

# Ponteiros

## Programação de Computadores 1



**INSTITUTO  
FEDERAL**  
Brasília

Prof. Daniel Saad Nogueira  
Nunes

IFB – Instituto Federal de Brasília,  
Campus Taguatinga



# Sumário

---

- 1 Introdução
- 2 Ponteiros
- 3 Alocação dinâmica
- 4 Cuidados



# Sumário

---

## 1 Introdução



# Introdução

---

- Em C, ponteiros são tipos de variáveis que armazenam como valor um endereço de memória.
- Eles são essenciais em algumas linguagens de programação (C) e em outras eles são restritos de alguma forma.



# Sintaxe

---

- Em C, a declaração de um ponteiro para uma região de memória é dada por: `<tipo*> nome_do_ponteiro;`



# Sintaxe

```
1  /* Variável do tipo ponteiro para inteiro.  
2  Tem como valor um endereço ocupado por um inteiro*/  
3  int* ptr;  
4  /* Variável do tipo ponteiro para double.  
5  Tem como valor um endereço ocupado por um double*/  
6  double* ptr;  
7  /* Variável do tipo ponteiro para ponteiro para inteiro.  
8  Tem como valor um endereço ocupado por um ponteiro  
9  para inteiro.*/  
10 int** ptr;  
11 /* O que é isso? */  
12 void* ptr;
```



# Sintaxe

---

- C é uma linguagem em que `void` quer dizer uma coisa e `void*` quer dizer praticamente o oposto.
- `void*`: tipo para um ponteiro que armazena um endereço qualquer.



# Sintaxe

```
1  /* Variável do tipo ponteiro para inteiro.  
2  Tem como valor um endereço ocupado por um inteiro*/  
3  int* ptr;  
4  /* Variável do tipo ponteiro para double.  
5  Tem como valor um endereço ocupado por um double*/  
6  double* ptr;  
7  /* Variável do tipo ponteiro para ponteiro para inteiro.  
8  Tem como valor um endereço ocupado por um ponteiro  
9  para inteiro.*/  
10 int** ptr;  
11 /* Variável do tipo ponteiro para ponteiro para inteiro.  
12 Tem como valor um endereço ocupado por alguma  
13 variável qualquer */
```





# Sintaxe

---

- Existem dois operadores relacionados aos ponteiros: `&` e `*`.
- O operador `&`, ao ser aplicado em uma variável, obtém o endereço daquela variável.
- Já o operador `*` acessa o conteúdo do endereço armazenado pelo ponteiro.



# Exemplo

```
1  #include <stdio.h>
2
3  int main(void) {
4
5      /* Ponteiro para inteiro, contém o valor de
6       * uma posição de memória que é ocupada por um inteiro*/
7      int *ptr;
8      int inteiro = 0;
9
10     /* ptr aponta para o endereço especial NULL*/
11     ptr = NULL;
12
13     /* Atribuímos a ptr, o endereço da variável inteiro */
14     ptr = &inteiro;
15     printf("O valor do ponteiro ptr = %p.\n", ptr);
16     return 0;
17 }
```

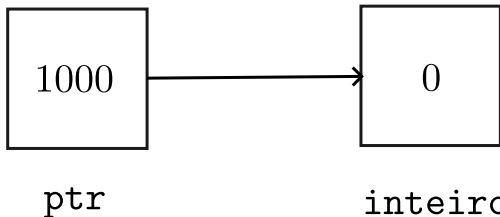


## Exemplo

- No exemplo anterior a variável `ptr` armazenará o endereço da variável `inteiro`.
- O endereço armazenado por `ptr` pode ser impresso na tela com o comando `printf` sob o formato `p`.

endereço 460

endereço 1000





## Exemplo

---

### NULL

É uma boa prática de programação inicializar os ponteiros com **NULL**. Ajuda a detectar possíveis erros na manipulação de ponteiros, visto que, acessar o conteúdo de um ponteiro que aponta para **NULL** é um erro de lógica.



