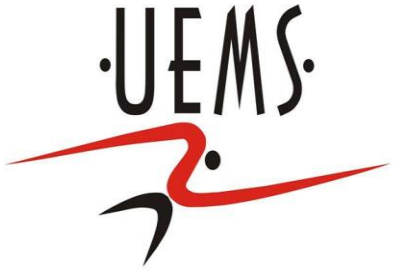


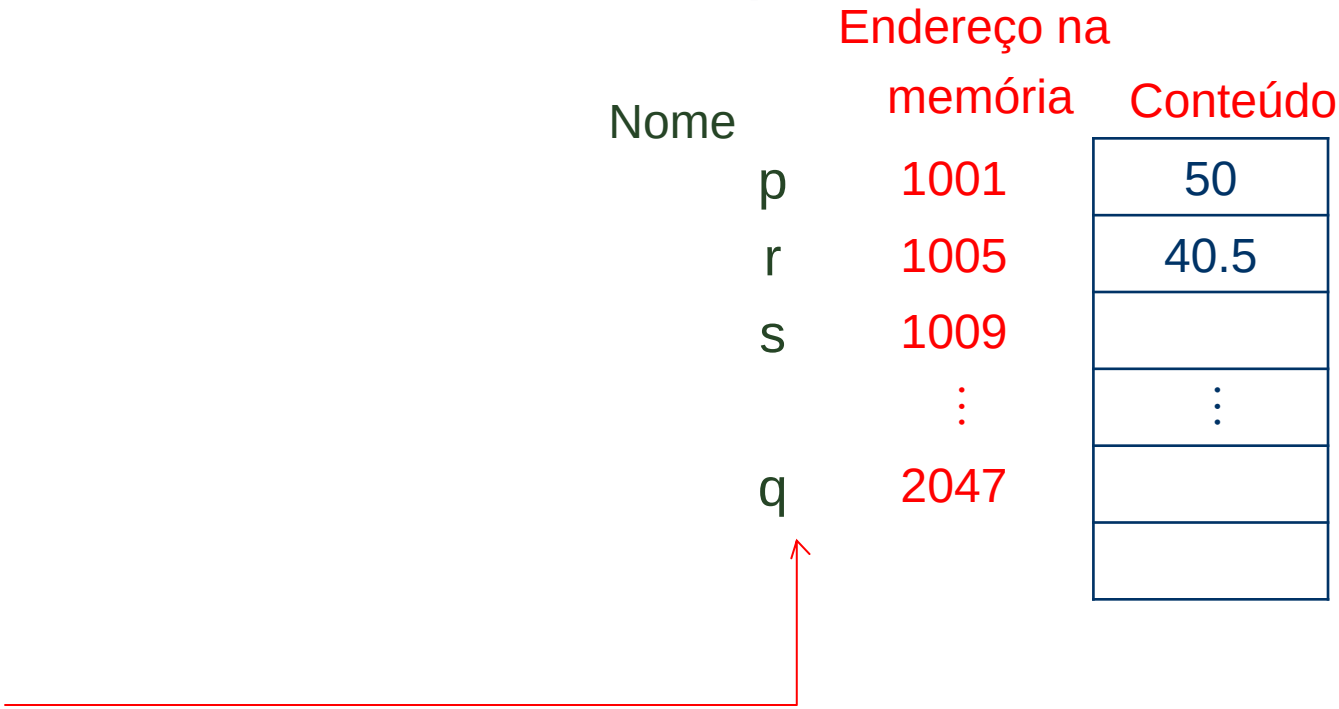
Programação de Computadores I – Ponteiros

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 é alocado para ela.

```
int p,q,s
float r
p =50
r =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 (0xbfd267c4);

• 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);
}
```

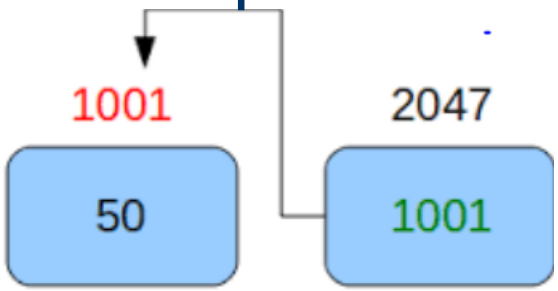
Ponteiros - Conceito

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

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

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

- 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.

