

# MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2015

# Roteiro

- 1 Ponteiros
- 2 Passagem de parâmetros por valor e por referência
- 3 Aritmética de ponteiros
- 4 Ponteiros e vetores
- 5 Alocação dinâmica de memória
- 6 Alocação dinâmica de matrizes
- 7 O depurador valgrind
- 8 Exercícios

# Ponteiros

- Ponteiros (também chamados de apontadores) são tipos especiais de dados que armazenam endereços de memória.
- Uma variável do tipo ponteiro deve ser declarada da seguinte forma:

```
tipo *nome_variável;
```

- A variável ponteiro armazenará um endereço de memória de uma outra variável do tipo especificado. Exemplo:

```
int *mema;  
float *memb;
```

- Neste exemplo, temos que:
  - ▶ mema pode armazenar o endereço de uma variável do tipo int.
  - ▶ memb pode armazenar o endereço de uma variável do tipo float.

# Operadores de ponteiros

- Existem dois operadores relacionados a ponteiros:
  - ▶ O operador `&` retorna o endereço de memória de uma variável:

```
int *mema;  
int a = 90;  
mema = &a;
```

- ▶ O operador `*` retorna o conteúdo do endereço armazenado no ponteiro:

```
printf("%d", *mema);
```



## Exemplo com ponteiros

```
#include <stdio.h>

int main() {
    int b, *c;

    b = 10;
    c = &b;
    *c = 11;

    printf("%d\n", b);

    return 0;
}
```

## Exemplo com ponteiros

```
#include <stdio.h>

int main() {
    int b, *c;

    b = 10;
    c = &b;
    *c = 11;

    printf("%d\n", b); /* 11 */

    return 0;
}
```

## Exemplo com ponteiros

```
#include <stdio.h>

int main() {
    int num, q = 1, *p;

    num = 100;
    p = &num;
    q = *p;

    printf("%d\n", q);

    return 0;
}
```

## Exemplo com ponteiros

```
#include <stdio.h>

int main() {
    int num, q = 1, *p;

    num = 100;
    p = &num;
    q = *p;

    printf("%d\n", q); /* 100 */

    return 0;
}
```



## Exemplo com ponteiros

```
#include <stdio.h>

int main() {
    int a = 3, b = 2;
    int *p, *q;

    p = &a;
    q = p;
    *q = *q + 1;
    q = &b;
    b = b + 1;

    printf("%d, %d\n", *q, *p);

    return 0;
}
```

## Exemplo com ponteiros

```
#include <stdio.h>

int main() {
    int a = 3, b = 2;
    int *p, *q;

    p = &a;
    q = p;
    *q = *q + 1;
    q = &b;
    b = b + 1;

    printf("%d, %d\n", *q, *p); /* 3, 4 */

    return 0;
}
```

## Cuidados com uso de ponteiros

- Não se deve atribuir um valor ao endereço armazenado num ponteiro, sem antes ter certeza de que o endereço é válido:

```
int a, b, *c;  
  
b = 10;  
*c = 13; /* Erro: onde o valor 13 sera armazenado? */
```

- O correto seria algo como:

```
int a, b, *c;  
  
b = 10;  
c = &a;  
*c = 13;
```

- Como o operador \* de ponteiros é igual ao operador \* utilizado na multiplicação, deve-se ter cuidado no uso desses operadores.

## Cuidados com uso de ponteiros

```
#include <stdio.h>

int main() {
    int b, a, *c;

    b = 10;
    c = &a;
    *c = 11;
    a = b * c; /* Erro: operacao invalida */

    printf("%d\n", a);

    return 0;
}
```

# Cuidados com uso de ponteiros

```
#include <stdio.h>

int main() {
    int b, a, *c;

    b = 10;
    c = &a;
    *c = 11;
    a = b * (*c);

    printf("%d\n", a); /* 110 */

    return 0;
}
```

## Cuidados com uso de ponteiros

```
#include <stdio.h>

int main() {
    double a;
    int *b, c;

    a = 10.89;
    b = &a;
    c = *b;

    printf("%d\n", c); /* 0 que sera impresso? */

    return 0;
}
```

## Cuidados com uso de ponteiros

```
#include <stdio.h>

int main() {
    double a;
    int *b, c;

    a = 10.89;
    b = &a; /* Erro: tipos incompatíveis */
    c = *b;

    printf("%d\n", c); /* 343597384 */

    return 0;
}
```

