

# Programação de Computadores I – Ponteiros

Profa. Mercedes Gonzales  
Márquez


---

# Variáveis (relembrando)

- Variável é um nome amigável (simbólico) que damos para uma localização de memória que receberá um dado de algum tipo.
- No momento da declaração de uma variável, um espaço de memória com um endereço já é alocado para ela.

```
int p,r,s  
float q  
p =50  
q =40.5
```

Nome	Endereço na memória	Conteúdo
p	1001	50
r	1005	
s	1009	
	⋮	⋮
q	2047	40.5



# Variáveis (relembrando)

- Resumindo:

A uma variável `x` como a declarada abaixo:

```
int x = 20;
```

tem a ela associados:

- um nome (`x`);
- um endereço de memória ou referência (`0xbf267c4`);
- um valor : 20.

# Operador &

- Retorna o endereço de uma variável:

```
int main() {  
    int x = 20;  
    printf("valor de x = %d\n", x);  
    printf("endereço de x = 0x%x\n", &x);  
}
```

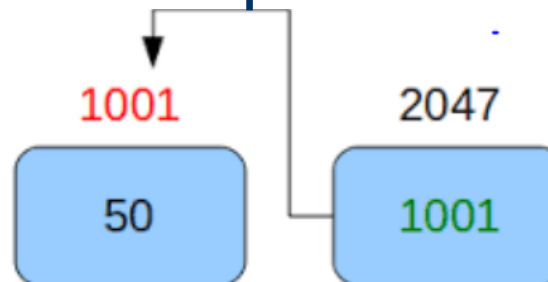
# Ponteiros - Conceito

- Um ponteiro é uma variável que armazena um endereço de memória.

```
int r,*s, p
float q
p = 50
q =40.5
s =&p
```

Nome	Endereço na memória	Conteúdo
p	1001	50
r	1005	
s	2047	1001
	⋮	⋮
q	3092	40.5

- O nome ponteiro refere ao fato deste tipo especial de variável “apontar” para um endereço.



# Ponteiros - Conceito

- Um ponteiro é uma variável que armazena um endereço de memória.
- Falaremos mais adiante também de ponteiros constantes, os quais não tem a possibilidade de mudar o seu conteúdo.

```
int main() {  
    int p, *q;  
    p=100;  
    q = &p;  
    printf("valor de p = %d\n", p);  
    printf("endereço de p = 0x%x\n", &p);  
    printf("endereço de q = 0x%x\n", q);  
    *q=200;  
    printf("valor de p = %d\n", p);  
}
```

# Ponteiros - Declaração

- Ponteiro tem um tipo associado que determina o tipo de dado contido no endereço apontado. Pode ser um tipo pré-definido da linguagem ou um tipo definido pelo programador. Exemplo: inteiro, literal, real, TipoProduto, etc.
- Utilizamos o operador unário \*

```
int *p_int;
```

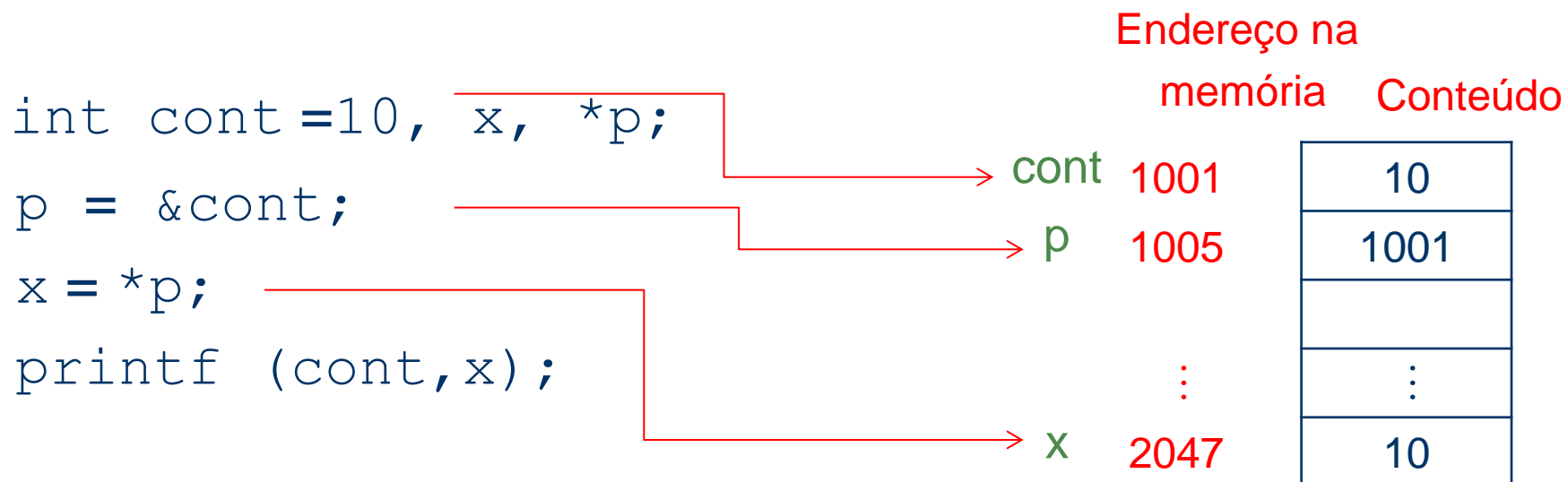
```
char *p_char;
```

```
float *p_float;
```

```
double *p_double;
```