# Ponteiros

---

## Desreferência

- Ponteiros são mecanismos de manipulação indireta de dados.
- Através de um ponteiro, é possível modificar um valor da variável apontada por ele.
- Utilizamos o operador `*` de desreferência.
- Sintaxe: `*nome_do_ponteiro`.



# Exemplo

```
1  #include <stdio.h>
2
3  int main() {
4      /* Ponteiro para inteiro, contém o valor de uma posição
5       de memória que é ocupada por um inteiro */
6      int *ptr;
7      /*um inteiro*/
8      int var = 0;
9      printf("Var = %d\n", var);
10     /*O valor de ptr aponta agora para o endereço de
11     memória que corresponde à variável var. */
12     ptr = &var;
13     /*Modificamos o *conteúdo* da região
14     de memória apontada por ptr*/
15     (*ptr) = 1; // Equivale a fazer var=1
16     /*Note que agora o novo valor de var é 1*/
17     printf("Var = %d\n", var);
18     printf("Var = %d\n", *ptr);
19     return 0;
20 }
```

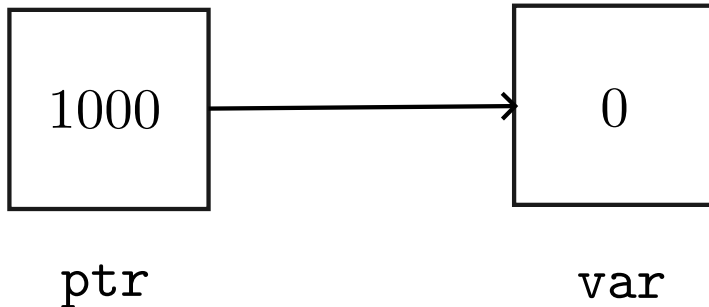


## Desreferência

---

endereço 460

endereço 1000

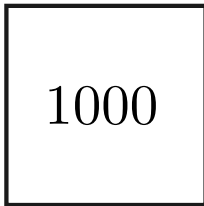




## Desreferência

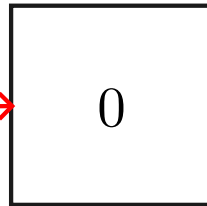
---

endereço 460



ptr

endereço 1000



var



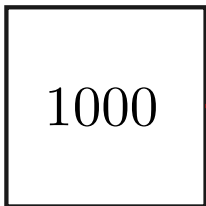




## Desreferência

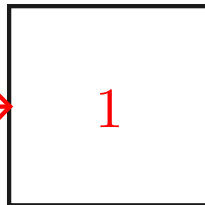
---

endereço 460



`ptr`

endereço 1000



`var`



# Ponteiros

---

- Vamos nos aprofundar agora sobre o que ponteiros podem fazer por nós.



# Sumário

---

## 2 Ponteiros



# Sumário

---

## 2 Ponteiros

- Passagem por referência
- Ponteiros e vetores
- Aritmética de ponteiros



## Passagem por referência

---

- A linguagem C possui dois tipos de passagem de parâmetros para função. Por valor e por referência.
- A afirmação acima está Certa ou Errada?



## Passagem por referência

---

- A linguagem C possui dois tipos de passagem de parâmetros para função. Por valor e por referência.
- A afirmação acima está Certa ou Errada?
- **Errada**. C só possui passagem por valor. A passagem por referência é apenas emulada através de ponteiros.
- A passagem por valor cria uma variável e **copia** o valor da variável passada por parâmetro.
- A emulação de passagem por referência é obtida ao passarmos o **endereço** da variável que queremos modificar. Assim cria-se um novo ponteiro com valor igual a esse endereço. Desta forma, conseguimos manipular a variável com a cópia deste endereço.



