Introdução a Ponteiros

Estrutura de Dados — QXD0010

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará
Campus Quixadá

2º semestre/2025

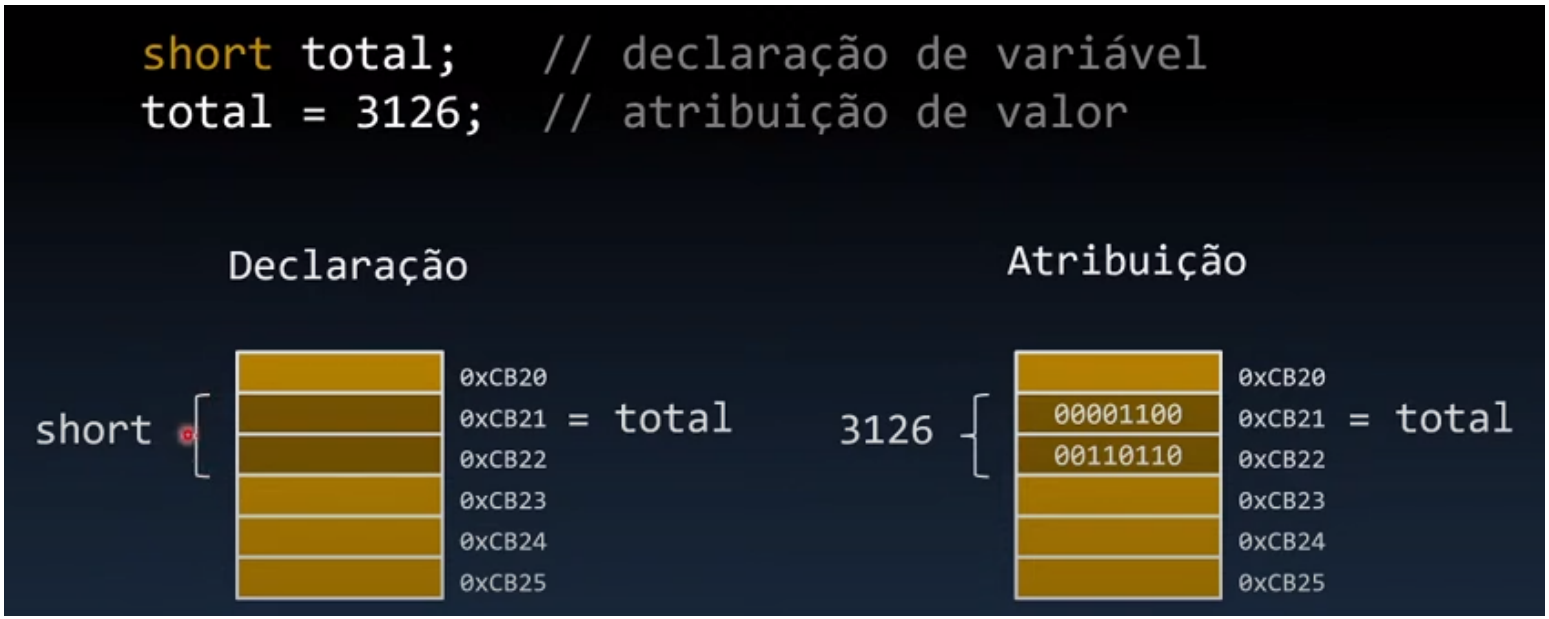
Memória e Variáveis

- Pense na memória do computador como um grande vetor de células consecutivamente numeradas.

- Quando uma variável x é alocada, um grupo de células é separado para armazenar o dado que x vai conter, e aquele grupo de células passa a ser referenciado com o nome x.

Memória e Variáveis

- Ao criar uma variável, um programa gerencia:
  - onde a informação está armazenada (em que posição da memória está)
  - que tipo de informação é armazenada
  - o valor que é mantido lá

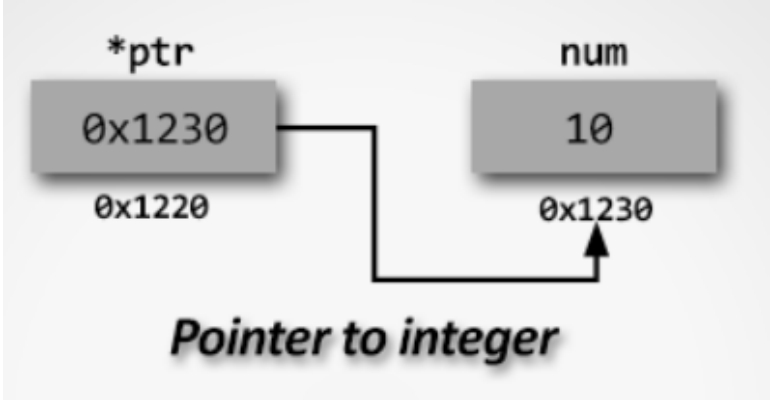


Endereços de Variáveis

- Todo nome de variável está associado a um endereço de memória.
  - O operador de endereço & obtém a sua localização
- Exemplo: `printf("%p", &total)`
- Por diversos motivos que ficarão claros futuramente, gostaríamos de guardar o endereço de uma variável em algum lugar, para uso posterior.

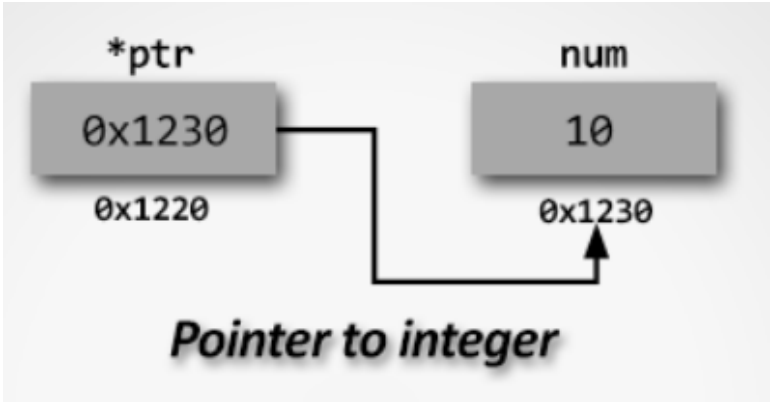
Ponteiros

- Um **ponteiro** é um tipo especial de variável que armazena um endereço de memória.



