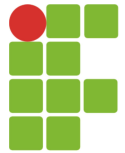
Ponteiros

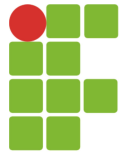
Estruturas de Dados e Algoritmos

Prof. Daniel Saad Nogueira Nunes

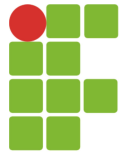
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

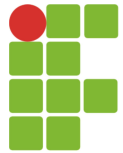


Sumário



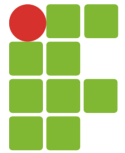
Introdução

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



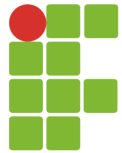
Sintaxe

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



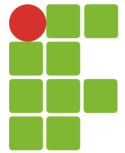
Sintaxe

```
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 para inteiro.*/
9  int** ptr;
10 /* O que é isso? */
11 void* ptr
```



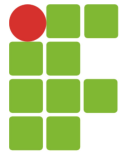
Sintaxe

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



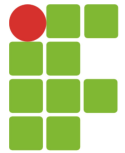
Sintaxe

```
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 para inteiro.*/
9  int** ptr;
10 /* Variável do tipo ponteiro para ponteiro para inteiro.
11 Tem como valor um endereço ocupado por alguma variável qualquer */
12 void* ptr
```



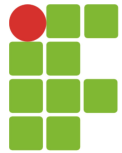
Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa conceitos básicos
4   * sobre ponteiros são explicados.
5   **/
6
7  #include <stdio.h>
8  #include <stdlib.h>
```



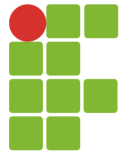
Exemplo

```
9  int main(void) {
10
11     /* Ponteiro para inteiro, contém o valor de
12      * uma posição de memória que é ocupada por um inteiro*/
13     int *ptr;
14     int inteiro;
15
16     /* ptr aponta para o endereço especial NULL**/
17     ptr = NULL;
18
19     /* Atribuímos a ptr, o endereco da variável inteiro */
20     ptr = &inteiro;
21     printf("O valor do ponteiro ptr = %p.\n", ptr);
22     return (0);
23 }
```



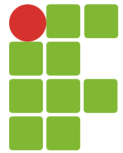
Ponteiros

- 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`.



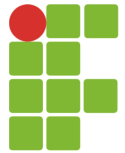
Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Este programa aborda conceitos
4   * básicos sobre ponteiros e deferência.
5   **/
6  #include <stdio.h>
7
```



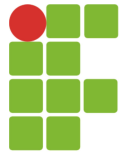
Exemplo

```
8  int main() {
9      /* Ponteiro para inteiro, contém o valor de uma posição
10     de memória que é ocupada por um inteiro */
11     int *ptr;
12     /*um inteiro*/
13     int var = 0;
14     printf("Var = %d\n", var);
15     /*O valor de ptr aponta agora para o endereço de
16      memória que corresponde à variável var. */
17     ptr = &var;
18     /*Modificamos o *conteúdo* da regiao
19     de memoria apontada por ptr*/
20     (*ptr) = 1; // Equivale a fazer var=1
21     /*Note que agora o novo valor de var é 1*/
22     printf("Var = %d\n", var);
23     printf("Var = %d\n", *ptr);
24     return (0);
25 }
```

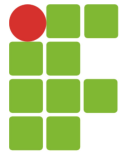


Ponteiros

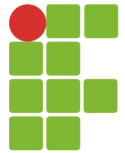
- Vamos nos aprofundar agora sobre o que ponteiros podem fazer por nós.



Sumário

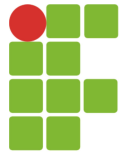


Sumário



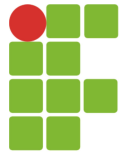
Vetores e ponteiros

- Vetores na verdade se comportam como ponteiros.
- Isto é, vetores são ponteiros para o primeiro elemento do bloco contíguo de memória.
- Baseando-se nisso, podemos utilizar vetores e ponteiros de maneira quase equivalente.
- A única limitação é que um vetor não pode apontar para outra região.
- Vetores são ponteiros constantes!