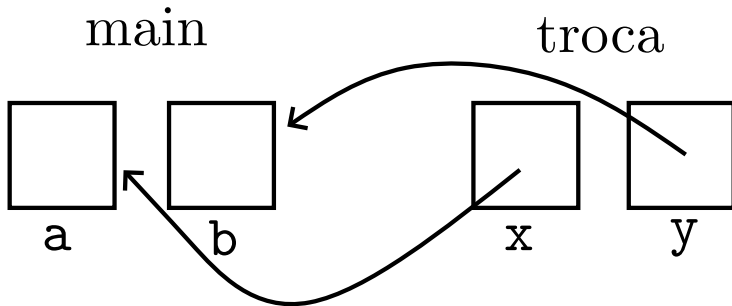
## Passagem por referência

```
1  #include <stdio.h>
2
3  void troca(int *x, int *y) {
4      int aux = *x;
5      *x = *y;
6      *y = aux;
7  }
8
9  int main(void) {
10     int a, b;
11     a = 2;
12     b = 3;
13     troca(&a, &b);
14     printf("a = %d b = %d\n", a, b);
15     return 0;
16 }
```



## Passagem por referência

---







## Passagem por referência

---

- O uso de ponteiros é extremamente recomendável caso desejemos alterar variáveis na função.
- Exemplo: se quisermos criar uma função que retorna o menor e o maior valor de um inteiro, podemos passar duas variáveis através de ponteiros que modificarão as variáveis originais.
- Contorna a limitação das funções em C, que só retornam um único valor.



# Passagem por referência

```
1  #include <stdio.h>
2
3  void min_max(int v[], int n, int *min, int *max) {
4      *min = v[0];
5      *max = v[0];
6      for (int i = 1; i < n; i++) {
7          if (v[i] < *min)
8              *min = v[i];
9          if (v[i] > *max)
10             *max = v[i];
11     }
12 }
13 int main(void) {
14     int v[] = {10, 80, 5, -10, 45, -20, 100, 200, 10};
15     int min, max;
16     min_max(v, 9, &min, &max);
17     printf("%d %d\n", min, max);
18     return 0;
19 }
```



# Sumário

---

## 2 Ponteiros

- Passagem por referência
- Ponteiros e vetores
- Aritmética de ponteiros



# Ponteiros e vetores

---

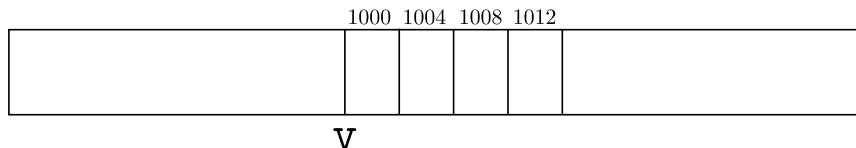
- Quando declaramos um vetor  $v$  de tamanho  $n$ , são alocadas , em uma região da memória,  $n$  posições consecutivas.
- O nome  $v$  corresponde ao endereço base daquela região, isto é, o endereço de início do vetor.



# Ponteiros e vetores

---

```
int v[4];
```





## Ponteiros e vetores

---

- Quando passamos o vetor  $v$  para uma função, não estamos realizando uma cópia do mesmo, mas sim passando apenas o seu endereço base.
- Por isso, seu conteúdo pode ser alterado em uma função. Trata-se do mesmo objeto!
- Note que podemos alterar o endereço de um ponteiro, mas nunca de um vetor.



## Ponteiros e vetores

---

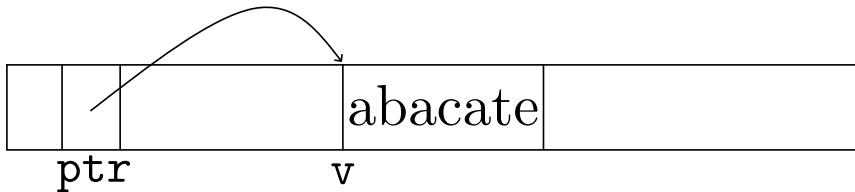
Se um ponteiro aponta para o endereço base do vetor, podemos manipular esse vetor inteiramente,

```
1  #include <stdio.h>
2
3  int main(void) {
4      char *ptr;
5      char v[] = {'a', 'b', 'a', 'c', 'a', 't', 'e', '\0'};
6      ptr = v;
7      printf("String original: %s\n", v);
8      ptr[2] = 'd';
9      printf("String modificada: %s\n", v);
10     return 0;
11 }
```