Ponteiros

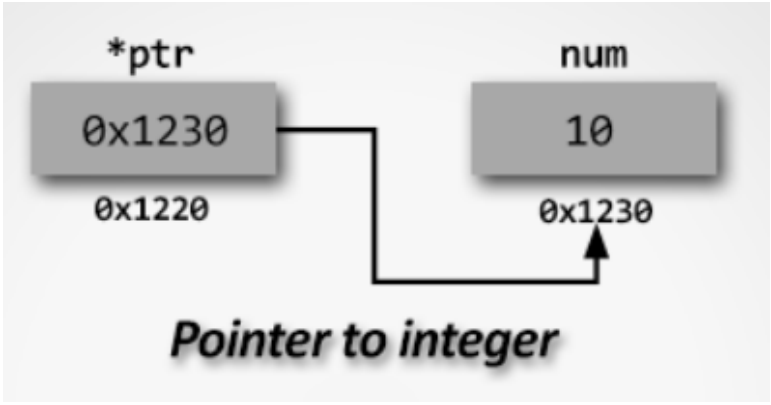
- Um **ponteiro** é um tipo especial de variável que armazena um endereço de memória.



- Cada variável tem um endereço, inclusive o ponteiro.

Ponteiros

- Um **ponteiro** é um tipo especial de variável que armazena um endereço de memória.



- Cada variável tem um endereço, inclusive o ponteiro.

- Com as variáveis que vimos até o momento, podemos acessar um valor em um local fixo da memória. Porém, os ponteiros podem acessar diferentes regiões de memória.

Declarando ponteiros

- Em C, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

Declarando ponteiros

- Em C, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o asterisco (*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

Declarando ponteiros

- Em C, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o asterisco (*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

- Exemplo:

```
1 int *p_int; // p_int eh um ponteiro para int
2 double *p_d; // p_d eh um ponteiro para double
```

Declarando ponteiros

- Em C, a declaração de um ponteiro pelo programador segue esta forma:

```
tipo_de_dado * nome_do_ponteiro;
```

- É o asterisco (*) que informa ao compilador que a variável `nome_do_ponteiro` não vai guardar um valor, mas um endereço de memória para o tipo especificado.

- Exemplo:

```
1 int *p_int; // p_int eh um ponteiro para int
2 double *p_d; // p_d eh um ponteiro para double
```

- Quando declaramos um ponteiro, informamos ao compilador para que tipo de variável poderemos apontá-lo.

Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador de endereçamento, &, na frente do nome da variável.

Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador de endereçamento, &, na frente do nome da variável.

```c
#include <stdio.h>

int main() {
    int x = 5;
    printf("%d\n", x); // imprime 5

    // imprime o endereco de memoria da variavel x
    printf("%p\n", &x);
}
```

Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador de endereçamento, &, na frente do nome da variável.

```c
#include <stdio.h>

int main() {
    int x = 5;
    printf("%d\n", x); // imprime 5

    // imprime o endereco de memoria da variavel x
    printf("%p\n", &x);
}
```

- Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:

Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador de endereçamento, &, na frente do nome da variável.

```c
#include <stdio.h>

int main() {
    int x = 5;
    printf("%d\n", x); // imprime 5

    // imprime o endereco de memoria da variavel x
    printf("%p\n", &x);
}
```

- Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:
  - Acessar o endereço de memória de uma variável, usando o operador &

Acessando endereços de memória

- Para saber o endereço onde uma variável está guardada na memória, usa-se o operador de endereçamento, &, na frente do nome da variável.

```c
#include <stdio.h>

int main() {
    int x = 5;
    printf("%d\n", x); // imprime 5

    // imprime o endereco de memoria da variavel x
    printf("%p\n", &x);
}
```

- Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:
  - Acessar o endereço de memória de uma variável, usando o operador &
  - Acessar o conteúdo de um endereço de memória, usando o operador *

Manipulando ponteiros — Exemplo


```c
#include <stdio.h> // prog35.c

int main() {
  // Declara uma variavel int contendo o valor 10
  int count = 10;
```

Manipulando ponteiros — Exemplo


```c
#include <stdio.h> // prog35.c

int main() {
  // Declara uma variavel int contendo o valor 10
  int count = 10;

  // Declara um ponteiro para int e atribui ao ponteiro
  // o endereco da variavel int
  int *p;
  p = &count;
```

Manipulando ponteiros — Exemplo


```c
#include <stdio.h> // prog35.c

int main() {
  // Declara uma variavel int contendo o valor 10
  int count = 10;

  // Declara um ponteiro para int e atribui ao ponteiro
  // o endereco da variavel int
  int *p;
  p = &count;
  printf("Conteudo de p: %p\n", p);
  //Imprime 10
  printf("Conteudo apontado por p: %d\n", *p);
```

Manipulando ponteiros — Exemplo


```c
#include <stdio.h> // prog35.c

int main() {
  // Declara uma variavel int contendo o valor 10
  int count = 10;

  // Declara um ponteiro para int e atribui ao ponteiro
  // o endereco da variavel int
  int *p;
  p = &count;
  printf("Conteudo de p: %p\n", p);
  //Imprime 10
  printf("Conteudo apontado por p: %d\n", *p);

  // Atribui um novo valor a posicao de memoria apontada por p
  *p = 12;
```

Manipulando ponteiros — Exemplo


```c
#include <stdio.h> // prog35.c

int main() {
  // Declara uma variavel int contendo o valor 10
  int count = 10;

  // Declara um ponteiro para int e atribui ao ponteiro
  // o endereco da variavel int
  int *p;
  p = &count;
  printf("Conteudo de p: %p\n", p);
  //Imprime 10
  printf("Conteudo apontado por p: %d\n", *p);

  // Atribui um novo valor a posicao de memoria apontada por p
  *p = 12;

  // As duas linhas abaixo imprimem o numero 12 na tela
  printf("Conteudo apontado por p: %d\n", *p);
  printf("Conteudo de count: %d\n", count);

  return 0;
}
```

Atribuição entre ponteiros

- Em geral, no C, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

Atribuição entre ponteiros

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Em geral, no C, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```
1 #include <stdio.h> // prog37.c
2
3 int main() {
4   float f = 45.78;
5
6   int *ptr = &f; // Erro de compilacao
```

Atribuição entre ponteiros

- Em geral, no C, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```c
#include <stdio.h> // prog37.c

int main() {
  float f = 45.78;

  int *ptr = &f; // Erro de compilacao

  int i = 54;

  float *f2 = &i; // Erro de compilacao
```

Atribuição entre ponteiros

- Em geral, no C, um ponteiro só pode receber o endereço de memória de uma variável do mesmo tipo do ponteiro.

```c
#include <stdio.h> // prog37.c

int main() {
  float f = 45.78;

  int *ptr = &f; // Erro de compilacao

  int i = 54;

  float *f2 = &i; // Erro de compilacao
}
```

Qual o tamanho de um ponteiro?

- O tamanho de um ponteiro depende da arquitetura para a qual o programa é compilado.
  - Arquitetura de 32 bits — ponteiro ocupa 32 bits (4 bytes).
  - Arquitetura de 64 bits — ponteiro ocupa 64 bits (8 bytes). Independentemente do que está sendo apontado.

Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.

Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.
- Valor nulo é um valor especial que significa que o ponteiro não está apontando para nada. Um ponteiro que contém um valor nulo é chamado de ponteiro nulo.

Ponteiro nulo

- Além de endereços de memória, todo ponteiro pode armazenar o valor nulo.
- Valor nulo é um valor especial que significa que o ponteiro não está apontando para nada. Um ponteiro que contém um valor nulo é chamado de ponteiro nulo.
- Em C, a forma de tornar um ponteiro nulo consiste em atribuir-lhe a palavra-chave `NULL`.

Ponteiro nulo

- Boa prática de programação 1: Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.

Ponteiro nulo

- Boa prática de programação 1: Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.

- Erro Comum em C: Tentar desreferenciar um ponteiro não inicializado ou um ponteiro nulo é ilegal e potencialmente fará com que seu programa seja encerrado.

```
1 int *ptr = nullptr;
2 *ptr = 2023; // erro: falha de segmentacao
```

Ponteiro nulo

- Boa prática de programação 1: Se, no momento da declaração de um ponteiro, nenhum endereço de memória válido for atribuído ao ponteiro, então inicialize o ponteiro com um valor nulo.

- Erro Comum em C: Tentar desreferenciar um ponteiro não inicializado ou um ponteiro nulo é ilegal e potencialmente fará com que seu programa seja encerrado.

```
1 int *ptr = nullptr;
2 *ptr = 2023; // erro: falha de segmentacao
```

- Boa prática de programação 2: Antes de usar um ponteiro, certifique-se de que ele não é um ponteiro nulo. Exemplo:

```
1 if (ptr != NULL) {
2   *ptr = 2023; // ptr NAO eh nulo
3 } else {
4   printf("ponteiro nulo");
5 }
```

Passagem de argumentos para funções

Passagem de argumentos para funções

- Quando passamos argumentos para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função. Este processo é idêntico a uma atribuição.

Passagem de argumentos para funções

- Quando passamos argumentos para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função. Este processo é idêntico a uma atribuição.

- Este processo é chamado de **passagem por valor**.

Passagem de argumentos para funções

- Quando passamos argumentos para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função. Este processo é idêntico a uma atribuição.

- Este processo é chamado de **passagem por valor**.

- Desta forma, alterações nos parâmetros dentro da função não alteram os valores que foram passados na chamada da função.

Passagem de argumentos para funções

- Quando passamos argumentos para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função. Este processo é idêntico a uma atribuição.

- Este processo é chamado de **passagem por valor**.

- Desta forma, alterações nos parâmetros dentro da função não alteram os valores que foram passados na chamada da função.

```
1  #include <stdio.h>
2
3  void dobra(int x) {
4      x = x * 2;
5  }
6
7  int main() {
8      int a = 3;
9      dobra(a);
10     printf("%d", a); // Qual valor é impresso aqui?
11 }
```

Passagem de argumentos para funções

- Em C só existe passagem de argumentos por valor.

Passagem de argumentos para funções

- Em C só existe passagem de argumentos por valor.
- Em algumas linguagens existem construções para se passar argumentos por referência. Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.

Passagem de argumentos para funções

- Em C só existe passagem de argumentos por valor.
- Em algumas linguagens existem construções para se passar argumentos por referência. Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
- Podemos obter algo semelhante em C utilizando ponteiros:

Passagem de argumentos para funções

- Em C só existe passagem de argumentos por valor.
- Em algumas linguagens existem construções para se passar argumentos por referência. Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
- Podemos obter algo semelhante em C utilizando ponteiros:
  - O artifício corresponde em passar como argumento para uma função o endereço da variável, e não o seu valor.

Passagem de argumentos para funções

- Em C só existe passagem de argumentos por valor.
- Em algumas linguagens existem construções para se passar argumentos por referência. Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
- Podemos obter algo semelhante em C utilizando ponteiros:
  - O artifício corresponde em passar como argumento para uma função o endereço da variável, e não o seu valor.
  - Desta forma podemos alterar o conteúdo da variável como se fizéssemos passagem por referência.

Exercício

- É muito comum, em diversos algoritmos, querermos trocar os valores de duas variáveis. Em inglês, essa operação é chamada de `swap`.
  Como fazer a troca de valores de duas variáveis inteiras `x` e `y`?

Exercício

- É muito comum, em diversos algoritmos, querermos trocar os valores de duas variáveis. Em inglês, essa operação é chamada de `swap`.
  Como fazer a troca de valores de duas variáveis inteiras `x` e `y`?

- Gostaríamos de criar uma função que recebe como argumento duas variáveis e faz a troca delas. Isso é possível em C?

Exercício

- É muito comum, em diversos algoritmos, querermos trocar os valores de duas variáveis. Em inglês, essa operação é chamada de `swap`.
  Como fazer a troca de valores de duas variáveis inteiras `x` e `y`?

- Gostaríamos de criar uma função que recebe como argumento duas variáveis e faz a troca delas. Isso é possível em C?

- Implemente uma função que recebe como argumento os endereços de duas variáveis do tipo `int` e troca os valores das variáveis.
  Sua função deve obedecer o protótipo:

```
void swap(int *p1, int *p2);
```

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Ponteiros e arrays

Ponteiros e arrays

- Quando declaramos uma variável do tipo vetor, é alocada uma quantidade de memória contígua cujo tamanho é especificado na declaração (e também depende do tipo do vetor)

  `int v[3];` `serão alocados 3*4 bytes de memória`

Ponteiros e arrays

- Quando declaramos uma variável do tipo vetor, é alocada uma quantidade de memória contígua cujo tamanho é especificado na declaração (e também depende do tipo do vetor)

  `int v[3];` `serão alocados 3*4 bytes de memória`

- Uma variável de vetor, assim como um ponteiro, armazena o endereço do início do vetor.

Ponteiros e arrays

- Quando declaramos uma variável do tipo vetor, é alocada uma quantidade de memória contígua cujo tamanho é especificado na declaração (e também depende do tipo do vetor)

  `int v[3];` `serão alocados 3*4 bytes de memória`

- Uma variável de vetor, assim como um ponteiro, armazena o endereço do início do vetor.

```
1 int main() {
2     int vet[3] = {10,20,30};
3     printf("%p\n", vet);  // endereço do primeiro elemento
4 }
```

Ponteiros e arrays

- Quando declaramos uma variável do tipo vetor, é alocada uma quantidade de memória contígua cujo tamanho é especificado na declaração (e também depende do tipo do vetor)

  `int v[3];` `serão alocados 3*4 bytes de memória`

- Uma variável de vetor, assim como um ponteiro, armazena o endereço do início do vetor.

```
1 int main() {
2     int vet[3] = {10,20,30};
3     printf("%p\n", vet);  // endereço do primeiro elemento
4 }
```

- Quando passamos um vetor como argumento para uma função, seu conteúdo pode ser alterado dentro da função pois estamos passando na realidade o endereço inicial do espaço alocado para o vetor.