# Comparação de valores

```
#include <stdio.h>

int main() {
    double *a, *b, c, d;
    a = &d;
    b = &c;
    scanf("%lf %lf", &c, &d);

    if (*a < *b)
        printf("Valor: %lf < %lf\n", *a, *b);
    else
        if (*b < *a)
            printf("Valor: %lf < %lf\n", *b, *a);
        else if (*a == *b)
            printf("Mesmo valor: %lf\n", *a);

    return 0;
}
```



## Comparação de endereços

```
#include <stdio.h>

int main() {
    double *a, *b, c, d;
    a = &d;
    b = &c;
    scanf("%lf %lf", &c, &d);

    if (a < b)
        printf("Endereco: %p < %p\n", a, b);
    else
        if (b < a)
            printf("Endereco: %p < %p\n", b, a);
        else if (a == b)
            printf("Mesmo endereco\n"); /* impossivel neste exemplo */

    return 0;
}
```

# Ponteiro nulo

- Quando um ponteiro não está associado a nenhum endereço válido é comum atribuir o valor NULL para este.
- NULL é definido como zero nas bibliotecas `stdio.h` e `stdlib.h`, entre outras.
- O valor NULL é usado em comparações com ponteiros para descobrir se um determinado ponteiro possui um endereço válido ou não.
- Note que um ponteiro que não recebeu um valor inicial não necessariamente possui valor NULL (zero).

# Ponteiro nulo

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

int main() {
    int numero, *ponteiro = NULL;

    scanf("%d", &numero);

    if (numero > 0)
        ponteiro = &numero;

    if (ponteiro != NULL)
        printf("Numero: %d\n", *ponteiro);
    else
        printf("Erro: ponteiro nulo.\n");

    return 0;
}
```

## Ponteiro nulo

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

int main() {
    int numero, *ponteiro = NULL;

    scanf("%d", &numero);

    if (numero > 0)
        ponteiro = &numero;

    if (ponteiro)
        printf("Numero: %d\n", *ponteiro);
    else
        printf("Erro!\n");

    return 0;
}
```

# Passagem de parâmetros

- Passagem de parâmetro é o mecanismo utilizado para fornecer informações para uma função.
- Há dois tipos de passagem de parâmetros:
  - ▶ Passagem por valor.
  - ▶ Passagem por referência.

# Passagem de parâmetros por valor

- Quando passamos parâmetros para uma função, os valores fornecidos são copiados para as variáveis parâmetros da função.
- Este processo é idêntico a uma atribuição e é chamado de passagem por valor.
- Desta forma, alterações nos parâmetros dentro da função não alteram os valores usados na chamada da função.

## Passagem de parâmetros por valor

```
#include <stdio.h>

void nao_troca(int a, int b) {
    int aux = a;
    a = b;
    b = aux;
}

int main() {
    int x = 4, y = 5;

    nao_troca(x, y);

    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

## Passagem de parâmetros por valor

```
#include <stdio.h>

void nao_troca(int a, int b) {
    int aux = a;
    a = b;
    b = aux;
}

int main() {
    int x = 4, y = 5;

    nao_troca(x, y);

    printf("x = %d, y = %d\n", x, y); /* x = 4, y = 5 */
    return 0;
}
```



# Passagem de parâmetros por referência

- Em C, só existe passagem de parâmetros por valor.
- Em algumas linguagens, há construções para se passar parâmetros por referência.
  - ▶ Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
  - ▶ No exemplo anterior, se  $x$  e  $y$  fossem passados por referência, seus conteúdos seriam trocados.

# Simulando passagem de parâmetros por referência

- Podemos simular o efeito de passagem de parâmetros por referência em C utilizando ponteiros.
- O artifício corresponde em fornecer como parâmetro para uma função o endereço de uma variável e não o seu valor.
- Desta forma, podemos alterar o conteúdo da variável, de forma equivalente a passagem de parâmetro por referência.

## Simulando passagem de parâmetros por referência

```
#include <stdio.h>

void troca(int *a, int *b) {
    int aux = *a;
    *a = *b;
    *b = aux;
}

int main() {
    int x = 4, y = 5;

    troca(&x, &y);

    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

## Simulando passagem de parâmetros por referência

```
#include <stdio.h>

void troca(int *a, int *b) {
    int aux = *a;
    *a = *b;
    *b = aux;
}

int main() {
    int x = 4, y = 5;

    troca(&x, &y);

    printf("x = %d, y = %d\n", x, y); /* x = 5, y = 4 */
    return 0;
}
```

## Simulando retorno de múltiplos valores

- A passagem de ponteiros como parâmetros é útil quando precisamos retornar mais do que um valor.
- Suponha que queremos criar uma função que receba um vetor como parâmetro e retorne o menor e o maior elemento do vetor.
  - ▶ Como sabemos, uma função pode retornar no máximo um valor.
  - ▶ Podemos passar ponteiros para variáveis que armazenarão o menor e o maior elemento, simulando assim o retorno de dois valores.

