Ponteiros em C

Adriano Joaquim de Oliveira Cruz

Instituto de Matemática
Departamento de Ciência da Computação
UFRJ

29 de outubro de 2013

Section Summary

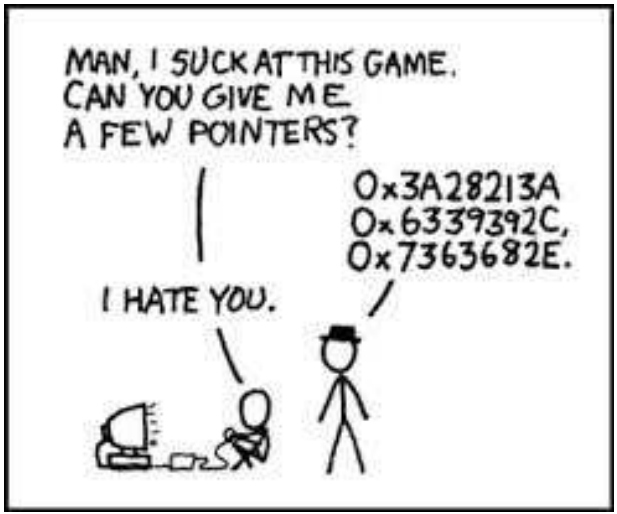
Bibliografia

1. Adriano Cruz. *Curso de Linguagem* C, Disponível em `http://equipe.nce.ufrj.br/adriano`

Bibliografia

1. Adriano Cruz. *Curso de Linguagem* `C`, Disponível em `http://equipe.nce.ufrj.br/adriano`
2. Ulysses de Oliveira. *Programando em* `C`, Editora Ciência Moderna.

Ponteiros?


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
I HATE YOU.
0x3A28213A
0x6339392C,
0x7363682E.

Ponteiros (Endereços) apontam

lugares na memória

para buscarmos dados.

Quando usar?

- Ponteiros são importantes quando se deseja que uma função retorne mais de um valor.

Quando usar?

- Ponteiros são importantes quando se deseja que uma função retorne mais de um valor.
- A função deve receber como argumentos não os valores dos parâmetros mas sim ponteiros que apontem para seus endereços.

Quando usar?

- Ponteiros são importantes quando se deseja que uma função retorne mais de um valor.
- A função deve receber como argumentos não os valores dos parâmetros mas sim ponteiros que apontem para seus endereços.
- Com os endereços é possível modificar diretamente os conteúdos destas variáveis.

Quando usar?

- Outra aplicação é apontar para áreas de memória que devem ser gerenciadas durante a execução do programa.

Quando usar?

- Outra aplicação é apontar para áreas de memória que devem ser gerenciadas durante a execução do programa.
- Por exemplo, vetores de tamanhos desconhecidos.

Quando usar?

- Outra aplicação é apontar para áreas de memória que devem ser gerenciadas durante a execução do programa.
- Por exemplo, vetores de tamanhos desconhecidos.
- Com ponteiros, é possível reservar as posições de memória necessárias para estas áreas.

Section Summary

Definindo

Antes de serem usados os ponteiros, como as variáveis, precisam ser declarados. A forma geral da declaração de um ponteiro é a seguinte:

Definindo

Antes de serem usados os ponteiros, como as variáveis, precisam ser declarados. A forma geral da declaração de um ponteiro é a seguinte:

```
tipo *nome;
```

Definindo

Antes de serem usados os ponteiros, como as variáveis, precisam ser declarados. A forma geral da declaração de um ponteiro é a seguinte:

```
tipo *nome;
```

Onde **tipo** é qualquer tipo válido em `C` e `nome` é o **nome** da variável ponteiro.

Definindo

Antes de serem usados os ponteiros, como as variáveis, precisam ser declarados. A forma geral da declaração de um ponteiro é a seguinte:

```
tipo *nome;
```

Onde **tipo** é qualquer tipo válido em `C` e `nome` é o **nome** da variável ponteiro.

O asterisco indica que a variável irá armazenar endereço e não dado.

Exemplos

```
int *res;     /* ponteiro para inteiro */
float *div;   /* ponteiro para ponto flutuante */
char *c;      /* ponteiro para caractere */
```

Inicializando

- Como as variáveis, os ponteiros devem ser inicializados antes de serem usados.

Inicializando

- Como as variáveis, os ponteiros devem ser inicializados antes de serem usados.
- Esta inicialização pode ser feita na declaração ou através de uma atribuição.

