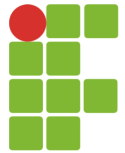
Ponteiros

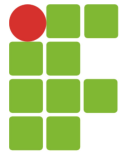
Estruturas de Dados e Algoritmos – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

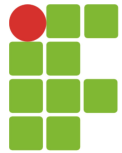
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

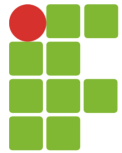


Sumário



Introdução

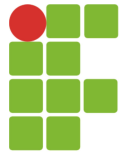
- Ponteiros são tipos de dados que armazenam endereços de memória.
- Eles são essenciais em algumas linguagens de programação (`C`) e em outras eles são restritos de alguma forma.
- Em `C`, ponteiros são tipos de variáveis que armazenam como valor um endereço de memória.



Sintaxe

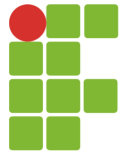
- Em C, a declaração de um ponteiro para uma região de memória é dada por:

```
<tipo>* nome_do_ponteiro;
```



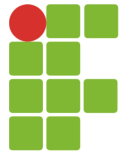
Sintaxe

```
int* ptr; /* Variável do tipo ponteiro para inteiro. Tem
     como valor um endereço ocupado por um inteiro*/

double* ptr; /* Variável do tipo ponteiro para double.
    Tem como valor um endereço ocupado por um double*/

int** ptr; /* Variável do tipo ponteiro para ponteiro
    para inteiro. Tem como valor um endereço ocupado por
     um ponteiro para inteiro.*/

void* ptr /* O que é isso? */
```



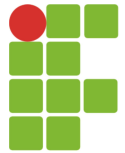
Sintaxe

- 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.



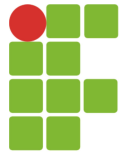
Sintaxe

```
int* ptr; /* Variável do tipo ponteiro para inteiro. Tem
     como valor um endereço ocupado por um inteiro*/

double* ptr; /* Variável do tipo ponteiro para double.
    Tem como valor um endereço ocupado por um double*/

int** ptr; /* Variável do tipo ponteiro para ponteiro
    para inteiro. Tem como valor um endereço ocupado por
     um ponteiro para inteiro.*/

void* ptr /* Variável do tipo ponteiro para ponteiro
    para inteiro. Tem como valor um endereço ocupado por
     alguma variável qualquer */
```



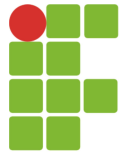
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa conceitos básicos
 * sobre ponteiros são explicados.
**/

#include <stdlib.h>
#include <stdio.h>
int main(void){

    /* Ponteiro para inteiro, contém o valor de
     * uma posição de memória que é ocupada por um
         inteiro*/
    int* ptr;
    int inteiro;
```



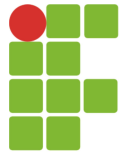
Exemplo

```
    /* ptr aponta para o endereço especial NULL**/
    ptr = NULL;

    /* Atribuímos a ptr, o endereco da variável inteiro
       */
    ptr = &inteiro;
    printf("O valor do ponteiro ptr = %p.\n",ptr);
    return(0);
}
```



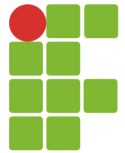
Ponteiros

- 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 deferenciação.
- Sintaxe: *<nome do ponteiro>.



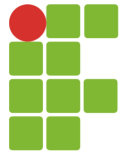
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Este programa aborda conceitos
 * básicos sobre ponteiros e deferência.
**/
#include <stdio.h>

int main(){

    /* Ponteiro para inteiro, contém o valor de uma posi
       ção de memória que é ocupada por um inteiro */
    int* ptr;

    /*um inteiro*/
    int var = 0;
```



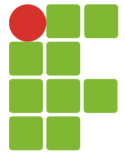
Exemplo

```
    printf("Var = %d\n",var);

    /*O valor de ptr aponta agora para o endereço de mem
       ória que corresponde ä variável var. */
    ptr = &var;

    /*Modificamos o *conteúdo* da regiao de memoria
       apontada por ptr*/
    (*ptr) = 1; /**Equivale a fazer var=1**/

    /*Note que agora o novo valor de var é 1*/
    printf("Var = %d\n",var);
    printf("Var = %d\n",*ptr);
    return(0);
}
```

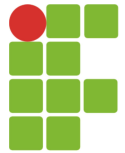


Ponteiros

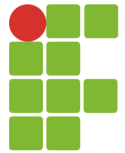
- Vamos nos aprofundar agora sobre o que ponteiros podem fazer por nós.