## Ponteiros e vetores

---







## Ponteiros e vetores

---

- Já que podemos manipular um vetor através de um ponteiro que aponta para o endereço base desse vetor, todas as funções do tipo `f(int v[], int n);` podem ser substituídas pela seguinte sintaxe: `f(int* v, int n);`



## Ponteiros e vetores

---

```
1  #include <stdio.h>
2
3  int soma(int x[], int n) {
4      int acc = 0;
5      for (int i = 0; i < n; i++)
6          acc += x[i];
7      return acc;
8  }
9
10 int main(void) {
11     int x[] = {1, 2, 3, 4, 5, 6};
12     int res = soma(x, 6);
13     printf("Resultado: %d\n", res);
14     return 0;
15 }
```



## Ponteiros e vetores

```
1  #include <stdio.h>
2
3  int soma(int* x, int n) {
4      int acc = 0;
5      for (int i = 0; i < n; i++)
6          acc += x[i];
7      return acc;
8  }
9
10 int main(void) {
11     int x[] = {1, 2, 3, 4, 5, 6};
12     int res = soma(x, 6);
13     printf("Resultado: %d\n", res);
14     return 0;
15 }
```



# Sumário

---

## 2 Ponteiros

- Passagem por referência
- Ponteiros e vetores
- Aritmética de ponteiros



## Aritmética de ponteiros

---

- Em C é possível realizar aritméticas com endereços.
- Somar um valor  $i$  a um endereço  $a$  equivale a ir para  $i$  endereços à direita de  $a$ .
- Subtrair um valor  $i$  a um endereço  $a$  equivale a ir para  $i$  endereços à esquerda de  $a$ .
- O exemplo a seguir implementa uma versão da função `strcat`, que, dadas duas strings, concatena a segunda ao final da primeira.



## Aritmética de ponteiros

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void my_strcat(char *str1, const char *str2) {
5      char *ptr = str1 + strlen(str1);
6      while (*str2 != '\0') {
7          *(ptr++) = *(str2++);
8      }
9      *ptr = '\0';
10 }
11
12 int main(void) {
13     char str1[200] = "abra";
14     char str2[200] = "cadabra";
15     my_strcat(str1, str2);
16     printf("%s\n", str1);
17     return 0;
18 }
```



## Aritmética de ponteiros

---

- O operador `[]`, utilizado para acessar vetores, basicamente consiste em uma aritmética de ponteiro seguido de uma desreferência.
- `ptr[2] == *(ptr+2);`
- O deslocamento é calculado de acordo com o tipo de ponteiro. Se `ptr` é um ponteiro para `char`, então `ptr+2` equivale a somar 2 ao endereço apontado por `ptr`.
- Se `ptr` é um ponteiro para `int`, então `ptr+2` equivale a somar 8 ao endereço apontado por `ptr`, considerando que um inteiro ocupa 4 bytes.



# Sumário

---

## 3 Alocação dinâmica





# Alocação dinâmica

---

- Um recurso extremamente poderoso da linguagem C é a alocação dinâmica de memória.
- Permite requisitar memória em **tempo de execução**.
- Antes de comentar os mecanismos de alocação dinâmica, vamos entender como a memória está organizada.



# Sumário

---

- 3 Alocação dinâmica
  - Organização da memória
  - Chamadas
  - Alocação de matrizes



## Organização da memória

---

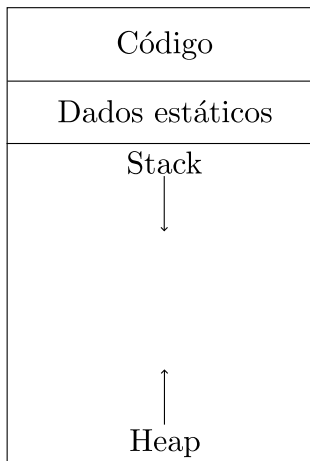
A memória de um programa é dividida em quatro segmentos:

- Código: contém o binário do programa.
- Dados estáticos: contém variáveis globais e estáticas que existem durante toda a execução do programa.
- Pilha (stack): contém as variáveis **locais** que são criadas na execução de uma função e depois são removidas da pilha ao término da função.
- Heap: contém as variáveis criadas por alocação dinâmica.



# Organização da memória

---





## Organização da memória

---

- O segmento de Pilha não é muito grande. É comum que tenha alguns megabytes. Por exemplo, seu valor padrão em sistemas GNU/Linux é de apenas 8MB.
- Isso significa que se você tentar alocar uma quantidade maior do que essa, obterá um estouro da pilha (**stack overflow**), fazendo com que o sistema operacional mate o programa.



## Organização da memória

---

- O programa a seguir lê um inteiro  $n$  de um usuário e cria um vetor estático com base no  $n$  lido, o que é possível a partir do C99 (`int v[n];`).
- Contudo, mesmo essa forma de declaração, está restrito ao tamanho máximo da pilha.
- Se  $n > 2 \cdot 10^6$  existe grande chance do programa quebrar por conta de um estouro de pilha, visto que 2 milhões de inteiros, tipicamente, equivalem a 8MB.



# Organização da memória

---