# Simulando retorno de múltiplos valores

```
#include <stdio.h>

void min_and_max(int vet[], int tam, int *min, int *max);

int main() {
    int v[10] = {10, 80, 5, -10, 45, -20, 100, 125, 200, 10};
    int min, max;
    min_and_max(v, 10, &min, &max);
    printf("Menor valor: %d\n", min);
    printf("Maior valor: %d\n", max);
    return 0;
}

void min_and_max(int vet[], int tam, int *min, int *max) {
    int i;
    *min = vet[0];
    *max = vet[0];

    for (i = 1; i < tam; i++)
        if (vet[i] < *min)
            *min = vet[i];
        else if (vet[i] > *max)
            *max = vet[i];
}
```

# Simulando retorno de múltiplos valores

```
#include <stdio.h>

void min_and_max(int vet[], int tam, int *min, int *max);

int main() {
    int v[10] = {10, 80, 5, -10, 45, -20, 100, 125, 200, 10};
    int min, max;
    min_and_max(v, 10, &min, &max);
    printf("Menor valor: %d\n", min); /* -20 */
    printf("Maior valor: %d\n", max); /* 200 */
    return 0;
}

void min_and_max(int vet[], int tam, int *min, int *max) {
    int i;
    *min = vet[0];
    *max = vet[0];

    for (i = 1; i < tam; i++)
        if (vet[i] < *min)
            *min = vet[i];
        else if (vet[i] > *max)
            *max = vet[i];
}
```

## Aritmética de ponteiros

- Os operadores  $+$  e  $-$ , assim como os operadores  $++$  e  $--$ , podem ser utilizados com ponteiros.
- Seja  $p$  um ponteiro para um inteiro (num computador de 32 bits) com um valor atual (endereço) igual a 3000.
- A expressão:

```
p++;
```

faz com que o conteúdo de  $p$  seja alterado para 3004 (e não 3001).

- A cada incremento de  $p$ , ele apontará para o próximo endereço de um tipo inteiro, cujo tamanho é de 4 bytes para o computador deste exemplo.
- O mesmo é válido para decrementos. Por exemplo:

```
p--;
```

faz com que  $p$  assuma o valor 2996 (considerando que o endereço anterior era 3000 e o tamanho de um inteiro é de 4 bytes).



## Ponteiros e vetores

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

```
int v[5]; /* aloca 5 * 4 = 20 bytes de memoria,  
          supondo que cada inteiro ocupe 4 bytes */
```

- Uma variável vetor, assim como um ponteiro, armazena um endereço de memória: o endereço do início do vetor.
- No exemplo acima, a variável `v` armazena o endereço de memória do início do vetor. Ou seja, `v` e `&v[0]` possuem o mesmo valor.
- Por este motivo, quando passamos um vetor como parâmetro para uma função, seu conteúdo pode ser alterado dentro da função, pois estamos passando, na realidade, o endereço do início do espaço alocado para o vetor.

## Ponteiros e vetores

```
#include <stdio.h>

void zeraVetor(int vet[], int tam) {
    int i;
    for (i = 0; i < tam; i++)
        vet[i] = 0;
}

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

    zeraVetor(vetor, 5);

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

    return 0;
}
```

## Ponteiros e vetores

```
#include <stdio.h>

void zeraVetor(int *vet, int tam) {
    int i;
    for (i = 0; i < tam; i++)
        vet[i] = 0;
}

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

    zeraVetor(&vetor[0], 5);

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

    return 0;
}
```

## Ponteiros e vetores

- De fato, como uma variável vetor possui um endereço, podemos atribuí-la a uma variável ponteiro:

```
int a[] = {1, 2, 3, 4, 5}, *p;  
p = a;
```

- E podemos então usar `p` como se fosse um vetor:

```
for (i = 0; i < 5; i++)  
    p[i] = i * i;
```

- Uma variável vetor, diferentemente de um ponteiro, possui um endereço fixo (o do começo do vetor).
- Ou seja, não podemos alterar o endereço de uma variável vetor.