Ponteiros e arrays

- Ponteiros e arrays estão intrinsecamente relacionados.

Ponteiros e arrays

- Ponteiros e arrays estão intrinsecamente relacionados.
- Quando um array é usado em uma expressão, ele é implicitamente convertido em um ponteiro que aponta para o primeiro elemento do vetor.

Ponteiros e arrays — Exemplo


```c
#include <stdio.h> // prog38.c

int main() {
    int array[5] = { 9,7,5,3,1 };

    // valor do primeiro elemento de 'array'
    printf("array[0]: %d\n", array[0]); // 9

    // imprime o endereco do primeiro elemento de 'array'
    printf("Endereco de array[0]: %p\n", &array[0]); // hexa

    // imprime o valor do ponteiro para o qual 'array' decai
    printf("array decai para um ponteiro "
        "com endereço: %p\n", array); // hexa

    for(int i = 0; i < 5; ++i)
    printf("%d: %d\n", i,  array[i]);

    return 0;
}
```

Ponteiros e arrays

- Como um array, usado em uma expressão, vira um ponteiro que aponta para o primeiro elemento do array, podemos desreferenciar o array a fim de obter o valor do primeiro elemento:

Ponteiros e arrays

- Como um array, usado em uma expressão, vira um ponteiro que aponta para o primeiro elemento do array, podemos desreferenciar o array a fim de obter o valor do primeiro elemento:

```
1 #include <stdio.h> // prog39.c
2
3 int main() {
4   int array[5] = { 9,7,5,3,1 };
```

Ponteiros e arrays

- Como um array, usado em uma expressão, vira um ponteiro que aponta para o primeiro elemento do array, podemos desreferenciar o array a fim de obter o valor do primeiro elemento:

```
1 #include <stdio.h> // prog39.c
2
3 int main() {
4   int array[5] = { 9,7,5,3,1 };
```

Ponteiros e arrays

- Como um array, usado em uma expressão, vira um ponteiro que aponta para o primeiro elemento do array, podemos desreferenciar o array a fim de obter o valor do primeiro elemento:

```c
#include <stdio.h> // prog39.c

int main() {
  int array[5] = { 9,7,5,3,1 };

  // dereferenciar um array retorna o primeiro elemento
  printf("%d\n", *array); // imprime 9!
```

Ponteiros e arrays

- Como um array, usado em uma expressão, vira um ponteiro que aponta para o primeiro elemento do array, podemos desreferenciar o array a fim de obter o valor do primeiro elemento:

```c
#include <stdio.h> // prog39.c

int main() {
  int array[5] = { 9,7,5,3,1 };

  // dereferenciar um array retorna o primeiro elemento
  printf("%d\n", *array); // imprime 9!

  // Dada essa propriedade dos arrays, podemos declarar
  // um ponteiro do tipo int e fazer ele apontar para array
  int *ptr = array;
  printf("%d\n", *ptr); // imprime 9

  return 0;
}
```

Passando arrays como argumentos para funções

Ao passar um array como um argumento para uma função, ele vira um ponteiro e o ponteiro é passado para a função.

Passando arrays como argumentos para funções

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Ao passar um array como um argumento para uma função, ele vira um ponteiro e o ponteiro é passado para a função.

```c
#include <stdio.h> // prog42.c

void printSize(int array[]) {
    // array eh tratado como ponteiro aqui
    // o tamanho do ponteiro sera impresso
    printf("%ld\n", sizeof(array));
}

int main() {
    int array[] = { 1,1,2,3,5,8,13,21 };

    // imprime sizeof(int) * array size que eh igual a 32
    printf("%ld\n", sizeof(array));

    printSize(array); // array decai para um ponteiro aqui
}
```

Passando arrays como argumentos para funções

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```c
#include <stdio.h> // prog43.c

// ptr contem uma copia do endereco passado como parametro
void modifica_array(int ptr[]) {
    *ptr = 5; // o que faz essa linha?
}

int main() {
    int vec[] = { 1,4,2,3,7,8,13,21 };
    modifica_array(vec);
    int size = sizeof(vec) / sizeof(vec[0]);
    for(int i = 0; i < size; i++)
        printf("vec[%d] = %d\n", i, vec[i]);
    return 0;
}
```

- Quando `modifica_array()` é chamado, `vec` decai em um ponteiro e o valor desse ponteiro é copiado no parâmetro `ptr`.

Passando arrays como argumentos para funções

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

```c
#include <stdio.h> // prog43.c

// ptr contem uma copia do endereco passado como parametro
void modifica_array(int ptr[]) {
    *ptr = 5; // o que faz essa linha?
}

int main() {
    int vec[] = { 1,4,2,3,7,8,13,21 };
    modifica_array(vec);
    int size = sizeof(vec) / sizeof(vec[0]);
    for(int i = 0; i < size; i++)
        printf("vec[%d] = %d\n", i, vec[i]);
    return 0;
}
```

- Quando `modifica_array()` é chamado, `vec` decai em um ponteiro e o valor desse ponteiro é copiado no parâmetro `ptr`.
- Embora o valor em `ptr` seja uma cópia do endereço de `vec`, `ptr` ainda aponta para o vetor real. Assim, quando `ptr` é desreferenciado, o vetor é desreferenciado.

Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: adição e subtração.

Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: adição e subtração.
- Se `ptr` apontar para um inteiro, `ptr+1` é o endereço do próximo inteiro na memória após o `ptr`.
  E `ptr-1` é o endereço do inteiro antes de `ptr`.

Aritmética de ponteiros

- Apenas duas operações aritméticas podem ser utilizadas nos endereços armazenados pelos ponteiros: adição e subtração.
- Se `ptr` apontar para um inteiro, `ptr+1` é o endereço do próximo inteiro na memória após o `ptr`.
  E `ptr-1` é o endereço do inteiro antes de `ptr`.

```
1  #include <stdio.h> // prog44.c
2
3  int main() {
4      int array[5] = { 0,1,2,3,4 };
5      int *ptr = array; // ptr aponta para o primeiro
6                        // elemento do array
7
8      printf("%p : %d\n", ptr, *ptr);
9      printf("%p : %d\n", ptr+1, *(ptr+1));
10     printf("%p : %d\n", ptr+2, *(ptr+2));
11     printf("%p : %d\n", ptr+3-2, *(ptr+3-2));
12
13     return 0;
14 }
```

Aritmética de ponteiros

- Subtrair dois ponteiros dá a distância entre eles. Os ponteiros subtraídos devem ser do mesmo tipo.

Aritmética de ponteiros

- Subtrair dois ponteiros dá a distância entre eles. Os ponteiros subtraídos devem ser do mesmo tipo.
- O valor resultante da subtração de dois ponteiros é do tipo `ptrdiff_t`, que é um inteiro com sinal.

Aritmética de ponteiros

- Subtrair dois ponteiros dá a distância entre eles. Os ponteiros subtraídos devem ser do mesmo tipo.
- O valor resultante da subtração de dois ponteiros é do tipo `ptrdiff_t`, que é um inteiro com sinal.