# Fazendo acesso aos valores das variáveis referenciadas

- O símbolo \* além de ser usado na declaração de ponteiros, quando aplicado a um ponteiro indica o valor armazenado no endereço contido no ponteiro.





# Ponteiros – Fazendo acesso aos valores das variáveis referenciadas

```
int main() {  
    int x;  
    int *p_x;  
    x = 10;  
    p_x = &x  
    printf("valor de x = %d\n", *p_x);  
    *p_x = 20;  
    printf("valor de x = %d\n", *p_x);  
    printf("valor de x = %d\n", x);  
}
```

# Ponteiros – Fazendo acesso aos valores das variáveis referenciadas

```
int main() {  
    int x,y,*p,*q;  
    x=15;  
    printf("valor de x = %d\n", x);  
    p=&x;  
    q=&x;  
    *q=2967;  
    printf("valor de x = %d\n", x);  
    *p=642362;  
    printf("valor de x = %d\n", x);  
}
```

# Ponteiro Nulo

- Quando um ponteiro não está associado a nenhum endereço válido podemos atribuir a ele o valor NULL .
- Um ponteiro nulo não aponta a posição alguma.
- Tome cuidado que um ponteiro que não recebeu um valor inicial não necessariamente possui valor NULL (zero).]
- É uma boa prática testar se o ponteiro é nulo antes de indagar sobre o valor apontado.

```
if (ip !=NULL)
```

```
...
```

# Aritmética de Ponteiros

- Os operadores `+` e `-` e os operadores `++` e `--`, podem ser utilizados com ponteiros.
- Considere um computador de 32 bits no qual um inteiro ocupa 4 bytes e considere `p` um ponteiro para um inteiro que aponta para um endereço 1000.
- A expressão: `p++`; faz com que o conteúdo de `p` seja alterado para 1004 (e não 1001).
- A cada incremento de `p`, ele apontará para o próximo endereço de um tipo inteiro.
- O mesmo é válido para decrementos.

# Aritmética de Ponteiros

- Em geral Se  $p$  é um ponteiro a tipo  $T$ ,  $p++$  incrementa o valor de  $p$  em  $\text{sizeof}(T)$  (onde  $\text{sizeof}(T)$  é a quantidade de armazenagem necessária para um dado de tipo  $T$ ).  
Similarmente,  $p--$  decrementa  $p$  em  $\text{sizeof}(T)$ ;

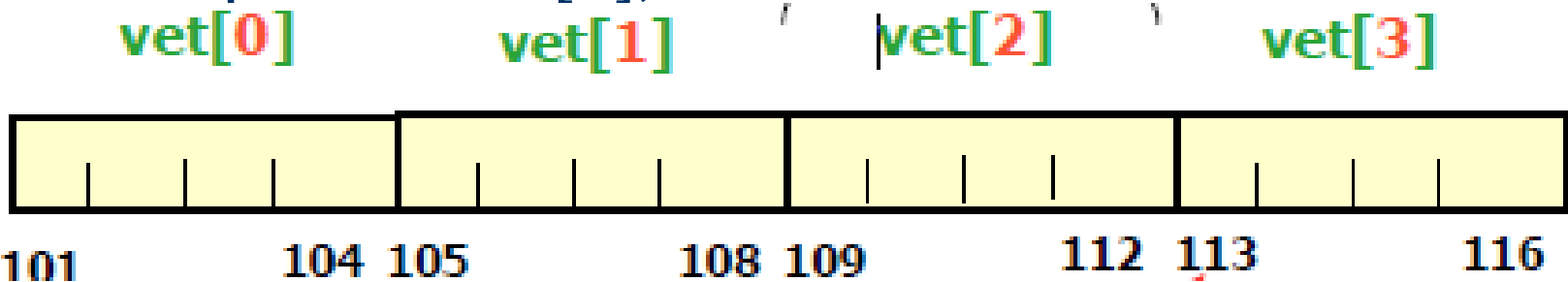
# Aritmética de Ponteiros

```
#include <stdio.h>
int main() {
    int x, *px=&x;
    char c, *pc=&c;
    double y, *py=&y;
    printf("valor de px = %d\n", px);
    px++;
    printf("valor de px = %d\n", px);
    printf("valor de pc = %d\n", pc);
    pc++;
    printf("valor de pc = %d\n", pc);
    printf("valor de py = %d\n", py);
    py++;
    printf("valor de py = %d\n", py);
}
```