## Ponteiros e vetores

```
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5}, b[5], i;

    b = a; /* Erro: atribuicao invalida */

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

    return 0;
}
```

## Ponteiros e vetores

```
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5}, *b, i;

    b = a; /* atribuicao valida */

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

    return 0;
}
```

## Ponteiros e vetores

- Aritmética de ponteiros pode ser utilizada com vetores. Exemplo:

```
char str[80], *p, c;  
p = str;
```

- O ponteiro `p` recebeu o endereço do primeiro elemento do vetor de caracteres (string) `str`.
- Para fazer acesso ao quinto elemento de `str`, podemos escrever:

```
c = p[4];
```

- O seguinte comando tem exatamente o mesmo efeito:

```
c = *(p + 4);
```

## Ponteiros e vetores

O resultado deste programa...

```
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5};
    int *b, i;

    b = a;

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

    return 0;
}
```



## Ponteiros e vetores

... é igual a este...

```
#include <stdio.h>
```

```
int main() {  
    int a[] = {1, 2, 3, 4, 5};  
    int *b, i;  
  
    b = a;  
  
    for (i = 0; i < 5; i++)  
        printf("%d ", *(a + i));  
    printf("\n");  
  
    return 0;  
}
```

# Ponteiros e vetores

... e a este...

```
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5};
    int *b, i;

    b = a;

    for (i = 0; i < 5; i++)
        printf("%d ", *(b + i));
    printf("\n");

    return 0;
}
```

## Ponteiros e vetores

... ou mesmo este.

```
#include <stdio.h>

int main() {
    int a[] = {1, 2, 3, 4, 5};
    int *b, i;

    b = a;

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

    return 0;
}
```

## Ponteiros e vetores

```
#include <stdio.h>

int main() {
    char string1[] = "teste";
    char string2[] = "teste";

    if (string1 == string2)
        printf("Iguais\n");
    else
        printf("Diferentes\n");

    return 0;
}
```

## Ponteiros e vetores

```
#include <stdio.h>

int main() {
    char string1[] = "teste";
    char string2[] = "teste";

    if (string1 == string2)
        printf("Iguais\n");
    else
        printf("Diferentes\n"); /* sempre diferentes,
                                   independente do conteudo */

    return 0;
}
```

## Ponteiros e vetores

```
#include <stdio.h>

int main() {
    char string1[] = "teste";
    char string2[] = "texto";

    if (*string1 == *string2)
        printf("Iguais\n");
    else
        printf("Diferentes\n");

    return 0;
}
```

## Ponteiros e vetores

```
#include <stdio.h>

int main() {
    char string1[] = "teste";
    char string2[] = "texto";

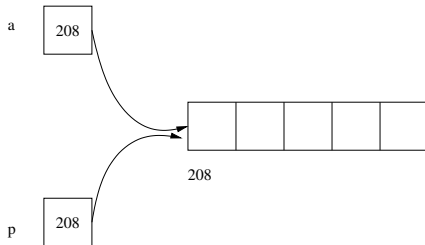
    if (*string1 == *string2)
        printf("Iguais\n"); /* string1[0] = string2[0] = 't' */
    else
        printf("Diferentes\n");

    return 0;
}
```

## Alocação dinâmica de memória

- Como vimos anteriormente, uma variável vetor possui um endereço, que podemos atribuí-lo para uma variável ponteiro  $p$  e, dessa forma, podemos usar  $p$  como se fosse um vetor:

```
int a[] = {1, 2, 3, 4, 5}, *p;  
p = a;  
for (i = 0; i < 5; i++)  
    p[i] = i * i;
```





## Alocação dinâmica de memória

- Em aulas anteriores, ao trabalharmos, por exemplo, com matrizes assumíamos que estas tinham dimensões máximas conhecidas:

```
#define MAX 100  
...  
int matriz[MAX][MAX];
```

- O que fazer se o usuário precisar trabalhar com matrizes maiores?
- Será que é possível alocar apenas a quantidade de memória necessária para o programa, evitando assim o desperdício de recursos computacionais?

## Alocando memória dinamicamente

- A biblioteca `stdlib.h` possui funções que permitem manipular memória dinamicamente.
- A função `malloc` aloca uma região contígua de memória, com número de bytes recebido como parâmetro, retornando um ponteiro para a primeira posição da região de memória alocada.
- Exemplo: alocação de memória para 100 números inteiros.

