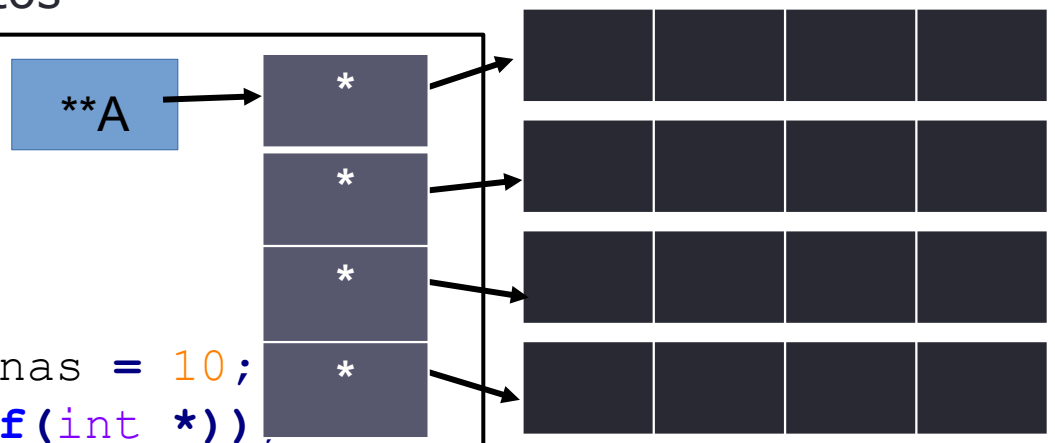
Alocação Dinâmica de Matrizes

- Matriz bidimensional é implementada como um vetor de vetores.
- Uma matriz bidimensional é um vetor ponteiros para linhas e cada linha é um vetor de elementos

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

int main(){
  int **A;
  int i, nlinhas = 10, ncolunas = 10;
  A = malloc(nlinhas * sizeof(int *));
  for (i = 0; i < nlinhas; ++i){
    A[i] = malloc( ncolunas * sizeof(int));
  }

  A[1][2] = 25;
  printf("%d\n",A[1][2]); // ou
  printf("%d\n",*(A[1] + 2));
}
```



Exemplo

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

int** cria_mat(int m, int n){
  int i, **M;
  M = (int**) malloc(m*sizeof(int *));
  for (i = 0; i < m; i++){
    M[i] =
      (int*)calloc(n,sizeof (int));
  }
  return M;
}

void imp_mat(int **M,int m,int n){
  int l, c;
  for (l = 0; l < m; l++){
    for (c = 0; c < n; c++){
      printf("%d ",M[l][c]);
    }
    printf("\n");
  }
}
```

```c
int main(){
  int **A;
  int i;
  A = cria_mat(3,4);
  A[1][1]=5;
  imp_mat(A,3,4);
}
```

Erros Comuns

- Esquecer de alocar memória e tentar acessar o conteúdo da variável
- Copiar o valor do apontador ao invés do valor da variável apontada
- Não inicializar o ponteiro
- Esquecer de liberar memória
- Ela é liberada ao fim do programa, mas pode ser um problema em loops
- Tentar acessar o conteúdo da variável depois da memória ter sido liberada