```c
#include <stdio.h> // prog45.c

int main() {
    int array[5] = { 9, 1, 2, 3, 4 };

    int *ptr = array;

    long unsigned valor = (ptr + 5) - ptr; // valor == 5

    printf("%ld\n", valor);

    return 0;
}
```

Ponteiros e indexação de arrays

- Quando o compilador vê o operador de indexação [ ], ele o traduz em uma adição de ponteiro seguida de derreferência.
  - ou seja, `array[n]` é o mesmo que `*(array+n)`, onde `n` é um inteiro.

Ponteiros e indexação de arrays

- Quando o compilador vê o operador de indexação [ ], ele o traduz em uma adição de ponteiro seguida de derreferência.
  - ou seja, `array[n]` é o mesmo que `*(array+n)`, onde `n` é um inteiro.

```
1  #include <stdio.h> // prog47.c
2
3  int main() {
4      int array [5] = { 9,7,5,3,1 };
5
6      // imprime o endereco do elemento array[1]
7      printf("%p\n", &array[1]);
8      // imprime o endereco do ponteiro (array+1)
9      printf("%p\n", array+1);
10
11     printf("%d\n", array[1]);   // imprime 7
12     printf("%d\n", *(array+1)); // imprime 7
13
14     return 0;
15 }
```

Ponteiros e operadores relacionais

- Os operadores relacionais ($>$, $<$, $>=$, $<=$, $==$, $!=$) podem ser usados para comparar dois ponteiros **do mesmo tipo**.

Ponteiros e operadores relacionais

- Os operadores relacionais ($>$, $<$, $>=$, $<=$, $==$, $!=$) podem ser usados para comparar dois ponteiros **do mesmo tipo**.
- Exemplo: (percorrendo um array)

Ponteiros e operadores relacionais

- Os operadores relacionais ($>$, $<$, $>=$, $<=$, $==$, $!=$) podem ser usados para comparar dois ponteiros **do mesmo tipo**.
- Exemplo: (percorrendo um array)

```
1  #include <stdio.h> // prog46.c
2
3  int main() {
4      int array[5] = { 0,1,2,3,4 };
5      int *b = array, *e = array + 5;
6
7      while (b < e) { // imprime os elementos do array
8          printf("%d ", *b);
9          b++;
10     }
11   printf("\n");
12   return 0;
13 }
```

Ponteiros e structs

Ponteiros e estruturas

- Ao criarmos uma variável de um tipo `struct`, esta é armazenada na memória como qualquer outra variável, e portanto possui um endereço.

- É possível então criar um ponteiro para uma variável de um tipo `struct`.

Ponteiros e estruturas

- Queremos acessar os campos de uma variável `struct` via um ponteiro.

Ponteiros e estruturas

- Queremos acessar os campos de uma variável `struct` via um ponteiro.

- **Jeito menos usual**: Para fazer isso, podemos utilizar o operador **(*)** juntamente com o operador **(.)** como de costume:

```
1 struct Ponto {
2   float x;
3   float y;
4 };
5
6 struct Ponto p1, *p3;
7 p3 = &p1;
8 (*p3).x = 1.5;
9 (*p3).y = 1.5;
```

Ponteiros e estruturas

- A forma mais comum de acessar os campos de uma variável `struct` via um ponteiro é essa:

Ponteiros e estruturas

- A forma mais comum de acessar os campos de uma variável `struct` via um ponteiro é essa:

- Em C também podemos usar o operador (`->`) para acessar campos de uma estrutura via um ponteiro.

```
1 struct Ponto {
2   float x;
3   float y;
4 };
5
6 struct Ponto p1, *p3;
7 p3 = &p1;
8 p3->x = 1.5;
9 p3->y = 1.5;
```

O que será impresso pelo programa abaixo?


```c
#include <stdio.h> // prog98.c

struct Ponto {
  double x;
  double y;
};

int main() {
  struct Ponto p1, *p2;
  p2 = &p1;

  p1.x = 1;
  p1.y = 2;

  p2->x = 4.5;
  p2->y = 6.5;

  printf("p1 = (%lf,%lf)\n", p1.x, p1.y);
}
```

Exercício

Um ponto no plano cartesiano é definido pela sua coordenada `x` e sua coordenada `y`. Seja `Ponto` um struct com dois campos `x` e `y`, do tipo `float`.

- Implemente uma função que recebe como argumento os endereços de dois `Pontos` e troca os valores das coordenadas dos pontos correspondentes.
- Sua função deve obedecer o protótipo:
  `void troca(Ponto *p1, Ponto *p2);`
- Implemente uma função que recebe um valor do tipo `Ponto` como argumento e dobra os valores das suas coordenadas.

Exemplo 1 — Ponteiros e Estruturas


```c
#include <stdio.h> // prog99.c

typedef struct {
  float x;
  float y;
} Ponto;

void troca(Ponto *p1, Ponto *p2) {
  Ponto aux;
  aux = *p1;
  *p1 = *p2;
  *p2 = aux;
}

int main() {
  Ponto a = {2, 3}, b = {4.5, 4.5};
  troca(&a, &b);
  // O que sera impresso?
  printf("a = (%f,%f)\n", a.x, a.y);
  // O que sera impresso?
  printf("b = (%f,%f)\n", b.x, b.y);
}
```

Exemplo 2 — Ponteiros e Estruturas

Equivalente ao exemplo anterior:

```
1  #include <stdio.h> // prog100.c
2
3  typedef struct {
4    float x;
5    float y;
6  } Ponto;
7
8  void dobraCoordenada(Ponto *p) {
9    p->x = 2 * p->x;
10   p->y = 2 * p->y;
11 }
12
13 int main() {
14   Ponto a = {2, 3};
15   dobraCoordenada(&a);
16
17   // O que sera impresso?
18   printf("a = (%f,%f)\n", a.x, a.y);
19 }
```

Ponteiros genéricos

Ponteiros genéricos

- Normalmente, um ponteiro aponta para um tipo específico de dado. Porém, pode-se criar um ponteiro genérico.

- Um ponteiro genérico pode apontar para todos os tipos de dados existentes ou que ainda serão criados.

Ponteiros genéricos

- Normalmente, um ponteiro aponta para um tipo específico de dado. Porém, pode-se criar um ponteiro genérico.

- Um ponteiro genérico pode apontar para todos os tipos de dados existentes ou que ainda serão criados.

- Em C, a declaração de um ponteiro genérico segue esta forma geral:

```
void * nome_do_ponteiro;
```

Ponteiro genérico — Exemplo


```c
#include <stdio.h> // prog28.c

int main() {
  void *pp;  // ponteiro genérico
  float x = 23.45;

  // recebe o endereço de um float
  pp = &x;
  printf("Endereco em pp: %p\n", pp);

  // imprime valor apontado por pp
  printf("Valor apontado: %f\n", *((float*)pp));

  return 0;
}
```

Ponteiro genérico — Vantagem e Desvantagem

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Vantagem do ponteiro genérico: permite guardar o endereço de qualquer tipo de dado.

- Desvantagem: sempre que tivermos de acessar o conteúdo da variável apontada pelo ponteiro genérico, será necessário utilizar o operador de typecast sobre ele antes de acessar o seu conteúdo.

- Exemplo:

```
void *pp;
float num = 23;
pp = &num;
printf("%f", *(float *) pp);
```

Ponteiros genéricos — Exemplo


```c
#include <stdio.h> // prog29.c

int main() {
  void *pp;
  int p2 = 10;

  // ponteiro genérico recebe o endereço de um inteiro
  pp = &p2;

  // tenta acessar o conteúdo do ponteiro genérico
  printf("Conteudo: %d\n", *pp); // ERRO

  //converte o ponteiro genérico pp para (int *)
  // antes de acessar seu conteúdo.
  printf("Conteudo: %d\n", *(int*) pp); // CORRETO

  return 0;
}
```

Alocação Dinâmica de Memória

Alocação Estática de Memória

- Sempre que escrevemos um programa, é preciso reservar espaço para os dados que serão processados. Usamos as variáveis para isso.

Alocação Estática de Memória

- Sempre que escrevemos um programa, é preciso reservar espaço para os dados que serão processados. Usamos as variáveis para isso.

- ☹ Bad News: Infelizmente, nem sempre é possível saber o quanto de memória um programa vai precisar.

Alocação estática versus Alocação dinâmica

Mudança de paradigma: Alocação dinâmica

- A memória é alocada em tempo de execução, via comandos, quando é necessária.

Alocação estática versus Alocação dinâmica

Mudança de paradigma: Alocação dinâmica

- A memória é alocada em tempo de execução, via comandos, quando é necessária.

- As estruturas de dados podem crescer e diminuir para se adequar às mudanças da quantidade de dados armazenada (aloca e desaloca quando necessário).

Alocação estática versus Alocação dinâmica

Mudança de paradigma: Alocação dinâmica

- A memória é alocada em tempo de execução, via comandos, quando é necessária.
- As estruturas de dados podem crescer e diminuir para se adequar às mudanças da quantidade de dados armazenada (aloca e desaloca quando necessário).
- É necessário muito cuidado para gerenciar a memória alocada para:
  - não danificar dados;
  - não deixar memória alocada perdida.

Alocação Dinâmica de Memória

Alocação dinâmica

A **alocação dinâmica** consiste em requisitar um espaço de memória ao computador, em tempo de execução. Se houver memória disponível, o computador então devolve para o programa o endereço do início do espaço alocado. Se não houver memória disponível, o computador devolve `NULL`.

Organização da memória

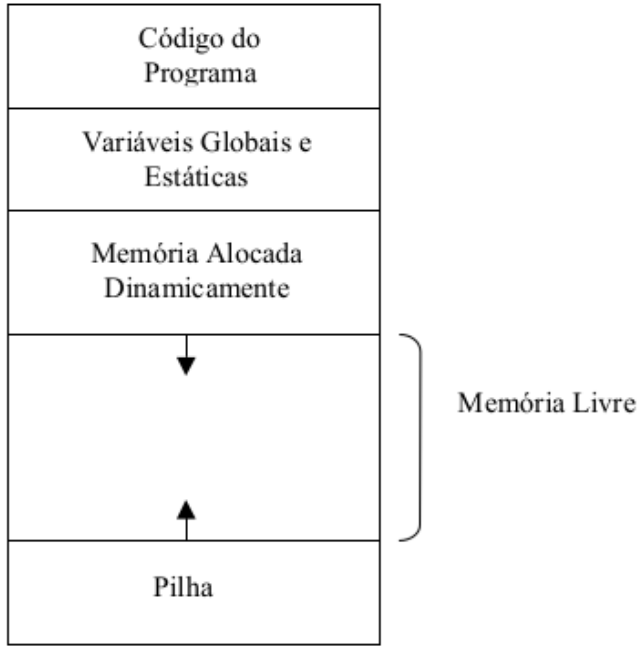
Os programas escritos em C, enxergam e dividem a memória em três diferentes regiões

Organização da memória

Os programas escritos em C, enxergam e dividem a memória em três diferentes regiões

Static: persiste durante toda a vida do programa e é geralmente usada para armazenar o código fonte, variáveis globais e variáveis `static`.

Stack (Pilha): usada para armazenar variáveis locais. É gerenciada automaticamente pela CPU. Quando uma função termina a execução, todas as variáveis associadas a essa função na pilha são excluídas e a memória que elas usam é liberada.

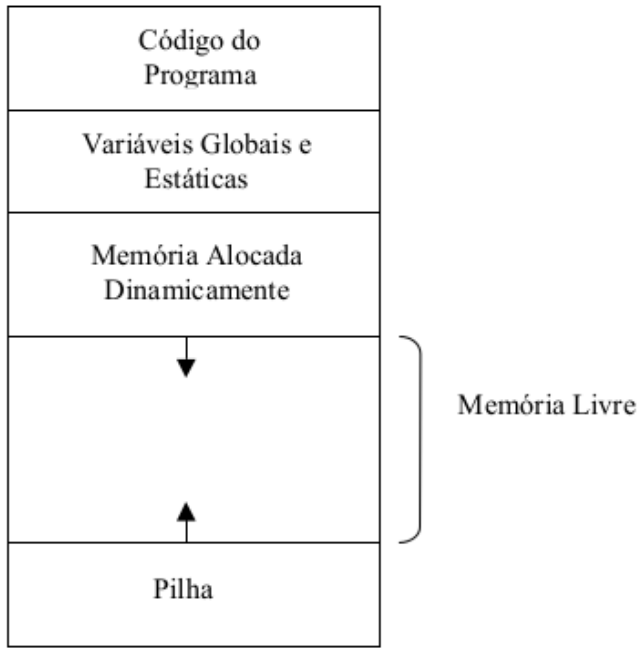


Organização da memória

Os programas escritos em C, enxergam e dividem a memória em três diferentes regiões

Static: persiste durante toda a vida do programa e é geralmente usada para armazenar o código fonte, variáveis globais e variáveis `static`.

Stack (Pilha): usada para armazenar variáveis locais. É gerenciada automaticamente pela CPU. Quando uma função termina a execução, todas as variáveis associadas a essa função na pilha são excluídas e a memória que elas usam é liberada.



Heap: memória livre disponível que não é gerenciada automaticamente pelo compilador — o programador deve alocar e desalocar explicitamente, usando funções como `malloc` e `free`.

Funções para alocação de Memória

A linguagem C possui pelo menos quatro funções para o sistema de alocação dinâmica, disponíveis na biblioteca `stdlib.h`:

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

Juntamente com essas funções geralmente utiliza-se o operador `sizeof`, que retorna o número de bytes de uma variável ou tipo de dado.

O operador `sizeof`

No momento da alocação de memória deve-se levar em conta o tamanho do dado alocado.

- `sizeof`: quando aplicado a um tipo de dado ou variável, retorna o seu tamanho em **bytes** (geralmente como um `long unsigned int`).

- Ele pode ser usado de três formas:

  - `sizeof (nome_do_tipo)`

  - `sizeof (nome_da_variavel)`

  - `sizeof nome_da_variavel`

Função `malloc`

malloc

Usada para alocar memória durante a execução do programa.