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

- Exemplo: alocação de memória para 80 caracteres.

```
char *p;  
p = malloc(80 * sizeof(char));
```

- Caso ocorra um erro na alocação de memória, a função `malloc` retornará um ponteiro nulo (`NULL`).

## Liberando memória alocada dinamicamente

- A função `free` recebe como parâmetro um ponteiro e libera a memória previamente alocada e apontada pelo ponteiro.
  - ▶ Exemplo: liberando memória previamente alocada.

```
free(p);
```

- Importante: toda memória alocada dinamicamente durante a execução de um programa (com `malloc`) deve ser desalocada (com o `free`) quando não for mais necessária.

## Exemplo - produto interno de vetores

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

int main() {
    double *v1, *v2, produto;
    int i, n;

    printf("Qual o tamanho dos vetores?\n");
    scanf("%d", &n);

    v1 = malloc(n * sizeof(double));
    v2 = malloc(n * sizeof(double));

    printf("Entre com os valores do primeiro vetor: ");
    for (i = 0; i < n; i++)
        scanf("%lf", &v1[i]);

    ...
}
```

## Exemplo - produto interno de vetores

...

```
printf("Entre com os valores do segundo vetor: ");
for (i = 0; i < n; i++)
    scanf("%lf", &v2[i]);

produto = 0;
for (i = 0; i < n; i++)
    produto = produto + (v1[i] * v2[i]);

printf("Produto interno dos dois vetores: %f\n", produto);

free(v1);
free(v2);

return 0;
}
```

## Ponteiros e alocação dinâmica

- Podemos fazer ponteiros distintos apontarem para uma mesma região de memória.
- Neste caso, precisamos tomar cuidado para não acessar uma região de memória (através de um ponteiro) que foi previamente desalocada.
- Exemplo:

```
double *v1, *v2;

v1 = malloc(100 * sizeof(double));
v2 = v1;
free(v1);

for (i = 0; i < n; i++)
    v2[i] = i;
```

- O código acima está errado e irá causar um erro de execução já que v2 está acessando posições de memória que não estão mais reservadas para o programa.

# Alocação dinâmica de matrizes

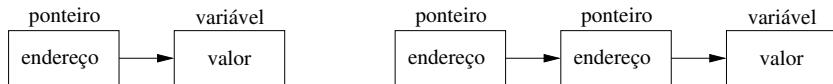
- Em aplicações científicas e de engenharias, é muito comum a realização de diversas operações sobre matrizes.
- Como vimos, em situações reais o ideal é alocar memória suficiente para conter os dados a serem tratados, sem usar nem mais e nem menos memória do que o necessário.
- Como alocar vetores multidimensionais dinamicamente?

# Ponteiros de ponteiros

- Uma variável ponteiro é alocada na memória do computador como qualquer outra variável.
- Portanto, podemos criar um ponteiro que contém o endereço de memória de um outro ponteiro.
- O ponteiro de um ponteiro é uma forma de endereçamento encadeado.



# Ponteiros de ponteiros



- Na figura à esquerda, o valor do ponteiro é o endereço da variável que contém o valor desejado.
- Na figura à direita, o primeiro ponteiro contém o endereço de um segundo ponteiro, que aponta para a variável que tem o valor desejado.

# Ponteiros de ponteiros

- O que o programa abaixo irá imprimir quando executado?

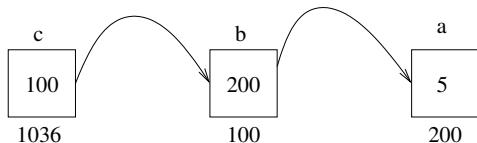
```
#include <stdio.h>

int main() {
    int a, *b, **c;

    a = 5;
    b = &a;
    c = &b;
    printf("%d\n", *(*c));