# Ponteiros e Vetores

- Quando declaramos uma variável do tipo vetor, aloca-se uma quantidade de memória contígua cujo tamanho é especificado na declaração

Exemplo: `int vet[4];`



- Uma variável vetor (igual que um ponteiro) armazena um endereço de memória (o endereço do início do vetor). No exemplo, `vet` armazena o endereço de `vet[0]`, ou seja, `vet` e `&vet[0]` possuem o mesmo valor.

# Ponteiros e Vetores

- Por tal motivo, quando passamos um vetor como parâmetro para uma função, seu conteúdo pode ser alterado dentro da função, já que estamos passando, na verdade, o endereço do início do espaço alocado para o vetor.
- Tome cuidado porque um vetor, diferentemente de um ponteiro, possui um endereço fixo (o do começo do vetor), portanto, não podemos alterar o seu endereço. Este na verdade é conhecido na literatura como um ponteiro constante, não variável.



# Ponteiros e Vetores

- Aritmética de ponteiros pode ser utilizada com vetores. Exemplo:
- `int vet[4], *p, c;`  
`p = vet;`
- O ponteiro `p` recebeu o endereço do primeiro elemento do vetor de inteiros `vet`.
- Para fazer acesso ao quarto elemento de `vet`, podemos escrever: `c = p[3];` ou também `c=*(p+3);`

# Ponteiros e Vetores

```
#include <stdio.h>
void imprime_vetor(int v[], int n){
    int i;
    for (i = 0; i < n; i++){
        printf("%d ", v[i]);
        printf("\n");
    }
void imprime_vetor2(int *p, int n){
    int i;
    for (i = 0; i < n; i++) {
        printf("%d ", *p);
        p++;
    }
    printf("\n");
}
```

```
int main() {
    int i;
    int v[] = {10, 20, 30, 40, 50};
    for (i
    imprime_vetor(v, 5);
    imprime_vetor2(v, 5);
    imprime_vetor(&v[1], 4);
    imprime_vetor2(&v[1], 4);
}
```

# Ponteiros e Vetores

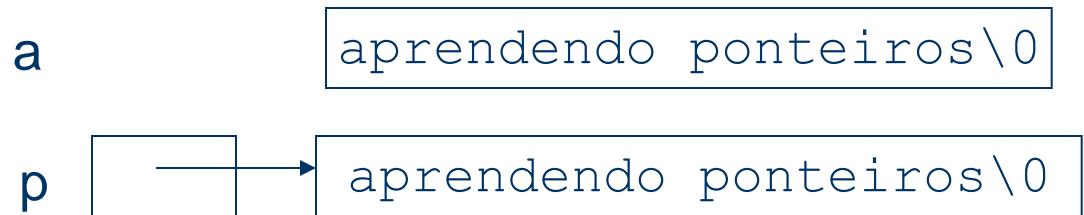
```
#include <stdio.h>
#define MAX 5
int main() {
    int v[] = {10, 20, 30, 40, 50}, i,*p,s=0;
    p=v;
    for (i=0;i<MAX; ++i)
        s+=p[i];
    printf ("Soma1=%d\n",s);
    for (i=0; i<MAX; ++i)
        s+=*(v+i);
    printf ("Soma2=%d\n",s);
    for (p=v;p<&v[MAX]; ++p)
        s+=*p;
    printf ("Soma3=%d\n",s);
}
```

# Mais aritmética de Ponteiros

- **Comparação de dois ponteiros.**
  - ***Se  $p$  e  $q$  apontam a membros do mesmo array***, então são válidas as relações como  $==$ ,  $!=$ ,  $<$ ,  $>=$ , etc. Por exemplo,  $p < q$  é verdade se  $p$  aponta a um elemento anterior do array que  $q$  aponta.
- **Subtração de ponteiros:**
  - ***Se  $p$  e  $q$  apontam a elementos do mesmo array***, e  $p < q$ , então  $q - p + 1$  é o número de elementos de  $p$  até  $q$  inclusive.
- O comportamento é indefinido para aritmética ou comparação com ponteiros que não apontam a membros do mesmo array.