```
1  #include <stdio.h>
2
3  int main(void) {
4      int n;
5      printf("Digite o valor de n: ");
6      scanf("%d", &n);
7      int v[n];
8      for (int i = 0; i < n; i++)
9          v[i] = i;
10     for (int i = 0; i < n; i++)
11         printf("%d\n", v[i]);
12     return 0;
13 }
```



# Organização da memória

---

- Para contornar essa limitação, recorreremos à **alocação dinâmica de memória**.





# Sumário

---

- 3 Alocação dinâmica
  - Organização da memória
  - Chamadas
  - Alocação de matrizes



## Alocação dinâmica de memória

---

- É possível requisitar alocação de memória ao S.O em tempo de execução.
- Os dados são colocados no segmento de Heap.
- Geralmente através das chamadas `realloc`, `malloc` e `calloc`, presentes sob o cabeçalho `stdlib.h`.
- Estas chamadas retornam ponteiros para as regiões alocadas em caso de sucesso.
- Em caso de falha, **NULL** é retornado.
- Verifiquemos as assinaturas destas funções.



## Alocação dinâmica de memória

---

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

- `size`: tamanho em bytes da região a ser alocada.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



## Exemplo

---

- O exemplo a seguir aloca uma região com  $n$  inteiros, em que  $n$  é lido do teclado, e aloca um vetor de  $n$  elementos.
- Se houver falha na alocação, o programa é encerrado através do comando `exit(EXIT_FAILURE);`.
- Em seguida, o vetor é preenchido com os valores de 0 a  $n - 1$ .
- Para solicitar a quantidade correta de bytes, multiplicamos  $n$  pelo tamanho de um inteiro, o qual pode ser obtido através do operador `sizeof(int);`.
- Ao final, a memória é desalocada com o `free`.



# Exemplo

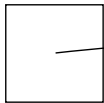
```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int n;
6      scanf("%d", &n);
7      int *ptr = malloc(sizeof(int) * n);
8      if (ptr == NULL) {
9          printf("Erro de alocação.\n");
10         exit(EXIT_FAILURE);
11     }
12     for (int i = 0; i < n; i++)
13         ptr[i] = i;
14     for (int i = 0; i < n; i++)
15         printf("%d\n", ptr[i]);
16     free(ptr);
17     return 0;
18 }
```



## Exemplo

---