Sumário

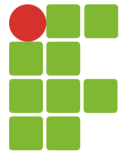


Sumário



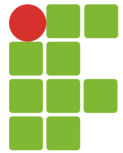
Vetores e Ponteiros

- 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!



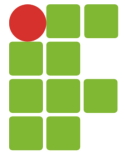
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa é ilustrado o fato
 * de que vetores podem ser vistos como ponteiros,
 * e vice-versa através da aritmética de ponteiros
**/

#include <stdio.h>

int main(void){

    char* ptr;
    char v[] = {'a','b','a','c','a','t','e','\0'};

    /* O nome de um vetor equivale ao endereço inicial
        de memória
```



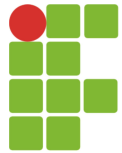
Exemplo

```
     * ocupado por ele. Logo ptr agora aponta para este
         mesmo
     * endereco. */
    ptr = v;
    printf("String original: %s\n",v);
    ptr[2] = 'd';
    printf("String modificada: %s\n",v);
    return(0);
}
```



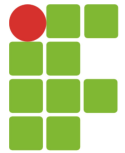
Vetores e Ponteiros

- 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)`



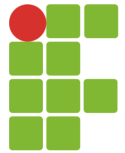
Aritmética de Ponteiros

- 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.



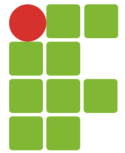
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa é ilustrado o fato
 * de que vetores podem ser vistos como ponteiros,
 * e vice-versa através da aritmética de ponteiros
**/


#include <stdio.h>

int main(void){
    char s[] = {'a','b','r','a','\0'};
    int v[] = {0,1,2,3};
    char* ptr_s = s;
    int* ptr_v = v;
    size_t i;
```

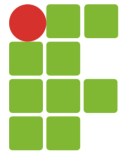


Exemplo

```
    for(i=0;i<4;i++){
        printf("Endereço de s[%d] = %p.\n",i,&s[i]);
        printf("Endereço de v[%d] = %p.\n",i,&v[i]);
    }
    for(i=0;i<4;i++){
        printf("ptr_s + %d = %p.\n",i,ptr_s + i);
        printf("ptr_v + %d = %p.\n",i,ptr_v + i);
    }
        return 0;
}
```

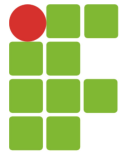


Sumário



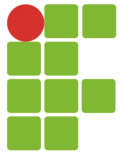
Passagem por Referência

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



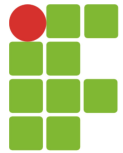
Passagem por Referência

- 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.



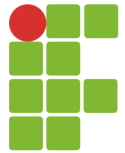
Passagem por Referência

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa é explorada
 * a emulação de passagem por referência em funções na
 * linguagem C.
**/

#include <stdio.h>

/* Em C, podemos emular uma passagem por referência
    através de
 * ponteiros.
 * Neste caso, uma cópia do ponteiro que aponta para o
    endereço de x
 * é criada. Como a cópia aponta para o endereço de x,
    podemos modificar
```

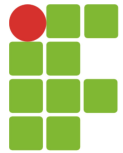


Passagem por Referência