# Ponteiros e strings

```
char a[] = "aprendendo ponteiros"; /* um array */  
char *p = "aprendendo ponteiros"; /* um ponteiro */
```



- a é um array, apenas do tamanho suficiente para armazenar a sequência de caracteres e '\0'.
- p é um ponteiro inicializado para apontar uma string constante;

# Exemplo de subtração de ponteiros a string

```
/* strlen: retorna o comprimento da string s */
int strlen(char *st){
    char *p=st;
    while (*p != '\0')
        p++;
    return p - st;
}
int main(){
    char s[30];
    int n, comp;
    scanf ("%s", s);
    comp=strlen(s);
    printf ("O comp. da string %s eh %d", s,comp);
}
```

p->

st[0]	st[1]	st[2]	st[3]	st[4]	st[5]	st[6]	st[7]	st[8]	st[9]	st[10]
A	l	g	o	r	i	t	m	o	s	\0
109	110	111	112	113	114	115	116	117	118	119

# Ponteiros - Exercícios

```
int x=3, y=5, *p=&x, *q=&y, *r;  
double z;
```

Expressão	Valor
<code>p==&amp;x</code>	1
<code>**&amp;p</code>	3
<code>r=&amp;z</code>	erro
<code>7**p/*q+7</code>	11
<code>*(&amp;y)*=*p</code>	15

# Ponteiros (Passagem por referência)

```
#include <stdio.h>
void nao_troca(int x, int y) {
    int aux;
    aux = x;
    x = y;
    y = aux; }
void troca(int *p_x, int *p_y) { }
    int aux;
    aux = *p_x;
    *p_x = *p_y;
    *p_y = aux;
}
```

```
int main() {
    int x = 20, y = 30;
    nao_troca(x, y); /*por valor*/
    printf("x = %d, y = %d\n", x, y);
    troca(&x, &y); /*por referencia*/
    printf("x = %d, y = %d\n", x, y);
}
```



# Ponteiros (Passagem por referência)

- O que será impresso pelo programa abaixo considerando  $x=2$  e  $y=5$ ?

```
#include <stdio.h>
void p(int a, int *b);
int main(){
    int x,y;
    scanf ("%d %d",&x,&y);
    p(x,&y);
    p(x,&y);
    printf ("%d e %d\n",x,y);
}
```

```
void p(int a, int *b){
    int aux;
    aux=*b;
    a=*b+4;
    *b=aux-3+a;
}
```

# Ponteiros (Passagem por referência)

- O que será impresso pelo programa abaixo considerando  $x=2$  e  $y=5$ ?

```
#include <stdio.h>
void p(int *a, int *b);
int main(){
    int x,y;
    scanf ("%d %d",&x,&y);
    p(&x,&y);
    p(&x,&y);
    printf ("%d e %d\n",x,y);
}
```

```
void p(int *a, int *b){
    int aux;
    aux=*b;
    *a=*b+4;
    *b=aux-3+*a;
}
```

# Ponteiros - Exercícios

- O que faz a seguinte função?

```
void imprime (char *v, int n) {  
    char *c;  
    for (c = v; c < v + n; c++)  
        printf ("%c", *c);  
}
```

- Analise o seguinte programa (Os valores de x e v[0] não deveriam ser iguais?)

```
void func1 (int x) {  
    x = 9 * x;  
}  
void func2 (int v[]) {  
    v[0] = 9 * v[0];  
}
```

```
int main (void) {  
    int x, v[2];  
    x = 111;  
    func1 (x);  
    printf ("x: %d\n", x);  
    v[0] = 111;  
    func2 (v);  
    printf ("v[0]: %d\n", v[0]);  
}
```

# Ponteiros para ponteiros

- Um ponteiro de ponteiro nada mais é que uma variável que contém o endereço de outra que, por sua vez, contém um endereço de memória onde há um dado.
- Declararamos um ponteiro para um ponteiro com da seguinte forma:

```
tipo_da_variável **nome_da_variável;
```

- Algumas considerações:
  - **\*\*nome\_da\_variável** é o conteúdo final da variável apontada;
  - **\*nome\_da\_variável** é o conteúdo do ponteiro intermediário.

# Ponteiros para ponteiros

```
int p,s,*q, **pp
```

```
float r
```

```
p =50
```

```
r =40.5
```

```
q =&p
```

```
pp=&q
```

```
q      1001
```

```
*q     50
```

```
pp     2047
```

```
*pp    1001
```

```
**pp   50
```

Nome	Endereço na memória	Conteúdo
p	1001	50
r	1005	40.5
s	1009	
	⋮	⋮
q	2047	1001
pp	3903	2047

# Ponteiros para ponteiros

- Para acessar o valor desejado apontado por um ponteiro para ponteiro, o operador asterisco deve ser aplicado duas vezes, como mostrado no exemplo abaixo:

```
float fpi =3.1415;
float *pf, **ppf;
pf = &fpi; /* pf armazena o endereco de fpi */
ppf = &pf; /* ppf armazena o endereco de pf */

printf (**ppf); /* Imprime o valor de fpi */

printf (*pf); /* Tambem imprime o valor de fpi */
```

# Alocação estática

- Até agora, todas as estruturas tinham tamanho pré-definido, exemplo matrizes e vetores.
- Esta alocação estática pode implicar em:
  - desperdício dos recursos pois podemos definir um tamanho muito maior do que aquele que normalmente será usado em tempo de execução.
  - Pode-se, em contraste, subestimar o tamanho necessário para armazenar os dados.