```
malloc(sizeof(int)*100);
```



ptr



100 inteiros



## Alocação dinâmica de memória

---

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

- Parecida com `malloc`, mas inicializa a área alocada com zeros.
- Recebe dois parâmetros, em vez de um só.
- `num`: número de elementos.
- `size`: tamanho em bytes de cada elemento.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



# Exemplo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      int n;
6      scanf("%d", &n);
7      int *ptr = calloc(n, sizeof(int));
8      if (ptr == NULL) {
9          printf("Erro de alocação.\n");
10         exit(EXIT_FAILURE);
11     }
12     for (int i = 0; i < n; i++)
13         ptr[i] = i;
14     for (int i = 0; i < n; i++)
15         printf("%d\n", ptr[i]);
16     free(ptr);
17     return 0;
18 }
```

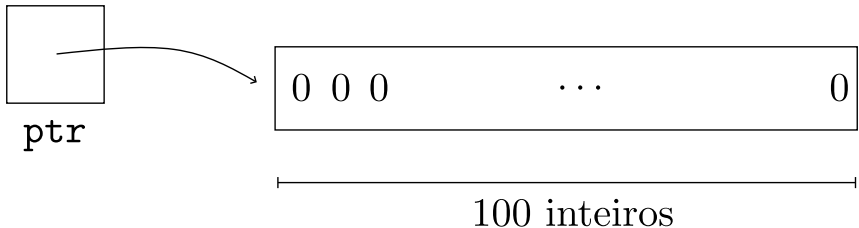




## Exemplo

---

```
calloc(100, sizeof(int));
```





## Alocação inâmica de memória

---

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

- Redimensiona a área alocada.
- `ptr` : ponteiro para região antiga.
- `size` : quantidade em bytes da região a ser realocada.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



## Alocação dinâmica de memória

---

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

- Se `size` for menor que a área antiga, a área é encolhida e o espaço excedente é liberado.
- Se `size` for maior que a área antiga a área é aumentada no número de bytes necessários.
- Se `ptr==NULL`, equivale a uma chamada `malloc`.
- Dependendo, se a área antiga não puder ser expandida devido à falta de espaço contíguo, é alocada uma nova área e o conteúdo antigo é copiado para essa nova área.
- Se `size==0` equivale à liberar a área alocada.



## Alocação dinâmica de memória

---

- Toda área alocada deve ser desalocada após o seu uso.
- Boa prática de programação!
- Utiliza-se a chamada `free`.

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

- `ptr` corresponde à uma região de memória alocada dinamicamente.



# Sumário

---

- 3 Alocação dinâmica
  - Organização da memória
  - Chamadas
  - Alocação de matrizes



## Matrizes dinâmicas

---

- Para criar uma matriz alocada dinamicamente a estratégia é um pouco diferente.
- Primeiro criamos um vetor de ponteiros.
- Cada ponteiro desse vetor, irá apontar para uma linha da matriz alocada.
- É necessário atribuir a cada ponteiro, o endereço do início de cada linha.
- Como trata-se de um vetor de ponteiros, precisamos declarar um **ponteiro para ponteiro**.



# Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa realiza-se a
4     alocação dinâmica de memória de uma matriz.
5     Nele são lidas os número de linhas e colunas
6     e a matriz é preenchida aleatoriamente através
7     da função rand();
8   **/
9
10
11  #include <stdio.h>
12  #include <time.h>
13  #include <stdlib.h>
```



## Exemplo

```
15 int main(void){
16     int l,c;
17     int i,j;
18     srand(time(NULL));
19     printf("Digite o número de linhas da matriz: ");
20     scanf("%d",&l);
21     printf("Digite o número de colunas da matriz: ");
22     scanf("%d",&c);
23     /* Alocamos um vetor de ponteiros */
24     int** m = calloc(l,sizeof(int*));
25     if(m==NULL){
26         printf("Erro na alocação.\n");
27         exit(EXIT_FAILURE);
28     }
```





## Exemplo