    return 0;
}
```

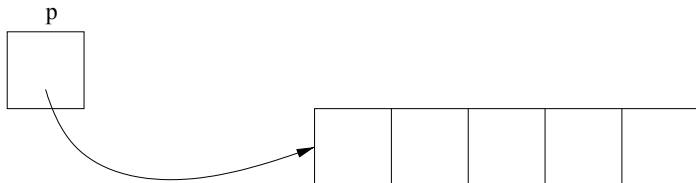
- O programa imprimirá o valor de a, ou seja, 5.



# Ponteiros de ponteiros

- Sabemos que um ponteiro pode ser usado para referenciar um vetor alocado dinamicamente.

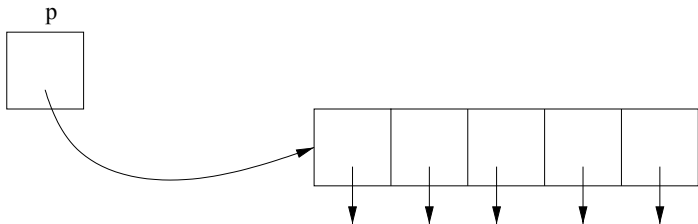
```
int *p;  
p = malloc(5 * sizeof(int));
```



## Ponteiros de ponteiros

- Da mesma forma, podemos usar um ponteiro de ponteiro para referenciar um vetor de ponteiros alocado dinamicamente.

```
int **p;  
p = malloc(5 * sizeof(int *));
```

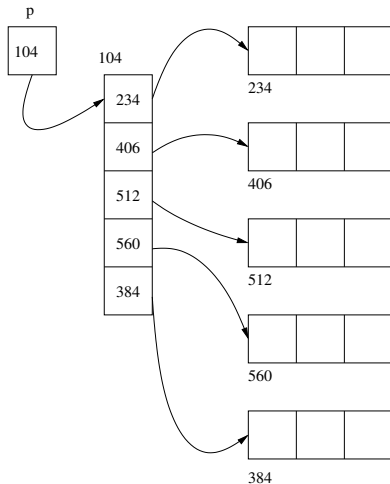


- Note que cada posição do vetor acima é do tipo `int *`, ou seja, um ponteiro para inteiro.

## Ponteiros de ponteiros

- Como cada posição do vetor é um ponteiro para inteiro, podemos associar cada posição dinamicamente com um vetor de inteiros.

```
int **p, i;  
p = malloc(5 * sizeof(int *));  
  
for (i = 0; i < 5; i++)  
    p[i] = malloc(3 * sizeof(int));
```



# Alocação dinâmica de matrizes

- Podemos alocar matrizes dinamicamente da seguinte forma:
  - ▶ Crie um ponteiro para ponteiro.
  - ▶ Associe um vetor de ponteiros dinamicamente com este ponteiro de ponteiro. O tamanho deste vetor será o número de linhas da matriz.
  - ▶ Associe cada posição do vetor com um outro vetor do tipo a ser armazenado. Cada um destes vetores será uma linha da matriz (portanto, possuirá tamanho igual ao número de colunas).
- Lembre que devemos desalocar toda a memória alocada por este processo assim que ela não for mais necessária.

## Exemplo - alocação dinâmica de matrizes

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

int main() {
    int **matriz, linhas, colunas, i, j;

    printf("Entre com o numero de linhas: ");
    scanf("%d", &linhas);

    printf("Entre com o numero de colunas: ");
    scanf("%d", &colunas);

    printf("Alocando a matriz...\n");
    matriz = malloc(linhas * sizeof(int *));
    for (i = 0; i < linhas; i++)
        matriz[i] = malloc(colunas * sizeof(int));
    ...
}
```

## Exemplo - alocação dinâmica de matrizes

```
...
printf("Obtendo os valores da matriz...\n");
for (i = 0; i < linhas; i++)
    for (j = 0; j < colunas; j++)
        scanf("%d", &matriz[i][j]);

printf("Imprimindo a matriz...\n");
for (i = 0; i < linhas; i++) {
    for (j = 0; j < colunas; j++)
        printf("%d ", matriz[i][j]);
    printf("\n");
}

printf("Desalocando a matriz...\n");
for (i = 0; i < linhas; i++)
    free(matriz[i]);
free(matriz);

return 0;
}
```



# Exemplo - alocação dinâmica de matrizes usando funções

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

int ** aloca_matriz(int linhas, int colunas) {
    int i, **matriz;

    matriz = malloc(linhas * sizeof(int *));

    for (i = 0; i < linhas; i++)
        matriz[i] = malloc(colunas * sizeof(int));

    return matriz;
}

void desaloca_matriz(int **matriz, int linhas) {
    int i;

    for (i = 0; i < linhas; i++)
        free(matriz[i]);

    free(matriz);
}
```

# Exemplo - alocação dinâmica de matrizes usando funções

```
void obtem_matriz(int **matriz, int linhas, int colunas) {
    int i, j;

    for (i = 0; i < linhas; i++)
        for (j = 0; j < colunas; j++)
            scanf("%d", &matriz[i][j]);
}
```