# Alocação dinâmica

- Na alocação dinâmica os espaços são alocados durante a execução do programa, conforme este necessitar de memória.
- Não há reserva antecipada de espaço.
- Alocar memória dinamicamente significa gerenciar memória (isto é, reservar, utilizar e liberar espaço) durante o tempo de execução.



# Alocação dinâmica

- função **malloc**:

- Reserva um bloco de memória.

`void malloc(num_bytes);`

- A função recebe como parâmetro o tamanho em bytes do bloco de memória a ser alocada
- E retorna um ponteiro para o início do bloco de memória reservado.
- Caso seja impossível, a função retorna **NULL**

# Alocação dinâmica

Programa

```
int *p;
```

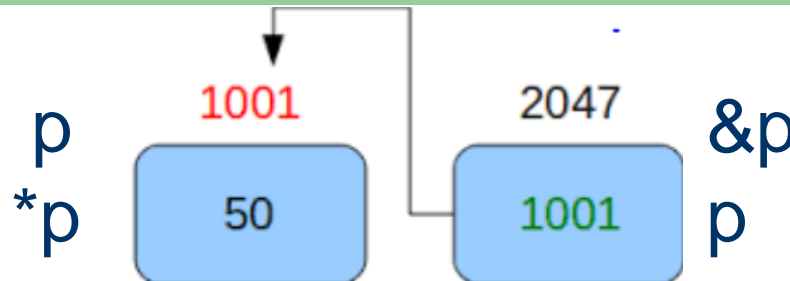
```
p = malloc(4);
```

```
*p = 5;
```

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

# Alocação Dinâmica

```
#include <stdio.h>
int main(){
    int *p;
    printf ("Endereco de p: %d\n", &p);
    printf ("Valor de p: %d\n", p);
    p = malloc(4);
    printf ("Valor de p: %d\n", p);
    printf ("Valor do endereço apontado por p: %d\n", *p);
    *p = 50;
    printf ("Valor do endereço apontado por p: %d\n", *p);
}
```



# Alocação dinâmica

- função **free**:
  - Parâmetro: endereço já alocado
  - Resultado: libera a área alocada

## Programa

```
int *p;
```

```
p = malloc(4);
```

```
*p = 5;
```

```
free(p);
```

# Alocação dinâmica

função **sizeof**:

- `sizeof(variavel)`
  - Retorna quantidade de memória ocupada pela variável
- `sizeof(tipo)`
  - Retorna quantidade de memória ocupada por uma variável com este tipo.

# Alocação dinâmica de um vetor

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int n, i, *vet;
    printf("Informe numero de elementos: \n");
    scanf("%d", &n);
    if ((vet=malloc(n * sizeof(int)))==NULL) {
        printf("Erro de alocação de memoria !\n");
        exit(1);
    }
    for (i=0; i<n; i++)
        scanf("%d", &vet[i]);
    for (i=0; i<n; i++)
        printf("%d ", vet[i]);
    free(vet);
}
```

# Alocação dinâmica de strings