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

Ponteiros - Declaração

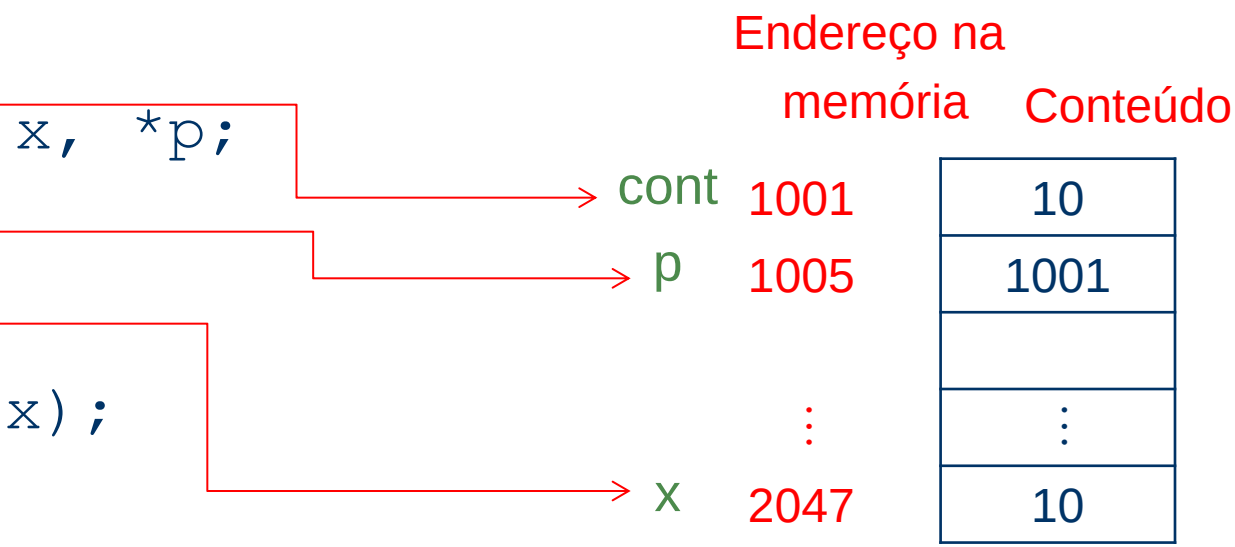
- Ponteiro tem um tipo associado que determina o tipo da dado contido no endereço apontado. Pode ser um tipo pré-definido da linguagem ou um tipo definido pelo programador. Exemplo: inteiro, literal, real, TipoProduto,etc.
- Utilizamos o operador unário *

```
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.

```
int cont =10, x, *p;
p = &cont;
x = *p;
printf (cont,x);
```



Ponteiros – Fazendo acesso aos valores das variáveis referenciadas