```
29  /* O ponteiro zero recebe o espaço da matriz
30   * isto é, l*c */
31  m[0] = calloc(l*c, sizeof(int));
32  if(m[0]==NULL){
33      printf("Erro na alocação.\n");
34      exit(EXIT_FAILURE);
35  }
36  /* Cada um dos ponteiros recebe o início de uma região
37   * de memória apontada por m[0] */
38  for(j=1; j<l; j++){
39      m[j] = m[0]+j*c;
40  }
41  for(i=0; i<l; i++){
42      for(j=0; j<c; j++){
43          m[i][j] = rand() % 1000; /* Um inteiro aleatório [0,999] é g
44      }
45  }
```



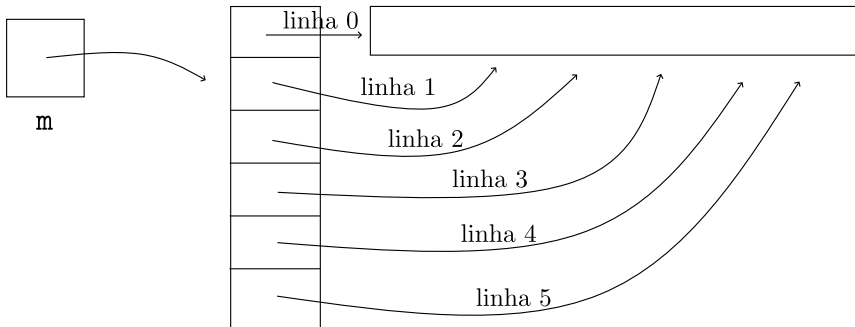
## Exemplo

---

```
46  /* Impressão da matriz */
47  for(i=0;i<l;i++){
48      for(j=0;j<c;j++){
49          printf("%3d ",m[i][j]);
50      }
51      printf("\n");
52  }
53  /* O espaço alocado é liberado */
54  free(m[0]);
55  free(m);
56
57  return 0;
58 }
```



# Matrizes dinâmicas





## Matrizes dinâmicas

---

- Repare que para liberar o espaço alocado, primeiramente deletamos a primeira linha da matriz, para depois liberar o vetor de ponteiros.
- Ordem inversa da alocação.



# Matrizes dinâmicas

---

- Uma função que recebesse essa matriz alocada dinamicamente teria uma assinatura como essa:

```
void f(int** m, int l, int c);
```



# Sumário

---

## 4 Cuidados



# Cuidados

---

- Apesar de serem ferramentas poderosas nas linguagens de programação. Temos que ter cuidado ao manipular ponteiros.
- Erros que ocorrem frequentemente são:
  - ▶ Vazamento de memória (Memory Leak);
  - ▶ Ponteiros Selvagens (Wild Pointers).



# Sumário

---

- 4 Cuidados
  - Memory Leaks
  - Wild Pointers





# Memory Leak

---

- Memory leaks ocorrem quando áreas de memória alocadas não são liberadas quando não são mais necessárias.
- Ao perder a referência para esta área, ela se torna um consumo de memória extra que nunca poderá ser acessada novamente.
- Quando isto ocorre, temos um vazamento de Memória.
- A ferramenta `valgrind` pode ajudar na detecção de vazamentos de memória.



## Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Este problema aborda uma péssima prática
4   * de programação. Os vazamentos de memória (memory leaks).
5   * Estes vazamentos consistem na perda da referência para uma
6   * área alocada, tornando impossível acessar esta área novamente.
7   * O consumo de memória é aumentado desnecessariamente e memory l
8   * são muitas das vezes decorrência de um erro de lógica.
9   * Para detectá-los, podemos usar a ferramenta valgrind.
10  */
11
12  #include <stdlib.h>
13  int main(void) {
```



## Exemplo

---

```
14
15     /* Aloca-se um vetor de 100000 posições */
16     int *ptr = malloc(sizeof(int) * 100000);
17     /* MEMORY LEAK: atribui um novo endereço de memória para
18         * sem desalocar o bloco de memória alocado */
19     ptr = NULL;
20     return (0);
21 }
```