```
void main(void) {
    char sir1[40];
    char *sir2;
    printf("Informe uma string: \n");
    scanf("%40s", sir1);
    if ((sir2=malloc(strlen(sir1)+1))==NULL) {
        printf("Erro na alocação de memória !\n");
        exit(1);
    }
    strcpy(sir2, sir1);
    printf("A cópia e: %s \n", sir2);
    .....
}
```

# Alocação dinâmica de matrizes (ponteiros para ponteiros)

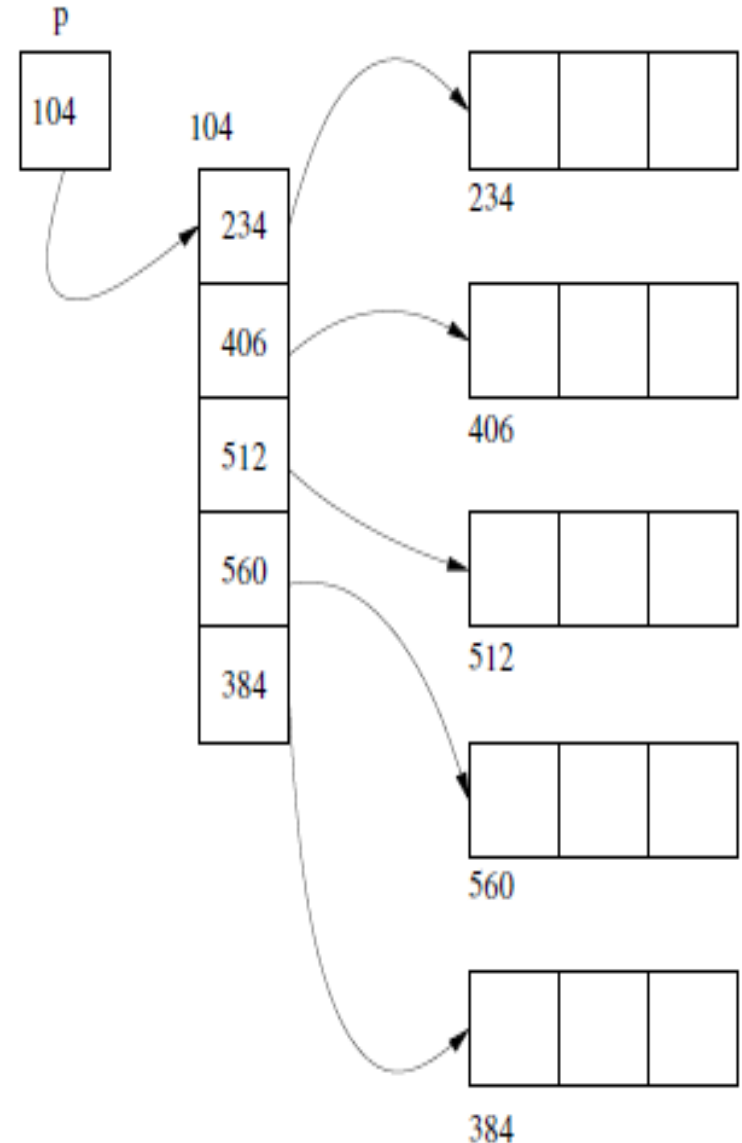
Exemplo:

```
int **p, i;
```

```
p ← aloque (5) /*  
alocando 5 posições de  
memória para p apontar*/  
para (i ← 0; i < 5; i ← i+1)
```

```
    p[i] ← aloque (3)
```

```
/*alocando 3 posições de  
memória para p[i]  
apontar*/
```





# Alocação dinâmica de matrizes

Inteiro: \*\*matriz, m, n, i, j

Escreva ("Entre com o numero de linhas: ")

Leia( m)

Escreva ("Entre com o numero de colunas: ")

Leia (n)

matriz ←aloque(m)

para (i ← 0; i < m; i ←i+1)

matriz[i] ←aloque (n)

# Alocação dinâmica de matrizes

```
Escreva("Obtendo os valores da matriz.")
```

```
para (i ← 0; i < m; i ← i+1)
```

```
    para (j ← 0; j < n; j ← j+1)
```

```
        leia (matriz[i][j])
```

```
Escreva ("Imprimindo a matriz")
```

```
for (i ← 0; i < m; i ← i+1)
```

```
    for (j ← 0; j < n; j ← j+1)
```

```
        escreva(matriz[i][j])
```

```
escreva("Desalocando a matriz.")
```

```
para (i ← 0; i < m; i ← i+1)
```

```
libere(matriz[i])
```

# Alocação dinâmica de matrizes

```
int m, n, i,j, **mat;
printf("Informe numero de linhas: \n");
scanf("%d", &m);
if ((mat=malloc(m * sizeof(int)))==NULL) {
    printf("Erro de alocação de memoria !\n");
    exit(1);
}
printf("Informe numero de colunas: \n");
scanf("%d", &n);
for (i=0; i<m; i++)
    if ((mat[i]=malloc(n * sizeof(int)))==NULL) {
        printf("Erro de alocação de memoria !\n");
        exit(1);
    }
printf ("Lendo matriz\n");
```

# Alocação dinâmica de matrizes

```
printf ("Lendo matriz\n");
for (i=0; i<m; i++){
    for (j=0; j<n; j++)
        scanf("%d", &mat[i][j]);

}
for (i=0; i<m; i++){
    for (j=0; j<n; j++){
        printf("%d ", mat[i][j]);
    }
    printf ("\n");
}
free(mat);
}
```

# Alocação dinâmica

- função **calloc**: aloca blocos de bytes em memória.
  - A sintaxe da `calloc()` é: `void *calloc(numero, tamanho_em_bytes);`
  - Ao inicializar o bloco de memória, a função `calloc` atribui o valor 0(zero) para cada elemento.
  - Com `malloc` faríamos: `malloc(numero * tamanho_em_bytes)` e ainda não teríamos os elementos inicializados com zero.