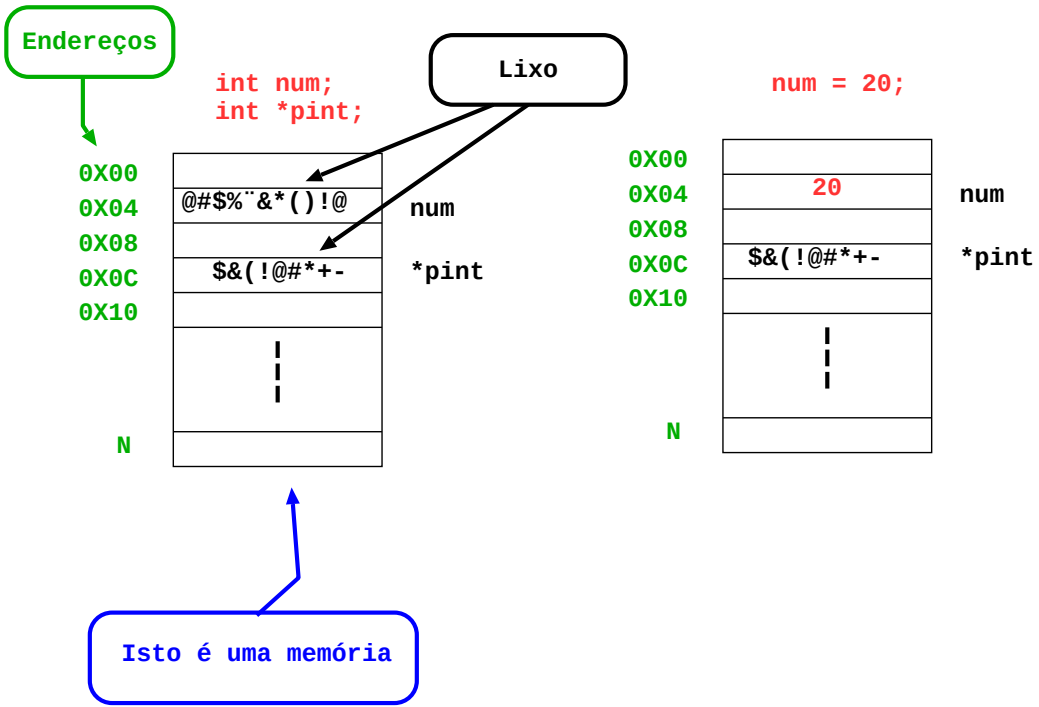
Inicializando

- Como as variáveis, os ponteiros devem ser inicializados antes de serem usados.
- Esta inicialização pode ser feita na declaração ou através de uma atribuição.
- Após a declaração o que temos é um espaço na memória reservado para armazenamento de endereços.

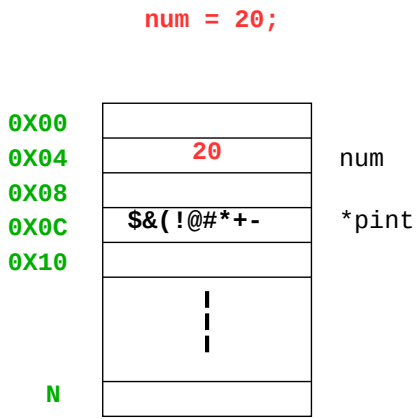
Inicializando

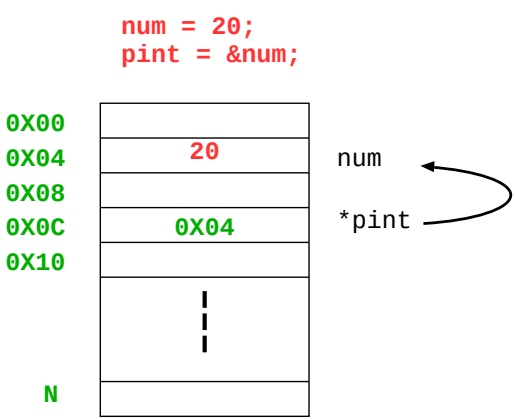
- Como as variáveis, os ponteiros devem ser inicializados antes de serem usados.
- Esta inicialização pode ser feita na declaração ou através de uma atribuição.
- Após a declaração o que temos é um espaço na memória reservado para armazenamento de endereços.
- O valor inicial da memória é indefinido como acontece com variáveis.

Variáveis que ...



Variáveis que armazenam endereços ...





Operadores Especiais

- Existem dois operadores especiais para ponteiros:

Operadores Especiais

- Existem dois operadores especiais para ponteiros:
- * e &.

Operadores Especiais

- Existem dois operadores especiais para ponteiros:
- * e &.
- Os dois operadores são unários, isto é requerem somente um operando.

Operadores Especiais

- Existem dois operadores especiais para ponteiros:
- * e &.
- Os dois operadores são unários, isto é requerem somente um operando.
- O operador & devolve o endereço de memória do seu operando.

Operadores Especiais

- Existem dois operadores especiais para ponteiros:
- * e &.
- Os dois operadores são unários, isto é requerem somente um operando.
- O operador & devolve o endereço de memória do seu operando.
- O operador * devolve o valor da variável localizada no endereço apontado pelo ponteiro.

Exemplo

```c
#include <stdio.h>
int main (void) {
    int i = 10;
    int *p;

    p = &i;
    *p = *p + 1;
    printf("%d\n", i);

    return 0;
}
```

Atribuições com ponteiros

- Da mesma maneira que ocorre com uma variável comum, o conteúdo de um ponteiro pode ser passado para outro ponteiro do mesmo tipo.

Atribuições com ponteiros

- Da mesma maneira que ocorre com uma variável comum, o conteúdo de um ponteiro pode ser passado para outro ponteiro do mesmo tipo.
- Por exemplo, uma variável ponteiro declarada como apontador de dados inteiros deve sempre apontar para dados deste tipo.