```
void imprime_matriz(int **matriz, int linhas, int colunas) {
    int i, j;

    for (i = 0; i < linhas; i++) {
        for (j = 0; j < colunas; j++)
            printf("%d ", matriz[i][j]);
        printf("\n");
    }
}
```

# Exemplo - alocação dinâmica de matrizes usando funções

```
int main() {
    int **matriz, linhas, colunas;

    printf("Entre com o numero de linhas: ");
    scanf("%d", &linhas);

    printf("Entre com o numero de colunas: ");
    scanf("%d", &colunas);

    printf("Alocando a matriz...\n");
    matriz = aloca_matriz(linhas, colunas);

    printf("Obtendo os valores da matriz...\n");
    obtem_matriz(matriz, linhas, colunas);

    printf("Imprimindo a matriz...\n");
    imprime_matriz(matriz, linhas, colunas);

    printf("Desalocando a matriz...\n");
    desaloca_matriz(matriz, linhas);

    return 0;
}
```

## O depurador valgrind

- Criado por Julian Seward, `valgrind` é um software livre que auxilia na depuração de programas.
- Ele ajuda a detecção de erros decorrentes do uso incorreto da memória dinâmica, como, por exemplo, os vazamentos de memória, alocação e desalocação incorretas e acessos a áreas inválidas.
- Ele usa uma máquina virtual para simular o acesso à memória do programa em teste, eliminando a necessidade de uso de bibliotecas auxiliares ou de mudanças no código.
- Apesar de ter sido desenvolvido para programas codificados em C ou C++, a máquina virtual torna possível o uso do `valgrind` com programas que foram codificados em outras linguagens, como Java.
- Na mitologia nórdica, Valgrind é o nome do portão do palácio de Valhalla. No palácio de Valhalla, que fica na terra mitológica de Asgard, entram pelo Valgrind apenas os cavaleiros mais bravos escolhidos dos deuses para enfrentar o Ragnarok, o fim do mundo.

## A ferramenta memcheck

- A ferramenta memcheck é a ferramenta padrão utilizada pelo valgrind, caso uma outra não seja explicitamente indicada.
- Com esta ferramenta é possível monitorar como o programa faz uso da memória, pois ela mantém um mapa de bits indicando quais áreas da memória estão alocadas, quais não estão alocadas e quais estão alocadas, mas não inicializadas.
- Para maiores informações sobre o valgrind, consulte seu site oficial: <http://valgrind.org>

## Exemplo de programa com erros (teste.c)

```
#include <stdlib.h>

void atualiza(int vetor[], int n) {
    int i;
    for (i = 0; i < n; i++)
        vetor[i-1]++;
}

int main() {
    int *vetor, i, n = 10;

    vetor = malloc(n * sizeof(int));

    for (i = 0; i < n; i++)
        vetor[i+1] = i;

    atualiza(vetor, n);

    return 0;
}
```

# Exemplo de uso do valgrind

```
$ gcc teste.c -o teste
$ valgrind --leak-check=full ./teste
==40475== Memcheck, a memory error detector
==40475== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==40475== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==40475== Command: ./teste
==40475==
==40475== Invalid write of size 4
==40475==    at 0x100000F50: main (in ./teste)
==40475==   Address 0x100011fb8 is 0 bytes after a block of size 40 alloc'd
==40475==    at 0x47E1: malloc (vg_replace_malloc.c:300)
==40475==   by 0x100000F25: main (in ./teste)
==40475==
==40475== Invalid read of size 4
==40475==    at 0x100000EDD: atualiza (in ./teste)
==40475==   by 0x100000F6E: main (in ./teste)
==40475==   Address 0x100011f8c is 4 bytes before a block of size 40 alloc'd
==40475==    at 0x47E1: malloc (vg_replace_malloc.c:300)
==40475==   by 0x100000F25: main (in ./teste)
```

# Exemplo de uso do valgrind