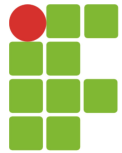
Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa é ilustrado o fato
4   * de que vetores podem ser vistos como ponteiros,
5   * e vice-versa através da aritmética de ponteiros
6   **/
7
8  #include <stdio.h>
9
```



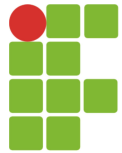
Exemplo

```
10  int main(void) {
11
12      char *ptr;
13      char v[] = {'a', 'b', 'a', 'c', 'a', 't', 'e', '\0'};
14
15      /* O nome de um vetor equivale ao endereço inicial de memória
16       * ocupado por ele. Logo ptr agora aponta para este mesmo
17       * endereco. */
18      ptr = v;
19      printf("String original: %s\n", v);
20      ptr[2] = 'd';
21      printf("String modificada: %s\n", v);
22      return (0);
23  }
```



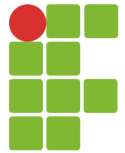
Vetores e ponteiros

- O que significa fazer `ptr[2]` em um ponteiro?
- Significa dizer: amigo, pegue o valor do inteiro duas posições à direita de onde você está no momento.
- Em código: `*(ptr+2)`



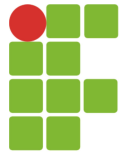
Aritmética de ponteiros

- Em C, é possível somar e subtrair ponteiros.
- Qual o significado?
- `ptr+i` := endereço da posição de memória i posições à direita.
- `ptr-i` := endereço da posição de memória i posições à esquerda.
- O compilador certifica que o deslocamento seja proporcional ao tamanho ocupado pelo tipo.
  - Chars ocupam um byte apenas.
  - Inteiros geralmente 4 bytes.
  - Double geralmente 8 bytes.



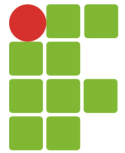
Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa é ilustrado o fato
4   * de que vetores podem ser vistos como ponteiros,
5   * e vice-versa através da aritmética de ponteiros
6   **/
7
8  #include <stdio.h>
9
10 int main(void) {
```

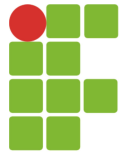


Exemplo

```
11      char s[] = {'a', 'b', 'r', 'a', '\0'};
12      int v[] = {0, 1, 2, 3};
13      char *ptr_s = s;
14      int *ptr_v = v;
15      size_t i;
16      for (i = 0; i < 4; i++) {
17          printf("Endereço de s[%zu] = %p.\n", i, &s[i]);
18          printf("Endereço de v[%zu] = %p.\n", i, &v[i]);
19      }
20      for (i = 0; i < 4; i++) {
21          printf("ptr_s + %zu = %p.\n", i, ptr_s + i);
22          printf("ptr_v + %zu = %p.\n", i, ptr_v + i);
23      }
24      return 0;
25  }
```

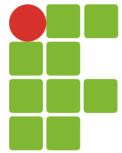


Sumário



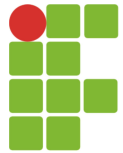
Passagem por referência

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



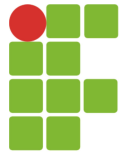
Passagem por referência

- 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 nova 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.



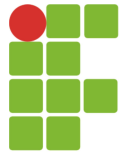
Passagem por referência

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa é explorada
4   * a emulação de passagem por referência em funções na
5   * linguagem C.
6   **/
7
8  #include <stdio.h>
9
10 /* Em C, podemos emular uma passagem por referência através de
11  * ponteiros.
12  * Neste caso, uma cópia do ponteiro que aponta para o endereço de x
13  * é criada. Como a cópia aponta para o endereço de x, podemos modificar
14  * o conteúdo da região de memória apontada por x. **/
15
```

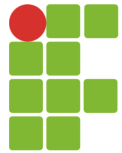


Passagem por referência