Atribuições com ponteiros

- Da mesma maneira que ocorre com uma variável comum, o conteúdo de um ponteiro pode ser passado para outro ponteiro do mesmo tipo.
- Por exemplo, uma variável ponteiro declarada como apontador de dados inteiros deve sempre apontar para dados deste tipo.
- Observar que em `C` é possível atribuir qualquer endereço a uma variável ponteiro. Deste modo é possível atribuir o endereço de uma variável do tipo **float** a um ponteiro do tipo **int**.

Atribuições com ponteiros

- Da mesma maneira que ocorre com uma variável comum, o conteúdo de um ponteiro pode ser passado para outro ponteiro do mesmo tipo.
- Por exemplo, uma variável ponteiro declarada como apontador de dados inteiros deve sempre apontar para dados deste tipo.
- Observar que em `C` é possível atribuir qualquer endereço a uma variável ponteiro. Deste modo é possível atribuir o endereço de uma variável do tipo **float** a um ponteiro do tipo **int**.
- No entanto, o programa não irá funcionar da maneira correta.

Exemplo

```c
#include <stdio.h>
int main(void) {
    int vetor[] = { 10, 20, 30, 40, 50 };
    int *p1, *p2;
    int i = 100;

    p1 = &vetor[2];
    printf("%d\n", *p1);
    p2 = &i;
    printf("%d\n", *p2);
    p2 = p1;
    printf("%d\n", *p2);
    return 0;
}
```

Incrementando/Decrementando ponteiros

```
int main(void) {
    int vetor[] = { 10, 20, 30, 40, 50 };
    int *p1;

    p1 = &vetor[2];
    printf("%d\n", *p1);
    p1++;
    printf("%d\n", *p1);
    p1 = p1 + 1;
    printf("%d\n", *p1);
    return 0;
}
```

Incrementando/Decrementando ponteiros

- Um ponteiro para um número inteiro, que é armazenado em quatro bytes, é incrementado por um e aponta para o próximo número inteiro.

Incrementando/Decrementando ponteiros

- Um ponteiro para um número inteiro, que é armazenado em quatro bytes, é incrementado por um e aponta para o próximo número inteiro.
- Não deveria ser de 4?

Incrementando/Decrementando ponteiros

- Um ponteiro para um número inteiro, que é armazenado em quatro bytes, é incrementado por um e aponta para o próximo número inteiro.
- Não deveria ser de 4?
- O compilador interpreta o comando `p1++` como: *passe a apontar para o próximo número inteiro* e, portanto, aumenta automaticamente o endereço do número de bytes correto.

Incrementando/Decrementando ponteiros

- Um ponteiro para um número inteiro, que é armazenado em quatro bytes, é incrementado por um e aponta para o próximo número inteiro.
- Não deveria ser de 4?
- O compilador interpreta o comando `p1++` como: *passe a apontar para o próximo número inteiro* e, portanto, aumenta automaticamente o endereço do número de bytes correto.
- Este ajuste é feito de acordo com o tipo do operando que o ponteiro está apontando.

Incrementando/Decrementando ponteiros

- Um ponteiro para um número inteiro, que é armazenado em quatro bytes, é incrementado por um e aponta para o próximo número inteiro.
- Não deveria ser de 4?
- O compilador interpreta o comando `p1++` como: *passe a apontar para o próximo número inteiro* e, portanto, aumenta automaticamente o endereço do número de bytes correto.
- Este ajuste é feito de acordo com o tipo do operando que o ponteiro está apontando.
- Do mesmo modo, somar três a um ponteiro faz com que ele passe apontar para o terceiro elemento após o atual.

Operações Aritméticas

- A diferença entre ponteiros fornece quantos elementos do tipo do ponteiro existem entre os dois ponteiros.

```
#include <stdio.h>
int main(void) {
    float vetor[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
    float *p1, *p2;

    p1 = &vetor[2]; /* endereco do terceiro elemento↩
        */
    p2 = vetor; /* endereco do primeiro elemento */
    printf("Diferenca entre ponteiros %d\n", p1-p2);
    return 0;
}
```

Operações Aritméticas

- A diferença entre ponteiros fornece quantos elementos do tipo do ponteiro existem entre os dois ponteiros.
- No exemplo é impresso o valor 2.

```
#include <stdio.h>
int main(void) {
    float vetor[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
    float *p1, *p2;

    p1 = &vetor[2]; /* endereco do terceiro elemento */
    p2 = vetor; /* endereco do primeiro elemento */
    printf("Diferenca entre ponteiros %d\n", p1-p2);
    return 0;
}
```

Operações Aritméticas: não usar

- Não é possível multiplicar ou dividir ponteiros.

Operações Aritméticas: não usar

- Não é possível multiplicar ou dividir ponteiros.
- Não se pode adicionar ou subtrair o tipo **float** ou o tipo **double** a ponteiros.

Operações Aritméticas: não usar

- Não é possível multiplicar ou dividir ponteiros.
- Não se pode adicionar ou subtrair o tipo **float** ou o tipo **double** a ponteiros.
- É possível comparar ponteiros em uma expressão relacional.