```
 * o conteúdo da região de memória apontada por x. **/

void cubo(double *x){
    *x = *x * *x * *x;
}

int main(void){
    double a = 3;
    printf("O cubo de %lf é ",a);
    cubo(&a);
    printf("%lf\n",a);
    return(0);
}
```

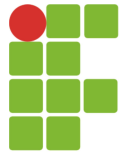


Sumário



Alocação Dinâmica de Memória

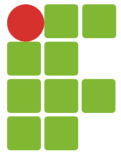
- É possível requisitar alocação de memória ao S.O de maneira dinâmica.
- Geralmente através das chamadas `realloc`,`malloc` e `calloc`.
- Estas chamadas retornam ponteiros para as regiões alocadas em caso de sucesso.
- Verifiquemos as assinaturas destas funções.



Alocação Dinâmica de Memória

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

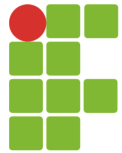
- `size`:= tamanho em bytes da região a ser alocada.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



Alocação Dinâmica de Memória

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

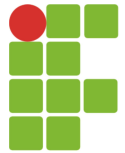
- Parecida com `malloc`, mas faz o favor de inicializar a área alocada com zeros.
- `num`:= número de elementos.
- `size`:= tamanho em bytes de cada elemento.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



Alocação Dinâmica de Memória

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

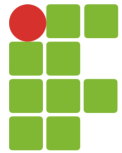
- Redimensiona a área alocada.
- `ptr`:= ponteiro para região antiga.
- `size`:= quantidade em bytes da região a ser realocada.
- Retorna um ponteiro para a região de memória alocada em caso de sucesso.
- Retorna `NULL` em caso de falha.



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.

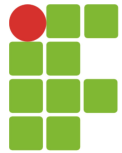


Alocação Dinâmica de Memória

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

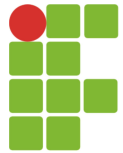
```
void free(void* ptr);
```

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



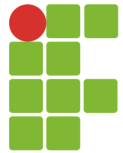
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa é explorada
 * a alocação dinâmica de memória através
 * de ponteiros em C.
**/

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

int main(void){

    /* Ponteiros inicialmente apontam para uma posição
     * de memória arbitrária.
     * É necessário atribuir a eles uma posição de memó
       ria válida
```



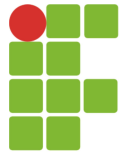
Exemplo

```
 * que pertença ao seu programa. */
int* ptr;

/* Podemos requisitar ao sistema operacional que ele
     aloque
 * uma porção de memória para o programa e devolva o
     início dessa
 * posição de memória. */

/*A função malloc é responsável por fazer essa
    requisição ao
 * sistema operacional */
ptr = malloc(sizeof(int));
if(ptr==NULL){
    printf("Erro de alocação.\n");
}
```



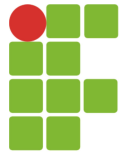
Exemplo

```
/*Dessa forma, ponteiros podem apontar para posições
  de memória alocadas pelo sistema operaconal*/

*ptr = 3; //modificamos o conteúdo da memória
    alocada e apontada por ptr

printf("Valor do conteúdo de ptr = %p\n",ptr);
printf("Valor do conteúdo apontado por ptr = %d\n",*
    ptr);

/* O espaço alocado é liberado */
free (ptr);

return 0;
```

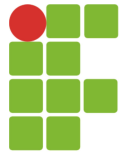


Exemplo

```
}
```

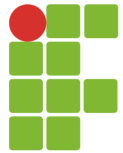


- Quando temos um ponteiro para `struct` é mais barato utilizar o operador -> para acessar seus membros.
- `ptr->membro` equivale à `(*ptr).membro`.



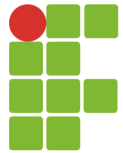
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa é explorada
 * a alocação dinâmica de structs através
 * de ponteiros em C.
**/


#include <stdlib.h>
#include <stdio.h>
typedef struct ExemploStruct{
        int a,b,c;
}ExemploStruct;

int main(void){
    ExemploStruct* ptr_estrutura;
```



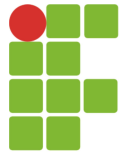
Exemplo

```
/* Aloca dinamicamente uma estrutura e passa o
    endereco inicial
 * da estrutura para o ponteiro */
ptr_estrutura = malloc(sizeof(ExemploStruct));
if(ptr_estrutura==NULL){
    printf("Erro de alocação.\n");
    exit(EXIT_FAILURE);
}