```
int main() {
   int x; int *p_x = &x; x = 10;
  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);
}
```

Ponteiro Nulo

- Quando um ponteiro não esta 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

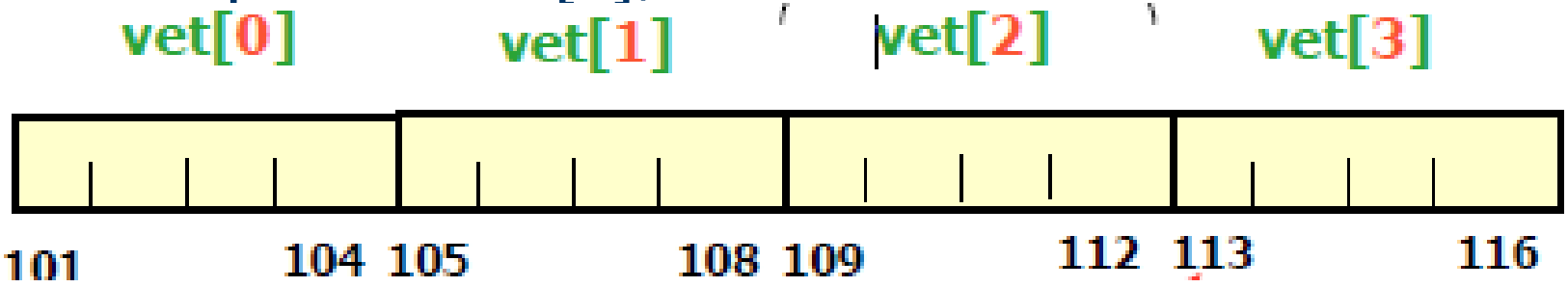
- Os operadores + e – e os operadores ++ e --, podem ser utilizados com ponteiros.
- Considere um computador de 32 bits no qual um inteiro ocupa 4 bytes e considere p um ponteiro para um inteiro que aponta para um endereço 1000.
- A expressão: p++; faz com que o conteúdo de p seja alterado para 1004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

Aritmética de Ponteiros

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

Ponteiros e Vetores

- Quando declaramos uma variável do tipo vetor, aloca-se uma quantidade de memoria contígua cujo tamanho e 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, v e &v[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 uma variável vetor, diferentemente de um ponteiro, possui um endereço fixo (o do começo do vetor), portanto, não podemos alterar o seu endereço.

Ponteiros e Vetores

- Aritmetica 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 quinto elemento de vet, podemos escrever: c = p[4]; ou também c=*(p+4);

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 v[] = {10, 20, 30, 40, 50};
   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);
}
```

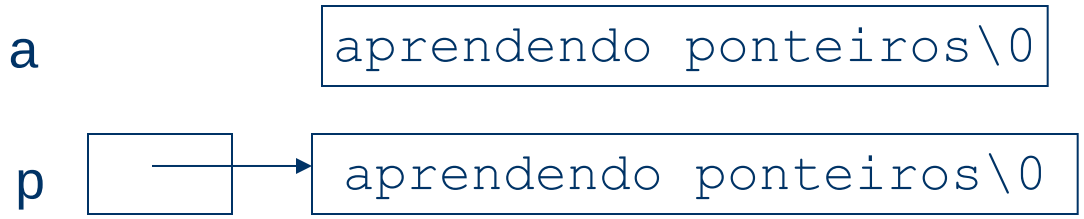
Mais aritmética de Ponteiros

- **Comparação de dois ponteiros.**
  - ***Se p e q aponta 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 *s){
      char *p = s;
      while (*p != '\0')
            p++;
      return p - s;
}
```

Ponteiros - Exercícios

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

Expressão	Valor
p==&x	1
**&p	3
r=&z	erro
7**p/*q+7	11
(&y)=*p	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 para ponteiros

- Um ponteiro para um ponteiro é como se você anotasse o endereço de um papel que tem o endereço da casa do seu amigo.
- 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

- 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 definirmos um tamanho muito maior 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 dinámica

- função **malloc**
- – Parâmetro: tamanho desejado, em bytes
- – Resultado: apontador para espaço na memória

Programa

```
int *p;
p = (int*)malloc(4);
*p = 5;
printf("%d", *p);
```

Alocação dinámica

- função **free**:

– Parâmetro: endereço já alocado

– Resultado: libera a área alocada

Programa

```
int *p;
p = (int*)malloc(4);
*p = 5;
free(p);
```

Alocação dinámica

função **sizeof**:

- sizeof(variavel)

– Retorna quantidade de memória ocupada pela variável

- sizeof(tipo)

– Retorna quantidade de memória ocupada por uma variável com este tipo.

VETOR DINÂMICO

• Calcular tamanho do vetor com **sizeof**

• Reservar memória com **malloc**

• Acessar elementos com **[ ]** ou ponteiros

• Liberar memória do vetor com **free**

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=(int *)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 dinámica de strings

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

Função que retorna um ponteiro

```
int * soma_vetor4(int *a, int *b, int n) {
    int * r;
    r=(int *) 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

```
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: ");
       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){
        pl++;
        if(pl==(tos+50)){
            printf("Estouro da pilha");
            exit(1);
        }
        *pl = i;
}
int pop(void){
        if(pl==tos){
            printf("Pilha vazia");
            exit(1);
        }
        pl--;
        return *(pl+1);
}
```