Operações Aritméticas: não usar

- Não é possível multiplicar ou dividir ponteiros.
- Não se pode adicionar ou subtrair o tipo **float** ou o tipo **double** a ponteiros.
- É possível comparar ponteiros em uma expressão relacional.
- No entanto, só é possível comparar ponteiros de mesmo tipo.

Section Summary

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.
- O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor.

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.
- O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor.
- Este é um valor que não pode ser alterado.

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.
- O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor.
- Este é um valor que não pode ser alterado.
- A declaração **int** `vetor[100]` cria um vetor de inteiros de 100 posições e permite que algumas operações com ponteiros possam ser realizadas com a variável `vetor`.

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.
- O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor.
- Este é um valor que não pode ser alterado.
- A declaração **int** `vetor[100]` cria um vetor de inteiros de 100 posições e permite que algumas operações com ponteiros possam ser realizadas com a variável `vetor`.
- No entanto, existe uma diferença fundamental entre declarar um conjunto de dados como um vetor ou através de um ponteiro.

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.
- O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor.
- Este é um valor que não pode ser alterado.
- A declaração **int** `vetor[100]` cria um vetor de inteiros de 100 posições e permite que algumas operações com ponteiros possam ser realizadas com a variável `vetor`.
- No entanto, existe uma diferença fundamental entre declarar um conjunto de dados como um vetor ou através de um ponteiro.
- Na declaração de vetor, o compilador automaticamente reserva um bloco de memória para que o vetor seja armazenado.

Ponteiros e Vetores

- Ponteiros e Vetores estão fortemente relacionados na linguagem C.
- O nome de um vetor é um ponteiro que aponta para a primeira posição do vetor.
- Este é um valor que não pode ser alterado.
- A declaração **int** `vetor[100]` cria um vetor de inteiros de 100 posições e permite que algumas operações com ponteiros possam ser realizadas com a variável `vetor`.
- No entanto, existe uma diferença fundamental entre declarar um conjunto de dados como um vetor ou através de um ponteiro.
- Na declaração de vetor, o compilador automaticamente reserva um bloco de memória para que o vetor seja armazenado.
- Quando apenas um ponteiro é declarado a única coisa que o compilador faz é alocar um ponteiro para

Ponteiros e Vetores

```c
#include <stdio.h>
int main(void) {
    float v[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
    int i;
    for (i = 0; i < 7; i++) {
        printf("%.1f ", v[i]);
    }
    printf("\n");
    for (i = 0; i < 7; i++) {
        printf("%.1f ", *(v+i));
    }
    return 0;
}
```

Ponteiros e Vetores

- Para percorrer um vetor também é possível usar um ponteiro variável.

Ponteiros e Vetores

- Para percorrer um vetor também é possível usar um ponteiro variável.
- `int *pv;`

Ponteiros e Vetores

- Para percorrer um vetor também é possível usar um ponteiro variável.
- **int** `*pv;`
- Observar que o nome de um vetor de tamanho fixo é um ponteiro constante.

Ponteiros e Vetores

- Para percorrer um vetor também é possível usar um ponteiro variável.
- **int** `*pv;`
- Observar que o nome de um vetor de tamanho fixo é um ponteiro constante.
- **int** `v[10]`

Ponteiros e Vetores

- Para percorrer um vetor também é possível usar um ponteiro variável.
- **int** `*pv;`
- Observar que o nome de um vetor de tamanho fixo é um ponteiro constante.
- **int** `v[10]`
- Ponteiros constantes não podem ser alterados.

Ponteiros e Vetores

- Para percorrer um vetor também é possível usar um ponteiro variável.
- **int** `*pv;`
- Observar que o nome de um vetor de tamanho fixo é um ponteiro constante.
- **int** `v[10]`
- Ponteiros constantes não podem ser alterados.
- Ponteiros variáveis podem ser alterados.

Ponteiros Variáveis

```c
#include <stdio.h>
int main(void) {
    float v[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
    int i;
    float *p;

    for (i = 0; i < 7; i++) printf("%.1f ", v[i]);
    printf("\n");
    for (i = 0; i < 7; i++) printf("%.1f ", *(v+i));
    printf("\n");
    for (i = 0, p = v; i < 7; i++, p++) {
        printf("%.1f ", *p);
    }
    return 0;
}
```

Ponteiros Variáveis

```
#include<stdio.h>
int strcop(char *d, char *o);
int main(void) {
    char destino[20];
    char *origem="cadeia de caractere de origem";
    strcop(destino, origem);
    printf("%s\n", origem);
    printf("%s\n", destino);
    return 0;
}
int strcop(char *d, char *o) {
    while ((*d++ = *o++) != '\0');
    return 0;
}
```

Section Summary

Alocação Dinâmica

- O uso de ponteiros e vetores exige que após a definição da variável ponteiro uma área de memória deve ser reservada para armazenar os dados do vetor.

Alocação Dinâmica

- O uso de ponteiros e vetores exige que após a definição da variável ponteiro uma área de memória deve ser reservada para armazenar os dados do vetor.
- Para obter esta área o programa deve usar funções existentes na biblioteca `stdlib`.