/* O acesso em membros de estruturas apontadas por
 * ponteiros é feito através do operador seta.
 * Em resumo: (*estrutura_ptr).a é equivalente a
 * estrutura_ptr->a. Preferimos a segunda forma por
    ser
 * mais legível */
```



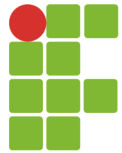
Exemplo

```
    ptr_estrutura->a=3;
    ptr_estrutura->b=4;
    ptr_estrutura->c=5;

    /* Liberação do espaço alocado */
    free (ptr_estrutura);

    return(0);

}
```



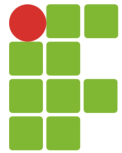
Vetores Dinâmicos

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



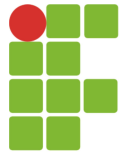
Exemplo

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void){
    int n;
    int j;
    srand(time(NULL));
    printf("Digite o tamanho do vetor a ser alocado: ");
    scanf("%d",&n);
    /* Aloca espaço para o vetor e inicializa com zero
       */
    int* v = calloc(n,sizeof(int));
    if(v==NULL){
        printf("Erro na alocação.\n");
        exit(EXIT_FAILURE);
    }
```



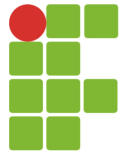
Exemplo

```
/* O vetor é preenchido com números aleatórios.
 * Repare que o acesso á qualquer posição é feito
     através
 * do operador [], como se fosse um vetor normal.
 * De fato o que é feito é uma aritmética de
     ponteiros.
 * v[i] = *(v+i) */
for(j=0;j<n;j++){
    v[j] = rand() % 1000; /** Gera um numero aleató
        rio entre 0 e 999 **/
}
/* Impressão do vetor */
for(j=0;j<n;j++){
    printf("v[%d] = %d\n",j,v[j]);
}
/* O vetor é liberado */
```



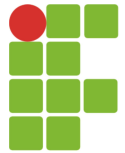
Exemplo

```
    free(v);
    return 0;
}
```



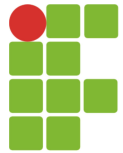
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Neste programa realiza-se a
    alocação dinâmica de memória de uma matriz.
    Nele são lidas os número de linhas e colunas
    e a matriz é preenchia aleatóriamente através
    da função rand();
**/


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

int main(void){
    int l,c;
```



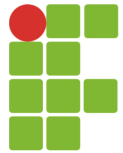
Exemplo

```
int i,j;
srand(time(NULL));
printf("Digite o número de linhas da matriz: ");
scanf("%d",&l);
printf("Digite o número de colunas da matriz: ");
scanf("%d",&c);
/* Alocamos um vetor de ponteiros */
int** m = calloc(l,sizeof(int*));
if(m==NULL){
    printf("Erro na alocação.\n");
    exit(EXIT_FAILURE);
}
/* O ponteiro zero recebe o espaço da matriz
 * isto é, l*c */
m[0] = calloc(l*c,sizeof(int));
if(m[0]==NULL){
```



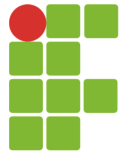
Exemplo

```
        printf("Erro na alocação.\n");
        exit(EXIT_FAILURE);
    }
    /* Cada um dos ponteiros recebe o início de uma regi
        ão
     * de memória apontada por m[0] */
    for(j=1;j<l;j++){
        m[j] = m[0]+j*c;
    }
    for(i=0;i<l;i++){
        for(j=0;j<c;j++){
            m[i][j] = rand() % 1000; /* Um inteiro aleat
                ório [0,999] é gerado */
        }
    }
    /* Impressão da matriz */
```

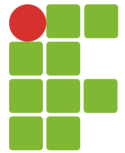


Exemplo

```
    for(i=0;i<l;i++){
        for(j=0;j<c;j++){
            printf("%3d ",m[i][j]);
        }
        printf("\n");
    }
    /* O espaço alocado é liberado */
    free(m[0]);
    free(m);

    return 0;
}
```

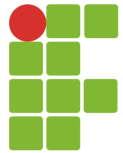


Sumário



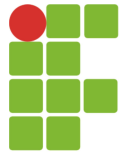
Ponteiros para Funções

- 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!
- Isso vai ser muito importante pra nós daqui pra frente.



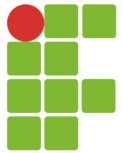
Ponteiros para Funções