# Valgrind

---

## Vejam os a saída do valgrind:

```
$ valgrind ./memory_leak_exemplo_1
==122784== Memcheck, a memory error detector
==122784== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==122784== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==122784== Command: ./memory_leak_exemplo_1
==122784==
==122784==
==122784== HEAP SUMMARY:
==122784==    in use at exit: 400,000 bytes in 1 blocks
==122784==    total heap usage: 1 allocs, 0 frees, 400,000 bytes allocated
==122784==
==122784== LEAK SUMMARY:
==122784==    definitely lost: 400,000 bytes in 1 blocks
==122784==    indirectly lost: 0 bytes in 0 blocks
==122784==    possibly lost: 0 bytes in 0 blocks
==122784==    still reachable: 0 bytes in 0 blocks
==122784==    suppressed: 0 bytes in 0 blocks
==122784== Rerun with --leak-check=full to see details of leaked memory
==122784==
==122784== For lists of detected and suppressed errors, rerun with: -s
==122784== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```



## Correção do vazamento

---

- Para corrigir o vazamento de memória anterior, precisamos desalocar o espaço alocado antes de atribuir outro valor ao ponteiro.
- Assim, o espaço é liberado antes da referência ser perdida.



## Exemplo

---

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Este programa aborda a correção do exemplo
4   * A memória é liberada antes de trocarmos o valor do ponteiro
5   * Rode ele com o valgrind e compare a diferença de saída e
6   */
7
8  #include <stdlib.h>
```



## Exemplo

---

```
9  int main(void) {
10     /* aloca-se um vetor de 100000 posições.*/
11     int *ptr = malloc(sizeof(int) * 100000);
12     /* libera-se a área de memória alocada */
13     free(ptr);
14     /* agora podemos mudar o valor de ptr */
15     ptr = NULL;
16     return 0;
17 }
```



# Sumário

---

- 4 Cuidados
  - Memory Leaks
  - Wild Pointers





## Wild Pointers

---

- **Wild Pointers, Dangling Pointers** ou **Ponteiros Selvagens** são outro erro de lógica comum na manipulação de ponteiros.
- Corresponde a uma violação de memória que não pertence ao seu programa.
- Geralmente acarreta `Segmentation Faults`, ou falhas de segmentação.
- Ocorrem quando o ponteiro está apontando para uma área inválida da memória que não pertence ao seu programa.
- Geralmente causado pela não atribuição correta dos ponteiros ou pela manipulação deles quando eles não apontam para um endereço válido.



## Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Este programa mostra um exemplo
4   * de um ponteiro selvagem. A variável num só
5   * existe no escopo de func, e portanto, seu endereço
6   * não é mais válido quando a função termina.
7   **/
8
9  #include <stdio.h>
10
11 int *func(void) {
12     int num = 1234;
13     /* ... */
14     return &num;
15 }
```



## Exemplo

---

```
17 int main(void) {  
18     /* A wild pointer has appeared! */  
19     int *ptr = func();  
20     printf("O valor do inteiro num = %d.\n", *ptr);  
21     return 0;  
22 }
```



## Exemplo

---

- O programa anterior falha pois o endereço da variável `num` não é mais válido após o término da função, visto que é uma variável local, e portanto alocada na memória de pilha (stack).
- Uma forma de retornar um endereço válido é fazer com que o endereço retornado seja um endereço presente na memória de (heap), reservada para alocações dinâmicas de memória.



# Exemplo

---

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Este programa corrige o anterior
4   * ao alocar dinamicamente a variável num.
5   * Ao alocarmos dinamicamente, o endereço persiste
6   * até que um free() seja utilizado, logo, o exemplo
7   * abaixo não configura um ponteiro selvagem;
8   */
9
10 #include <stdio.h>
11 #include <stdlib.h>
```



## Exemplo

---

```
13 int *func(void) {
14     int *num = malloc(sizeof(int));
15     if (num == NULL) {
16         printf("Erro de alocação.\n");
17         exit(EXIT_FAILURE);
18     }
19     *num = 1234;
20     return num;
21 }
22
23 int main(void) {
24     int *ptr = func();
25     printf("O valor do inteiro num = %d.\n", *ptr);
26     free(ptr);
27     return 0;
28 }
```