Alocação Dinâmica

- O uso de ponteiros e vetores exige que após a definição da variável ponteiro uma área de memória deve ser reservada para armazenar os dados do vetor.
- Para obter esta área o programa deve usar funções existentes na biblioteca `stdlib`.
- Estas funções pedem ao sistema operacional para separar pedaços da memória e devolvem ao programa que pediu o endereço inicial deste local.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.
- O valor armazenado no espaço é indefinido.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.
- O valor armazenado no espaço é indefinido.
- A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.
- O valor armazenado no espaço é indefinido.
- A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.
- **void** `*calloc(size_t num, size_t size);` Reserva espaço na memória para um vetor de `num` itens do programa.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.
- O valor armazenado no espaço é indefinido.
- A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.
- **void** `*calloc(size_t num, size_t size);` Reserva espaço na memória para um vetor de `num` itens do programa.
- Cada item tem tamanho `size`.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.
- O valor armazenado no espaço é indefinido.
- A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.
- **void** `*calloc(size_t num, size_t size);` Reserva espaço na memória para um vetor de `num` itens do programa.
- Cada item tem tamanho `size`.
- Todos os bits do espaço são inicializados com 0.

Funções para Alocação

- As funções básicas de alocação de memória que iremos discutir são:
- **void** `*malloc(size_t size);` Reserva `size` bytes na memória para algum item de um programa.
- O valor armazenado no espaço é indefinido.
- A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.
- **void** `*calloc(size_t num, size_t size);` Reserva espaço na memória para um vetor de `num` itens do programa.
- Cada item tem tamanho `size`.
- Todos os bits do espaço são inicializados com 0.
- A função retorna um ponteiro de tipo **void** para o espaço reservado ou `NULL` no caso de algum erro ocorrer.

Funções para Desalocação

- **void** `free(`**void** `*pont);`

Funções para Desalocação

- **void** `free(`**void** `*pont);`
- O espaço apontado por `pont` é devolvido ao sistema para uso.

Funções para Desalocação

- **void** `free(`**void** `*pont);`
- O espaço apontado por `pont` é devolvido ao sistema para uso.
- Caso `pont` seja um ponteiro nulo nenhuma ação é executada.

Funções para Desalocação

- **void** `free(`**void** `*pont);`
- O espaço apontado por `pont` é devolvido ao sistema para uso.
- Caso `pont` seja um ponteiro nulo nenhuma ação é executada.
- No caso do ponteiro não ter sido resultado de uma reserva feita por meio de uma das funções `calloc`, `realloc` ou `malloc` o resultado é indefinido.

Mudando o tamanho do vetor

- `void realloc(void *pont, size_t size);`

Mudando o tamanho do vetor

- **void** `realloc(`**void** `*pont, size_t size);`
- A função altera o tamanho do objeto na memória apontado por `pont` para o tamanho especificado por `size`.

Mudando o tamanho do vetor

- **void** `realloc(`**void** `*pont, size_t size);`
- A função altera o tamanho do objeto na memória apontado por `pont` para o tamanho especificado por `size`.
- O conteúdo do objeto será mantido até um tamanho igual ao menor dos dois tamanhos, novo e antigo.

Mudando o tamanho do vetor

- **void** `realloc(`**void** `*pont, size_t size);`
- A função altera o tamanho do objeto na memória apontado por `pont` para o tamanho especificado por `size`.
- O conteúdo do objeto será mantido até um tamanho igual ao menor dos dois tamanhos, novo e antigo.
- Se o novo tamanho requerer movimento, o espaço reservado anteriormente é liberado.

Mudando o tamanho do vetor

- **void** `realloc(`**void** `*pont, size_t size);`
- A função altera o tamanho do objeto na memória apontado por `pont` para o tamanho especificado por `size`.
- O conteúdo do objeto será mantido até um tamanho igual ao menor dos dois tamanhos, novo e antigo.
- Se o novo tamanho requerer movimento, o espaço reservado anteriormente é liberado.
- Caso o novo tamanho for maior, o conteúdo da porção de memória reservada a mais ficará com um valor sem especificação. Se o tamanho `size` for igual a 0 e `pont` não é um ponteiro nulo o objeto previamente reservado é liberado.

Usando calloc

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    float *v; int i, tam;
    printf("Qual o tamanho do vetor? ");
    scanf("%d", &tam);
    v = calloc(tam, sizeof(float));
    if (!v) {
        return 1;
    }
    for (i=0; i<tam; i++) {
        printf("Elemento %d ?", i);
        scanf("%f", v+i);
        printf("Li valor %f \n", *(v+i));
    }
    free(v);
    return 0;
}
```

Ponteiros Variáveis

```c
#include <stdio.h>
#include <stdlib.h>
void LeVetor (float *v, int tam);
float  ProcuraMaior (float *v, int tam, int *vezes);
int main(void) {
    float *v, maior;
    int i, tam, vezes;
    printf("Qual o tamanho do vetor? ");
    scanf("%d", &tam);
    v = (float *) malloc(tam * sizeof(float));
    if (!v) return 1;
    LeVetor(v, tam);
    maior = ProcuraMaior (v, tam, &vezes);
    printf("Maior = %f e aparece %d vezes.\n.",
                                    maior, vezes);
    free(v);
    return 0;
}
```

Ponteiros Variáveis