```
/**
 * 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);
}

int main(void){
    /* Declaração de ponteiro para função que
     * retorna um inteiro e recebe dois */
    int (*ptr)(int,int);
```



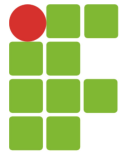
Ponteiros para Funções

```
    ptr = soma;
    printf("Soma = %d.\n",ptr(1,2));
    return 0;
}
```



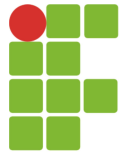
Ponteiros para Funções

```
/**
 * 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 (*ptr_soma)(int,int);

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

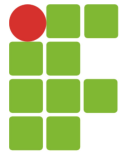


Ponteiros para Funções

```
int main(void){
    /* Declaração de ponteiro para função do tipo
     * ptr_soma */
    ptr_soma ptr = soma;
    printf("Soma = %d.\n",ptr(1,2));
    return 0;
}
```

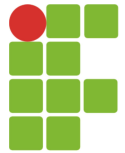


Sumário

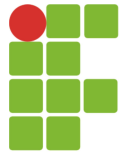


Cuidados

- 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).

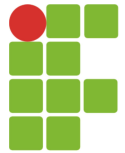


Sumário



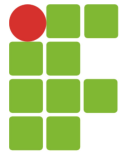
Memory Leak

- 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.



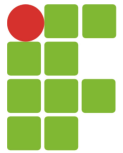
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Este problema aborda uma péssima prática
 * de programação. Os vazamentos de memória (memory
    leaks).
 * Estes vazamentos consistem na perda da referência
    para uma
 * área alocada, tornando impossível acessar esta área
    novamente.
 * O consumo de memória é aumentado desnecessariamente e
     memory leaks
 * são muitas das vezes decorrência de um erro de lógia.
 * Para detectá-los, podemos usar a ferramenta valgrind.
**/
```



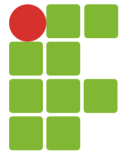
Exemplo

```c
#include <stdlib.h>
int main(void){

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



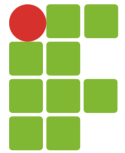
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Este programa aborda a correção do
     exemplo anterior.
 * A memória é liberada antes de trocarmos o valor do
     ponteiro.
 * Rode ele com o valgrind e compare a diferença de saí
     da entre os dois.
**/

#include <stdlib.h>
int main(void){
    /* aloca-se um vetor de 100000 posições.*/
    int* ptr = malloc(sizeof(int)* 100000);
    /* libera-se a área de memória alocada */
    free (ptr);
```

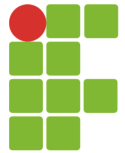


Exemplo

```
    /* agora podemos mudar o valor de ptr */
    ptr = NULL;

    return(0);
}
```

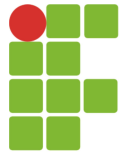


Sumário



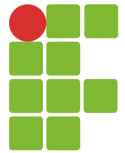
Wild Pointers

- **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.



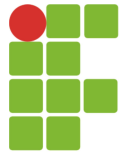
Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Este programa mostra um exemplo
 * de um ponteiro selvagem. A variável num só
 * existe no escopo de func, e portanto, seu endereço
 * não é mais válido quando a função termina.
**/

#include <stdio.h>

int* func(void){
     int num = 1234;
     /* ... */
     return &num;
}
```



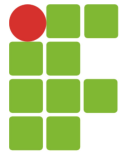
Exemplo

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



Exemplo

```
/**
 * Autor: Daniel Saad Nogueira Nunes
 * Comentários: Este programa corrige o anterior
 * ao alocar dinâmicamente a variável num.
 * Ao alocarmos dinâmicamente, o endereço persiste
 * até que um free() seja utilizado, logo, o exemplo
 * abaixo não configura um ponteiro selvagem;
**/

#include <stdio.h>

int* func(void){
    int* num = malloc(sizeof(int));
    if(num==NULL){
        printf("Erro de alocação.\n");
    }
```



Exemplo

```
    *num = 1234;
    return num;
}

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