```
16  void cubo(double *x) {
17      *x = *x * *x * *x;
18  }
19
20  int main(void) {
21      double a = 3;
22      printf("O cubo de %lf é ", a);
23      cubo(&a);
24      printf("%lf\n", a);
25      return (0);
26  }
```

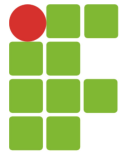


Sumário



Alocação dinâmica de memória

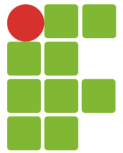
- É possível requisitar alocação de memória ao S.O de maneira dinâmica.
- Geralmente através das chamadas `realloc`, `malloc` e `calloc`.
- Estas chamadas retornam ponteiros para as regiões alocadas em caso de sucesso.
- Verifiquemos as assinaturas destas funções.



Alocação dinâmica de memória

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

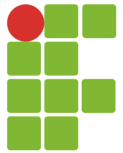
- `size`: tamanho em bytes da região a ser alocada.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



Alocação dinâmica de memória

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

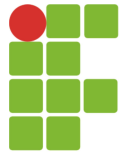
- Parecida com `malloc`, mas inicializa a área alocada com zeros.
- `num`: número de elementos.
- `size`: tamanho em bytes de cada elemento.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



Alocação inâmica de memória

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

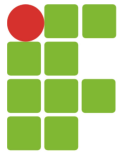
- Redimensiona a área alocada.
- `ptr`: ponteiro para região antiga.
- `size`: quantidade em bytes da região a ser realocada.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



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.

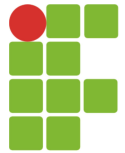


Alocação dinâmica de memória

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

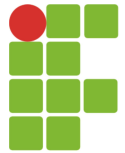
```
void free(void* ptr);
```

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



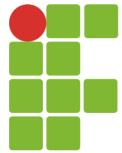
Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa é explorada
4   * a alocação dinâmica de memória através
5   * de ponteiros em C.
6   **/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
```



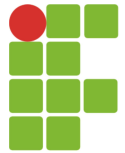
Exemplo

```
11  int main(void) {
12
13      /* Ponteiros inicialmente apontam para uma posição
14       * de memória arbitrária.
15       * É necessário atribuir a eles uma posição de memória válida
16       * que pertença ao seu programa. */
17
18      /* Podemos requisitar ao sistema operacional que ele aloque
19       * uma porção de memória para o programa e devolva o início dessa
20       * posição de memória. */
21
```

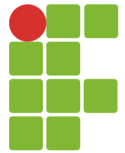


Exemplo

```
22      int *ptr;
23      /*A função malloc é responsável por fazer essa requisição ao
24       * sistema operacional */
25      ptr = malloc(sizeof(int));
26      if (ptr == NULL) {
27          printf("Erro de alocação.\n");
28      }
29      /*Dessa forma, ponteiros podem apontar para posições
30        de memória alocadas pelo sistema operaconal*/
31
32      *ptr = 3; // modificamos o conteúdo da memória alocada e apontada por ptr
33      printf("Valor do conteúdo de ptr = %p\n", ptr);
34      printf("Valor do conteúdo apontado por ptr = %d\n", *ptr);
35      /* O espaço alocado é liberado */
36      free(ptr);
37      return 0;
38  }
```

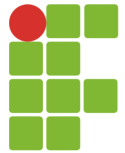


- Quando temos um ponteiro para `struct` é mais fácil utilizar o operador `->` para acessar seus membros.
- `ptr->membro` equivale à `(*ptr).membro`.