```
void LeVetor (float *v, int tam)  {
     int i;
     for (i=0; i<tam; i++) {
          scanf("%f", v+i);
     }
}
float ProcuraMaior (float *v, int tam, int *vezes) {
     int i;
     float maior;
     maior = v[0]; *vezes = 1;
     for (i=1; i<tam; i++) {
          if (v[i] > maior) {
               maior = v[i];
               *vezes = 1;
          }
          else if (maior == v[i]) *vezes=*vezes+1;
     }
     return maior;
}
```

Section Summary

Matrizes e Ponteiros

- Um ponteiro aponta para uma área de memória que é endereçada de maneira linear.

Matrizes e Ponteiros

- Um ponteiro aponta para uma área de memória que é endereçada de maneira linear.
- Vetores podem ser facilmente manipulados com ponteiros.

Matrizes e Ponteiros

- Um ponteiro aponta para uma área de memória que é endereçada de maneira linear.
- Vetores podem ser facilmente manipulados com ponteiros.
- Em estruturas de dados com maior dimensionalidade, como matrizes, por exemplo, que são arranjos bidimensionais de dados, é necessário mapear o espaço bidimensional (ou de maior ordem) para uma dimensão.

Matrizes e Ponteiros

- Um ponteiro aponta para uma área de memória que é endereçada de maneira linear.
- Vetores podem ser facilmente manipulados com ponteiros.
- Em estruturas de dados com maior dimensionalidade, como matrizes, por exemplo, que são arranjos bidimensionais de dados, é necessário mapear o espaço bidimensional (ou de maior ordem) para uma dimensão.
- No caso das matrizes é necessário mapear o endereço de cada elemento na matriz, que é definido por um par *(linha, coluna)* em um endereço linear.

Matrizes e Ponteiros

- Considere uma matriz chamada `matriz` de tamanho `LIN,COL`

Matrizes e Ponteiros

- Considere uma matriz chamada `matriz` de tamanho `LIN,COL`
- Podemos usar uma solução que mapeie a matriz que é um objeto de duas dimensões em um vetor que tem apenas uma.

Matrizes e Ponteiros

- Considere uma matriz chamada `matriz` de tamanho `LIN,COL`
- Podemos usar uma solução que mapeie a matriz que é um objeto de duas dimensões em um vetor que tem apenas uma.
- Neste caso o programa deve fazer a translação de endereços toda vez que precisar ler ou escrever na matriz.

Matrizes e Ponteiros

- Considere uma matriz chamada `matriz` de tamanho `LIN,COL`
- Podemos usar uma solução que mapeie a matriz que é um objeto de duas dimensões em um vetor que tem apenas uma.
- Neste caso o programa deve fazer a translação de endereços toda vez que precisar ler ou escrever na matriz.
- A expressão `matriz+(i*COL+j)` calcula a posição do elemento `matriz[i][j]` a partir do primeiro elemento da matriz que está no endereço inicial `matriz`.

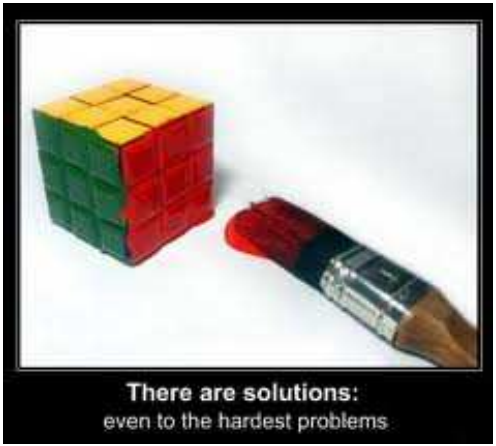
Matrizes que são vetores

```
#define LIN 3
#define COL 4

    int *matriz;
    int i, j;

    matriz = malloc(LIN*COL*sizeof(int));
    if (!matriz) {
        printf("Erro.\n");
        return 1;
    }
    for(i = 0; i < LIN; i++) {
        for (j = 0; j < COL; j++) {
            printf("Elemento %d %d = ", i, j);
            scanf("%d", matriz+(i*COL+j));
        }
    }
```

Solução não muito boa.



Section Summary

Um passo em frente

Vetores de Ponteiros

- Uma possibilidade mais interessante é utilizar vetores de ponteiros.

Vetores de Ponteiros

- Uma possibilidade mais interessante é utilizar vetores de ponteiros.
- Esta não é a notação ideal, mas é um passo na direção da notação mais efetiva.

Vetores de Ponteiros

- Uma possibilidade mais interessante é utilizar vetores de ponteiros.
- Esta não é a notação ideal, mas é um passo na direção da notação mais efetiva.
- Neste caso cada linha da matriz é apontada por um ponteiro armazenado no vetor de ponteiros.

Vetores de Ponteiros