```
int *v, n = 2, i;
```

```
v = calloc(n,sizeof (int));
```

```
if(v == NULL){
```

```
    printf("Espaço Insuficiente para alocar \n"); exit(1);
```

```
}else{
```

```
    for(i = 0; i < n; i++)
```

```
        printf("\n vet[i]=%d ",v[i]);
```

```
}
```

- Troque `calloc` por `malloc` no exemplo acima e veja a diferença.

# Alocação dinâmica

- função **realloc**: aumenta ou reduz o tamanho do bloco de memória previamente alocado com malloc ou calloc.
  - para usar realloc é necessário ter um ponteiro que foi usado para alocar um espaço de memória.
  - Seja ptr esse ponteiro, então a sintaxe da realloc() é:  
void \*realloc(ptr, tamanho\_em\_bytes);

```
int *v;
```

```
v = malloc (1000 * sizeof (int));
```

```
for (int i = 0; i < 990; i++)
```

```
    scanf ("%d", &v[i]);
```

```
v = realloc (v, 2000 * sizeof (int));
```

```
for (int i = 990; i < 2000; i++)
```

```
    scanf ("%d", &v[i]);
```

\* **Perceba que nos compiladores atuais não precisam cast para malloc, realloc e calloc.**

# Função que retorna um ponteiro

```
int *soma_vetor4(int *a, int *b, int n) {  
    int * r;  
    r=malloc(sizeof (int) * n);  
    if (r==NULL) exit(1);  
    int i;  
    for (i=0; i<n; i++, a++, b++)  
        r[i]=*a+*b;  
    return r;  
}
```

```
...  
int a[3]={1,2,3};  
int b[3]={4,5,6};  
int * rr;  
rr=soma_vetor4(a,b,3);  
imprime_vetor(rr,3);
```

# Exemplo – pilhas usando vetores

FIFO – First Input First Output (Fila)

FILO – First Input Last Output (Pilha)

Exemplo de Pilha



↑  
pl

↑  
tos



# Exemplo – pilhas usando vetores

```
int stack[50], *tos,*pl;
void main(void){
    int value;
    tos = stack;
    /*Faz tos conter o topo da pilha */
    pl = stack; /*inicializa pl */
    do {
        printf("Entre com o valor:
(0) Pop, (-1) Sair, Caso contrário Push ");
        scanf("%d",&value);
        if(value!=0)
            push(value);
        else
            printf("Valor do topo é %d\n",pop());
    }while(value != -1);
}
```

# Exemplo – pilhas

```
void push(int i){
    int *p=tos;
    pl++;
    if(pl==(tos+50)){
        printf("Estouro da pilha");
        exit(1);
    }
    *pl = i;
    for ( ;p<=pl ; p++)
        printf ("%d\t", *p);
}
```

```
int pop(void){
    int *p=tos;
    if(pl==tos){
        printf("Pilha vazia");
        exit(1);
    }
    pl--;
    for ( ;p<=pl ; p++)
        printf ("%d\t", *p);
    return *(pl+1);
}
```

# Ponteiros para registros

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

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

```
typedef struct {
```

```
    double x;
```

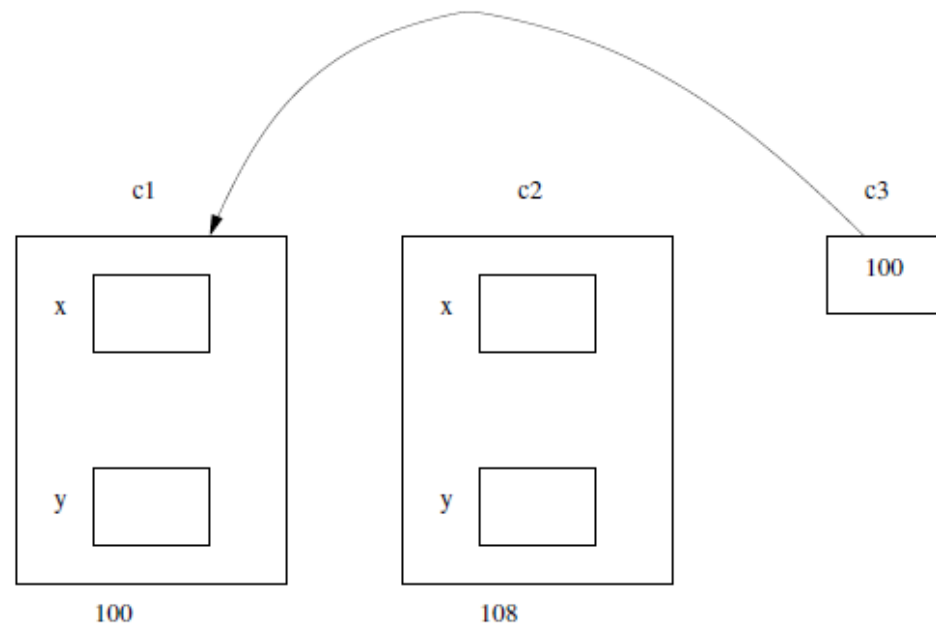
```
    double y;
```