```
==40475== Invalid write of size 4
==40475==    at 0x100000EE5: atualiza (in ./teste)
==40475==    by 0x100000F6E: main (in ./teste)
==40475== Address 0x100011f8c is 4 bytes before a block of size 40 alloc'd
==40475==    at 0x47E1: malloc (vg_replace_malloc.c:300)
==40475==    by 0x100000F25: main (in ./teste)
==40475==
==40475==
==40475== HEAP SUMMARY:
==40475==    in use at exit: 24,935 bytes in 371 blocks
==40475==    total heap usage: 447 allocs, 76 frees, 30,879 bytes allocated
==40475==
==40475== 40 bytes in 1 blocks are definitely lost in loss record 30 of 76
==40475==    at 0x47E1: malloc (vg_replace_malloc.c:300)
==40475==    by 0x100000F25: main (in ./teste)
```



# Exemplo de uso do valgrind

```
==40475== LEAK SUMMARY:  
==40475==      definitely lost: 40 bytes in 1 blocks  
==40475==      indirectly lost: 0 bytes in 0 blocks  
==40475==      possibly lost: 0 bytes in 0 blocks  
==40475==      still reachable: 0 bytes in 0 blocks  
==40475==           suppressed: 24,895 bytes in 370 blocks  
==40475==  
==40475== For counts of detected and suppressed errors, rerun with: -v  
==40475== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 15 from 15)
```

## Exercícios

- Escreva uma função `length(s)` que receba como parâmetro uma string `s` e retorne seu tamanho (equivalente a função `strlen(s)` da biblioteca `string.h`).
- Escreva uma função `copy(s,t)` que receba como parâmetro duas strings e copie o conteúdo da string `t` na string `s` (equivalente a função `strcpy(s,t)` da biblioteca `string.h`).
- Escreva uma função `compare(s,t)` que receba como parâmetro duas strings e compare `s` e `t`, retornando um valor negativo, zero ou positivo se `s` for, respectivamente, lexicograficamente menor, igual ou maior que `t` (equivalente a função `strcmp(s,t)` da biblioteca `string.h`).
- Escreva uma função `concatenate(s,t)` que receba como parâmetro duas strings e concatene `t` em `s` (equivalente a função `strcat(s,t)` da biblioteca `string.h`).

# Exercícios

- Escreva funções que, dados dois vetores  $A$  e  $B$ , representando conjuntos com  $n$  e  $m$  números inteiros respectivamente, calcule:
  - ▶  $C = A \cup B$
  - ▶  $C = A \cap B$
  - ▶  $C = A - B$
  - ▶  $C = A \triangle B = (A - B) \cup (B - A)$
- Escreva um programa que, dadas duas matrizes  $A$  e  $B$  de números inteiros, de dimensões  $p \times q$  e  $q \times r$  respectivamente, calcule a matriz produto  $C = A \times B$ , de dimensões  $p \times r$ . Seu programa deve alocar as 3 matrizes dinamicamente.

# Length

```
/* Versao com vetores */  
int length(char s[]) {  
    int i = 0;  
  
    while (s[i])  
        i++;  
  
    return i;  
}
```

# Length

```
/* Versao com ponteiros */  
int length(char *s) {  
    int i = 0;  
  
    while (*(s + i))  
        i++;  
  
    return i;  
}
```

# Copy

```
/* Versao com vetores */  
void copy(char s[], char t[]) {  
    int i = 0;  
  
    do {  
        s[i] = t[i];  
    } while (t[i++]);  
}
```

# Copy

```
/* Versao com ponteiros */  
void copy(char *s, char *t) {  
    int i = 0;  
  
    do {  
        *(s + i) = *(t + i);  
    } while (*(t + (i++)));  
}
```

# Copy

```
/* Versao com ponteiros */  
void copy(char *s, char *t) {  
    do {  
        *(s++) = *t;  
    } while (*(t++));  
}
```



# Compare

```
/* Versao com vetores */
int compare(char s[], char t[]) {
    int i = 0;

    while (s[i] == t[i])
        if (s[i++] == '\0')
            return 0;

    return (s[i] - t[i]);
}
```

# Compare

```
/* Versao com ponteiros */
int compare(char *s, char *t) {
    while (*s == *t) {
        if (*s == '\0')
            return 0;
        s++;
        t++;
    }

    return (*s - *t);
}
```

# Concatenate

```
/* Versao com vetores */  
void concatenate(char s[], char t[]) {  
    int i = 0, j = 0;  
  
    while (s[i])  
        i++;  
  
    while (t[j])  
        s[i++] = t[j++];  
  
    s[i] = '\0';  
}
```

# Concatenate

```
/* Versao com ponteiros */  
void concatenate(char *s, char *t) {  
    while (*s)  
        s++;  
  
    while (*t)  
        *(s++) = *(t++);  
  
    *s = '\0';  
}
```