```c
#define LIN 10
#define COL 60
int main(void) {
    char *l[LIN];  int i;
    for (i = 0; i < LIN; i++) {
        if (!(l[i] = malloc(COL*sizeof(char)))) {
            printf("Sem memória %d.\n", i);
            return i;
        }
    }
    for (i = 0; i < LIN; i++) {
        printf("Linha %d?\n", i);
        gets(l[i]);
    }
    for (i = 0; i < LIN; i++) {
        printf("Linha %d %s.\n", i, lin[i]);
    }
    return 0;
}
```

Section Summary

Ponteiros para ponteiros

- No exemplo anterior o número de linhas da matriz é fixa.

Ponteiros para ponteiros

- No exemplo anterior o número de linhas da matriz é fixa.
- Há uma mistura de notação de ponteiros com matrizes.

Ponteiros para ponteiros

- No exemplo anterior o número de linhas da matriz é fixa.
- Há uma mistura de notação de ponteiros com matrizes.
- Vamos considerar um exemplo onde tanto o número de linhas como o de colunas é desconhecido.

Ponteiros para ponteiros

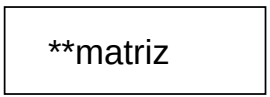
- No exemplo anterior o número de linhas da matriz é fixa.
- Há uma mistura de notação de ponteiros com matrizes.
- Vamos considerar um exemplo onde tanto o número de linhas como o de colunas é desconhecido.
- Agora vamos criar um vetor de ponteiros que irá armazenar o endereço inicial de cada linha.

Vetores de Ponteiros: Passo 1

Definindo um ponteiro para ponteiro: `matriz`.

```
int main (void) {
    int **matriz; /* ponteiro para os ponteiros */

return 0;
}
```



Vetores de Ponteiros: Passo 2

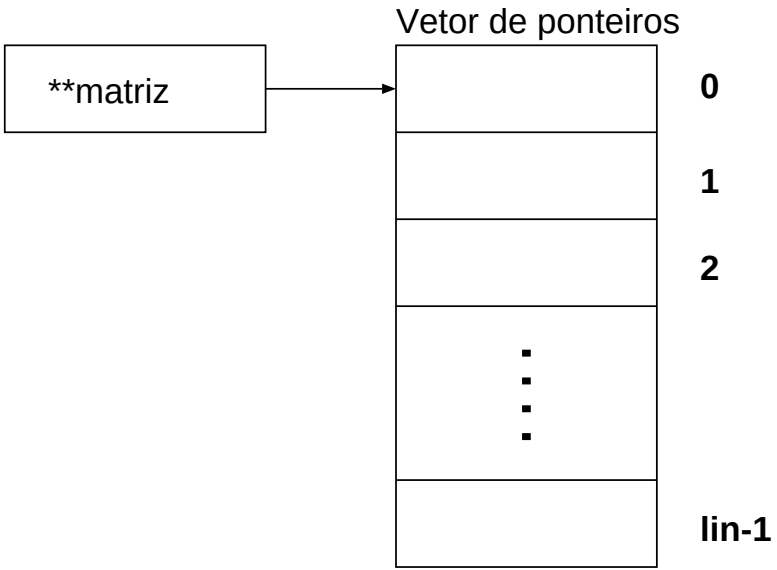
Criando um vetor de ponteiros. Precisamos saber o número de linhas.
Cada linha é um vetor.

```
int main (void) {
    int **matriz;
    int lin, col;

    scanf("%d", &lin);
    matriz=(int **)malloc(lin*sizeof(int *));
    if (!matriz) {
        puts("Nao há espaço.");
        return 1;
    }