```
void* malloc (unsigned int bytes);
```

- A função `malloc` possui o parâmetro `bytes`, que é o tamanho (em bytes) do espaço de memória a ser alocado. Ela retorna um ponteiro para a primeira posição do array alocado; ou retorna `NULL` em caso de erro.

Função `malloc`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

malloc

Usada para alocar memória durante a execução do programa.

```
void* malloc (unsigned int bytes);
```

- A função `malloc` possui o parâmetro `bytes`, que é o tamanho (em bytes) do espaço de memória a ser alocado. Ela retorna um ponteiro para a primeira posição do array alocado; ou retorna `NULL` em caso de erro.

- Como a função malloc retorna um **ponteiro genérico** (`void*`), esse ponteiro pode ser atribuído a qualquer tipo de ponteiro via *type cast*.

Uso da função `malloc`

- DICA: É importante sempre testar se foi possível fazer a alocação de memória.

Uso da função `malloc`

- DICA: É importante sempre testar se foi possível fazer a alocação de memória.

- A função `malloc` retorna um ponteiro NULL para indicar que não há memória disponível no computador ou que ocorreu um erro que impediu a memória de ser alocada

```
1   ptr = (int*) malloc( MAX * sizeof(int) );
2
3   // verifica se a alocação foi mal-sucedida e então
4   // aborta o programa
5   if(ptr == NULL) {
6     printf("erro: memória insuficiente!\n");
7     exit(EXIT_FAILURE); // aborta o programa
8   }
9
```

Uso da função `malloc`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Lembre-se: No momento da alocação da memória, deve-se levar em conta o tamanho do dado alocado.

Uso da função `malloc`

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Lembre-se: No momento da alocação da memória, deve-se levar em conta o tamanho do dado alocado.

```
1  #include <stdio.h> // prog04.c
2  #include <stdlib.h>
3
4  int main() {
5    char *cPtr;
6    // aloca espaço para 1000 chars
7    cPtr = (char*) malloc(1000);
8
9    int *iPtr;
10   // aloca espaço para 250 inteiros
11   iPtr = (int*) malloc(1000);
12 }
```

Uso da função `malloc`

- Lembre-se: No momento da alocação da memória, deve-se levar em conta o tamanho do dado alocado.

```
1  #include <stdio.h> // prog04.c
2  #include <stdlib.h>
3
4  int main() {
5    char *cPtr;
6    // aloca espaço para 1000 chars
7    cPtr = (char*) malloc(1000);
8
9    int *iPtr;
10   // aloca espaço para 250 inteiros
11   iPtr = (int*) malloc(1000);
12 }
```

- Alocar 1000 bytes de memória equivale a um número de elementos diferentes, dependendo do tipo do elemento. Daí a importância de usar o operador `sizeof`.

Função `free`

- As variáveis alocadas dinamicamente continuam a existir mesmo depois que a execução da função termina.
- É nececessário liberar toda memória alocada dinamicamente antes do término da execução do programa.
- Para isso, usa-se a função `free`.

Função `free`

free

Usada para liberar memória que foi alocada durante a execução do programa.

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

- A função `free` possui o parâmetro `ptr`, que é um ponteiro para uma região de memória alocada dinamicamente. Ela libera os bytes de memória para os quais `ptr` aponta.

Função `free`

free

Usada para liberar memória que foi alocada durante a execução do programa.

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

- A função `free` possui o parâmetro `ptr`, que é um ponteiro para uma região de memória alocada dinamicamente. Ela libera os bytes de memória para os quais `ptr` aponta.

- **Atenção:** não pode ser usado para liberar memória que não foi alocada dinamicamente.

Exemplo — Uso da função `free`


```c
#include <stdio.h> // prog05.c
#include <stdlib.h>

int main() {
  int *ptr;

  ptr = (int*) malloc( 50 * sizeof(int) );

  if(ptr == NULL) {
    printf("erro: memória insuficiente!\n");
    exit(1);
  }

  for(int i = 0; i < 50; i++)
    ptr[i] = i + 1;

  for(int i = 0; i < 50; i++)
    printf("%d\n", ptr[i]);

  free(ptr); // libera memória alocada
  return 0;
}
```

Observações sobre a função `free`

- Como o programa sabe quantos bytes devem ser liberados?

Observações sobre a função `free`

- Como o programa sabe quantos bytes devem ser liberados?
  - Quando se aloca memória o programa guarda o número de bytes alocados numa "**tabela de alocação**" interna.

Observações sobre a função `free`

- Como o programa sabe quantos bytes devem ser liberados?
  - Quando se aloca memória o programa guarda o número de bytes alocados numa "**tabela de alocação**" interna.

- Apenas libere a memória quando tiver certeza de que ela não será mais usada.
  - Do contrário, um erro pode ocorrer, ou o programa pode não funcionar como esperado.

Observações sobre a função `free`

- Como o programa sabe quantos bytes devem ser liberados?
  - Quando se aloca memória o programa guarda o número de bytes alocados numa "**tabela de alocação**" interna.

- Apenas libere a memória quando tiver certeza de que ela não será mais usada.
  - Do contrário, um erro pode ocorrer, ou o programa pode não funcionar como esperado.

- Convém não deixar ponteiros soltos (*dangling pointers*) no programa. Portanto, depois de chamar a função `free`, atribua `NULL` ao ponteiro. Por exemplo:

```
free(ptr);
ptr = NULL;
```

Função `calloc`

calloc

Aloca memória dinamicamente e inicializa todos os bits do espaço alocado com 0.

```
void *calloc (unsigned int qtd, unsigned int size);
```

- A função `calloc` possui dois parâmetros:

Função `calloc`

calloc

Aloca memória dinamicamente e inicializa todos os bits do espaço alocado com 0.