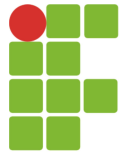
Exemplo

```
1  /**
2   * Autor: Daniel Saad Nogueira Nunes
3   * Comentários: Neste programa é explorada
4   * a alocação dinâmica de structs através
5   * de ponteiros em C.
6   **/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 typedef struct ExemploStruct {
12     int a, b, c;
13 } ExemploStruct;
14
```



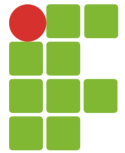
Exemplo

```
15  int main(void) {
16      ExemploStruct *ptr_estrutura;
17      /* Aloca dinamicamente uma estrutura e passa o endereco inicial
18       * da estrutura para o ponteiro */
19      ptr_estrutura = malloc(sizeof(ExemploStruct));
20      if (ptr_estrutura == NULL) {
21          printf("Erro de alocação.\n");
22          exit(EXIT_FAILURE);
23      }
24      /* O acesso em membros de estruturas apontadas por
25       * ponteiros é feito através do operador seta.
26       * Em resumo: (*estrutura_ptr).a é equivalente a
27       * estrutura_ptr->a. Preferimos a segunda forma por ser
28       * mais legível */
29      ptr_estrutura->a = 3;
30      ptr_estrutura->b = 4;
31      ptr_estrutura->c = 5;
32      /* Liberação do espaço alocado */
33      free(ptr_estrutura);
34      return (0);
35  }
```



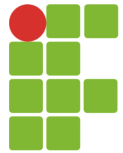
Vetores dinâmicos

- Lembra da similaridade entre ponteiros e vetores?
- Podemos criar vetores de quaisquer dimensão utilizando alocação dinâmica de memória.
- Nossa primeira ED dinâmica!



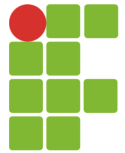
Exemplo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  int main(void){
5      int n;
6      int j;
7      srand(time(NULL));
8      printf("Digite o tamanho do vetor a ser alocado: ");
9      scanf("%d",&n);
10     /* Aloca espaço para o vetor e inicializa com zero */
11     int* v = calloc(n,sizeof(int));
12     if(v==NULL){
13         printf("Erro na alocação.\n");
14         exit(EXIT_FAILURE);
15     }
```



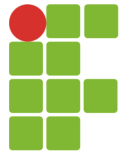
Exemplo

```
16      /* O vetor é preenchido com números aleatórios.
17       * Repare que o acesso á qualquer posição é feito através
18       * do operador [], como se fosse um vetor normal.
19       * De fato o que é feito é uma aritmética de ponteiros.
20       * v[i] = *(v+i) */
21      for(j=0;j<n;j++){
22          v[j] = rand() % 1000; /** Gera um numero aleatório entre 0 e 999 **/
23      }
24      /* Impressão do vetor */
25      for(j=0;j<n;j++){
26          printf("v[%d] = %d\n",j,v[j]);
27      }
28      /* O vetor é liberado */
29      free(v);
30      return 0;
31  }
```



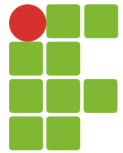
Matrizes dinâmicas

- Para criar uma matriz dinâmica 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**.



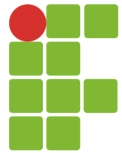
Exemplo

```
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 preenchia aleatóriamente através
7      da função rand();
8  **/
9
10
11 #include <stdio.h>
12 #include <time.h>
13 #include <stdlib.h>
14
```



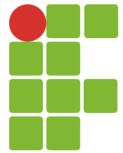
Exemplo

```
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      }
```



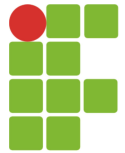
Exemplo

```
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 gerado */
44          }
45      }
```

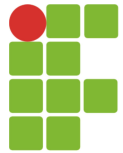


Exemplo

```
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  }
```

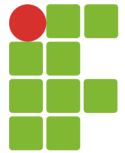


Sumário



Ponteiros para Funções

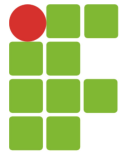
- Por de baixo dos panos um programa nada mais é que um conjunto de instruções.
- Uma função é um subconjunto de instruções que inicia em um dado endereço de memória.
- Se tivermos esse endereço através de um ponteiro, podemos invocar funções através de ponteiros para funções!



Ponteiros para Funções

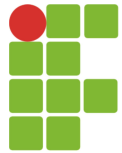
```c
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa realiza-se a
 * invocação de função através de um ponteiro
 * para a mesma.
 **/
#include <stdio.h>

int soma(int a, int b) {
    return (a + b);
}
```



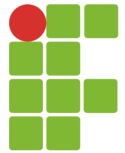
Ponteiros para Funções