    return 0;
}
```

Vetores de Ponteiros: Passo 2

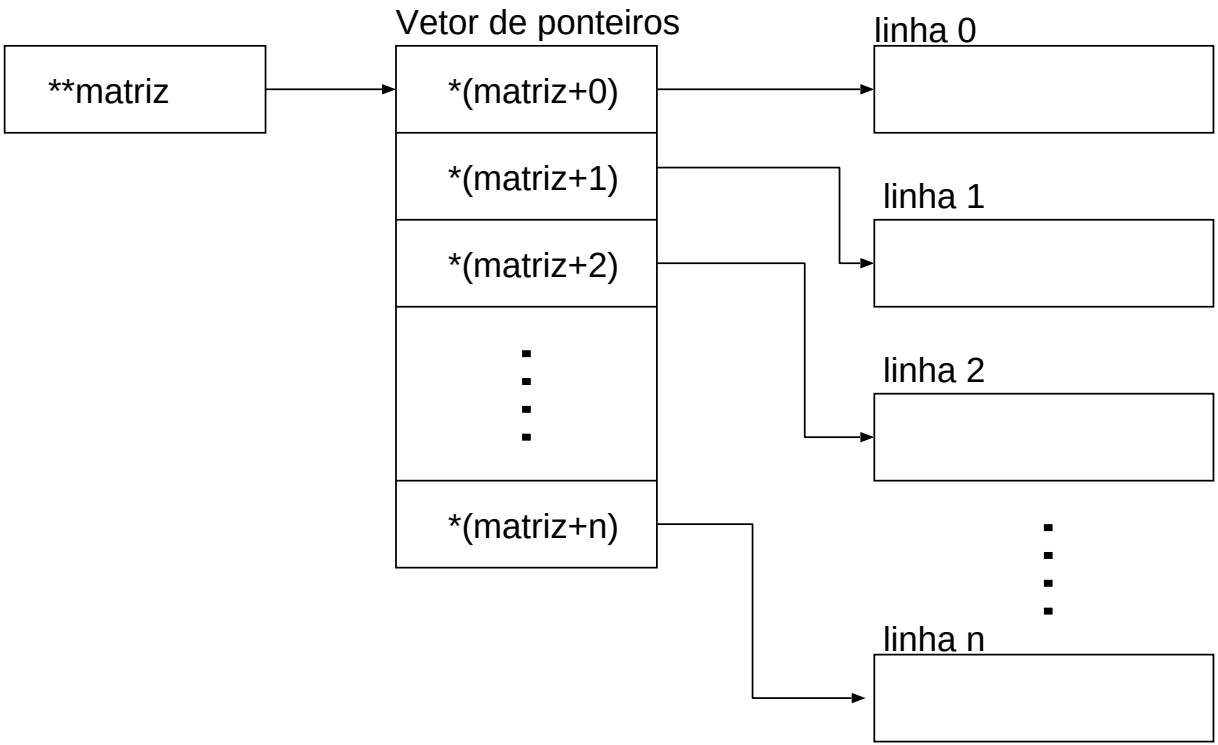


Vetores de Ponteiros: Passo 3

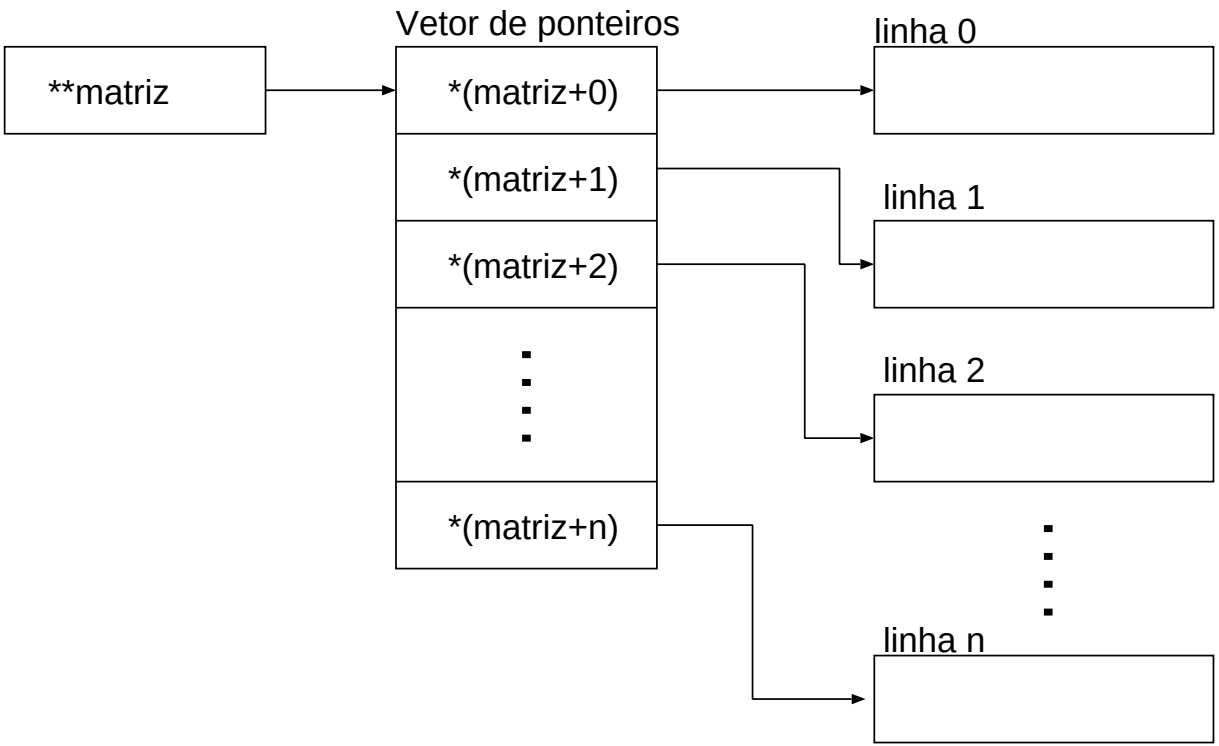
Precisamos saber quantas colunas cada linha vai ter. Cada linha pode ter um tamanho diferente.

```
int main (void) {
  int **matriz;
  int lin, col, i, j;
  /* .... */
  puts("Qual o numero de colunas?");
  scanf("%d", &col);
  for (i=0; i<lin; i++) {
    *(matriz +i) = (int *) malloc(col*sizeof(int));
      if (! *(matriz+i) ) {
        printf("Sem espaço para linha %d", i);
        return 1;
      }
  }
    /* ... */
}
```

Vetores de Ponteiros: Passo 3



Ponteiros para ponteiros



The End