```
} ponto2d;
```

```
int main(){
```

```
    ponto2d c1, c2, *c3;
```

```
    c3 = &c1;
```



# Ponteiros para registros

- Para acessarmos os campos de uma variavel struct via um ponteiro, podemos utilizar o operador \* juntamente com o operador . como de costume:

```
ponto2d c1, *c3;
```

```
c3 = &c1;
```

```
(*c3).x = 1.5;
```

```
(*c3).y = 1.5;
```

- Em C também podemos usar o operador -> para acessar campos de uma estrutura via um ponteiro. Podemos obter o mesmo resultado do exemplo anterior:

```
ponto2d c1, *c3;
```

```
c3 = &c1;
```

```
c3->x = 1.5; c3->y = 1.5;
```

# Ponteiros para registros

- Resumindo: Para acessar campos de estruturas via ponteiros use um dos dois:
- `ponteiroEstrutura->campo`
- `(*ponteiroEstrutura).campo`

## Exercício 1:

- Vamos criar as seguintes funções:
- `void lePonto3D(ponto3d *f);` lê dados de uma ponto no espaço tridimensional com coordenadas (x,y,z) passada como ponteiro. (Por quê como ponteiro?)
- `void imprimePonto3D(ponto3d f);` Imprime coordenadas de um ponto.
- O programa principal que requeira a leitura de 5 pontos no espaço tridimensional.

# Exemplo – Ponteiros para registros

```
#include <stdio.h>
typedef struct {
    float x;
    float y;
    float z;
} ponto3d;
void lePonto3D(ponto3d *);
void
imprimePonto3D(ponto3d );
int main(){
    int i;
    ponto3d p[5];
    for (i=0;i<5; i++){
        lePonto3D(&p[i]);
        imprimePonto3D(p[i]);
    }
}
```

```
void lePonto3D(ponto3d *f){
    printf ("\nInforme as
coordenadas (x,y,z) do ponto: ");
    scanf ("%f %f %f", &(f->x), &(f-
>y), &(f->z));
}
void imprimePonto3D(ponto3d
f){
    printf ("%f %f %f", f.x, f.y, f.z);
}
}
```

# Ponteiros para registros

Faça uma função que retorne o produto escalar de dois pontos no espaço 3D.

onde  $\text{produto\_escalar} = \langle v1, v2 \rangle = v1.x * v2.x + v1.y * v2.y + v1.z * v2.z$

```
float ProdutoEscalar(ponto3d v1, ponto3d v2){  
    float pe;  
    pe=v1.x*v2.x + v1.y*v2.y + v1.z*v2.z;  
    return (pe);  
}
```

# Exemplo – Ponteiros para registros

```
#include <stdio.h>
int main(){
    int i;
    ponto3d p[5];
    for (i=0;i<5; i++){
        lePonto3D(&p[i]);
        imprimePonto3D(p[i]);
    }
    for (i=0; i<4 ; i++)
        printf ("\nProduto escalar
entre p[%d] e p[%d] = %f", i,
i+1,
ProdutoEscalar(p[i],p[i+1]));
}
```

```
float ProdutoEscalar(ponto3d
f1,ponto3d f2){
    float pe;
    pe=f1.x*f2.x + f1.y*f2.y +
f1.z*f2.z;
    return (pe);
}
```



# Ponteiros para registros

Exemplo de alocação dinâmica de um vetor de registros (usando malloc e realloc). O programa aloca memória para um vetor de 3 elementos de tipo cad\_aluno e depois realoca para mais três.

```
typedef struct {
    char nome[10];
    int rgm;
}cad_aluno;

int main(){
    cad_aluno *v;
    int i;
    v = malloc (3 * sizeof (cad_aluno));
    for (i = 0; i < 3; i++){
        printf ("v[%d].nome=",i);
        scanf ("%s", v[i].nome);
        printf ("v[%d].rgm=",i);
        scanf ("%d", &(v[i].rgm));
    }
}
```

```
printf ("Realocando memoria\n");
v = realloc (v, 6 * sizeof (cad_aluno));
for (i = 3; i < 6; i++){
    printf ("v[%d].nome=",i);
    scanf ("%s", v[i].nome);
    printf ("v[%d].rgm=",i);
    scanf ("%d", &(v[i].rgm)); }
}
```