```
13  int main(void) {
14      /* Declaração de ponteiro para função que
15       * retorna um inteiro e recebe dois */
16      int (*ptr)(int, int);
17      ptr = soma;
18      printf("Soma = %d.\n", ptr(1, 2));
19      return 0;
20  }
```



Ponteiros para funções

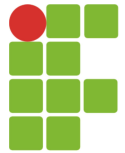
- Podemos utilizar um `typedef` para criar um tipo de ponteiro para função que retorna um inteiro e recebe dois inteiros.



Ponteiros para Funções

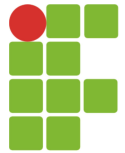
```c
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa realiza-se a
 * invocação de função através de um ponteiro
 * para a mesma.
 * Utiliza-se um typedef para declarar
 * um tipo de ponteiro para função que retorna
 * um inteiro e recebe dois argumentos inteiros.
**/
#include <stdio.h>

typedef int (*tipo_funcao_soma)(int,int);
```

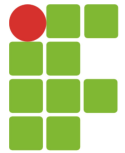


Ponteiros para Funções

```
13
14  int soma(int a,int b){
15      return(a+b);
16  }
17
18  int main(void){
19      /* Declaração de ponteiro para função do tipo
20       * tipo_funcao_soma */
21      tipo_funcao_soma ptr = soma;
22      printf("Soma = %d.\n",ptr(1,2));
23      return 0;
24  }
```

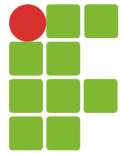


Sumário

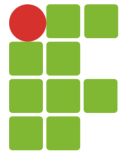


Cuidados

- 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).

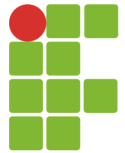


Sumário



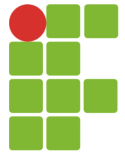
Memory Leak

- 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.



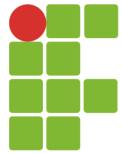
Exemplo

```
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 leaks
8   * são muitas das vezes decorrência de um erro de lógia.
9   * Para detectá-los, podemos usar a ferramenta valgrind.
10  **/
11
12 #include <stdlib.h>
13 int main(void) {
```



Exemplo

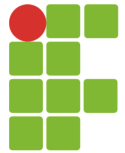
```
14
15      /* Aloca-se um etor de 100000 posições */
16      int *ptr = malloc(sizeof(int) * 100000);
17      /* MEMORY LEAK: atribui um novo endereço de memória para ptr
18       * sem desalocar o bloco de memória alocado */
19      ptr = NULL;
20      return (0);
21  }
```



Valgrind

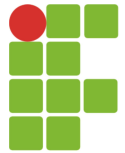
Vejamos a saída do `valgrind`:

```
$ 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)
```



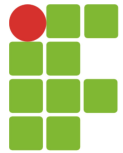
Correção do vazamento

- 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.



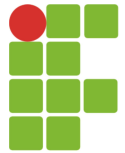
Exemplo

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

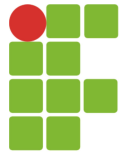


Exemplo

```
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 }
```

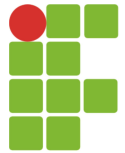


Sumário



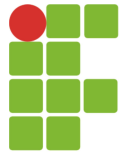
Wild Pointers

- **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 invalida 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.



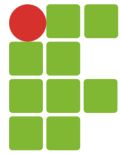
Exemplo

```
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 }
```



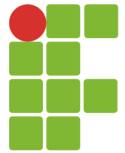
Exemplo

```
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  }
```



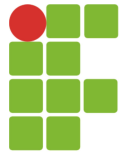
Exemplo

- 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 dinâmicamente a variável num.
5   * Ao alocarmos dinâmicamente, 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>
12
```



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  }
```