```
void *calloc (unsigned int qtd, unsigned int size);
```

- A função `calloc` possui dois parâmetros:
  - `qtd`: número de elementos do array a ser alocado.

Função `calloc`

calloc

Aloca memória dinamicamente e inicializa todos os bits do espaço alocado com 0.

```
void *calloc (unsigned int qtd, unsigned int size);
```

- A função `calloc` possui dois parâmetros:
  - `qtd`: número de elementos do array a ser alocado.
  - `size`: o tamanho de cada elemento do array.

Função `calloc`

calloc

Aloca memória dinamicamente e inicializa todos os bits do espaço alocado com 0.

```
void *calloc (unsigned int qtd, unsigned int size);
```

- A função `calloc` possui dois parâmetros:
  - `qtd`: número de elementos do array a ser alocado.
  - `size`: o tamanho de cada elemento do array.

- Ela devolve um ponteiro para a primeira posição do array alocado; ou retorna `NULL` em caso de erro.

Exemplo — malloc versus calloc


```c
#include <stdio.h> // prog07.c
#include <stdlib.h>
#define MAX 10

int main() {
    int *ptr1, *ptr2;

    ptr1 = (int*) malloc(MAX * sizeof(int)); // malloc
    ptr2 = (int*) calloc(MAX, sizeof(int));  // calloc

    if(ptr2 != NULL && ptr1 != NULL) {
        for(int i = 0; i < MAX; i++)
            printf("%d ", ptr1[i]);
        printf("\n");
        for(int j = 0; j < MAX; j++)
            printf("%d ", ptr2[j]);
        printf("\n");
    }
    else printf("erro: memória insuficiente.\n");
  free(ptr1);
    free(ptr2);
  return 0;
}
```

Atividade

Escreva um programa que aloca dinamicamente um array de inteiros de tamanho n. O valor n é entrado pelo usuário no início do programa.

(a) Escreva uma função que recebe como parâmetro o array alocado dinamicamente e preenche esse array com valores inteiros digitados pelo usuário. Sua função deve obedecer o protótipo:
`void preencheArray(int *A, int n);`

(b) Escreva uma função que recebe como parâmetro o array alocado dinamicamente e imprime na tela os seus elementos. Sua função deve obedecer o protótipo:
`void imprimeArray(int *A, int n);`

(c) Antes do programa acabar, libere a memória alocada dinamicamente.

Vazamento de memória

- Vazamentos de memória acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.

Vazamento de memória

- Vazamentos de memória acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.

Vazamento de memória

- Vazamentos de memória acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.
  - O sistema operacional também não pode usar essa memória, pois considera que ela ainda está sendo usada pelo seu programa.

Vazamento de memória

- Vazamentos de memória acontecem quando o programa perde o endereço de uma região de memória alocada dinamicamente antes de devolvê-lo ao sistema operacional.
  - Quando isso acontece, seu programa não pode excluir a memória alocada dinamicamente porque não sabe mais onde ela está.
  - O sistema operacional também não pode usar essa memória, pois considera que ela ainda está sendo usada pelo seu programa.
- Exemplo: que problema pode haver com esse trecho de código?

```
1 void funcao() {
2     int *ptr = (int*) malloc(10 * sizeof(int));
3 }
```

Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;
2 int *ptr = (int*) malloc(10*sizeof(int)); // aloca memoria
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;
2 int *ptr = (int*) malloc(10*sizeof(int)); // aloca memoria
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

- Também é possível obter um vazamento de memória via alocação dupla:

```
1 int *ptr = (int*) malloc(10*sizeof(int));
2 ptr = (int*) malloc(7*sizeof(int));
```

Vazamento de memória

- Há outras formas de vazamento de memória. Por exemplo, um vazamento de memória pode ocorrer se um ponteiro que contém o endereço da memória alocada dinamicamente receber outro valor:

```
1 int v = 5;
2 int *ptr = (int*) malloc(10*sizeof(int)); // aloca memoria
3 ptr = &v; // endereco antigo perdido: vazamento de memoria
```

- Também é possível obter um vazamento de memória via alocação dupla:

```
1 int *ptr = (int*) malloc(10*sizeof(int));
2 ptr = (int*) malloc(7*sizeof(int));
```

- Uma forma de evitar esse tipo de vazamento consiste em liberar a memória antes de atribuir novo valor ao ponteiro:

```
1 int *ptr = (int*) malloc(5*sizeof(int));
2 free(ptr);
3 ptr = (int*) malloc(6*sizeof(int));
```

Aplicações de ponteiros

Aplicações de ponteiros

- Passar argumentos por referência. A passagem por referência serve a dois propósitos:

Aplicações de ponteiros

- Passar argumentos por referência. A passagem por referência serve a dois propósitos:

  (i) Modificar o valor de uma variável externa dentro de uma função.
  Exemplo: trocar os valores de duas variáveis.

Aplicações de ponteiros

- Passar argumentos por referência. A passagem por referência serve a dois propósitos:

  (i) Modificar o valor de uma variável externa dentro de uma função.
  Exemplo: trocar os valores de duas variáveis.

  (ii) Eficiência: podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.

Aplicações de ponteiros

- Passar argumentos por referência. A passagem por referência serve a dois propósitos:
  (i) Modificar o valor de uma variável externa dentro de uma função.
  Exemplo: trocar os valores de duas variáveis.

  (ii) Eficiência: podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.

- Acessar elementos de um array. O compilador usa internamente ponteiros para acessar os elementos de um array.

Aplicações de ponteiros

- Passar argumentos por referência. A passagem por referência serve a dois propósitos:
  (i) Modificar o valor de uma variável externa dentro de uma função.
  Exemplo: trocar os valores de duas variáveis.

  (ii) Eficiência: podemos passar um dado de tamanho grande (por exemplo, um array) para uma função de forma que não envolva a cópia dos dados.

- Acessar elementos de um array. O compilador usa internamente ponteiros para acessar os elementos de um array.

- Permitir que uma função "retorne" vários valores.

Aplicações de ponteiros

- Programar no nível do sistema, onde os endereços de memória são úteis.

Aplicações de ponteiros

- Programar no nível do sistema, onde os endereços de memória são úteis.
- Alocação dinâmica de memória: podemos usar ponteiros para alocar memória dinamicamente. A vantagem da memória alocada dinamicamente é que ela não é excluída até que seja excluída explicitamente.

Aplicações de ponteiros

- Programar no nível do sistema, onde os endereços de memória são úteis.
- Alocação dinâmica de memória: podemos usar ponteiros para alocar memória dinamicamente. A vantagem da memória alocada dinamicamente é que ela não é excluída até que seja excluída explicitamente.
- Implementar diversas estruturas de dados. Eles serão usados na implementação eficiente de diversas estruturas de dados que veremos durante o curso.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

FIM