Ponteiros - Parte I

Ponteiros, Ponteiros e Vetores, Algoritmo Bubble Sort, Alocação Dinâmica de Memória

Ponteiros

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

- Esse endereço é normalmente a posição de uma outra variável; assim a primeira variável é dita para **apontar** para a segunda.

- Se uma variável irá conter um ponteiro, ela deve ser declarada. Uma declaração de ponteiro consiste da seguinte forma:

<tipo> * <nome_variável>

Ponteiros

- Existem dois operadores para ponteiros:
  * e &.

- & Representa um operador unário que devolve o endereço na memória do seu operando.

- Exemplo:**Coloca em m o endereço de memória de count.**

```
m = &count
```

- * É o complemento de &, devolve o valor da variável localizada no endereço.

- Exemplo:**Coloca o valor de count em q.**

```
q = *m
```

Ponteiros

Declaração e inicialização		
int i=3, j = 5, *p = &i, *q=&j, *r;		
double x;		
Expressão	**Expressão Equivalente**	**Valor**
p == & i	p == (&i)	1
* * & p	* (* (& p))	3
r = & x	r = (& x)	/* ilegal*/
7 * * p / * q + 7	(((7 * (* p))) / (* q)) +7	11
* (r = & j) *= * p	(* (r = (& j))) *= (* p)	15

Cuidado: (7 * * p / * q + 7) ≠ (7 * * p / * q + 7)

Ponteiros

```
int main(){
  int a; *aPtr;
  a=7;
  aPtr=&a;
  printf("O endereço de a é %p\n O valor de aPtr é %p",&a,aPtr);
  printf("O valor de a é %d\n O valor de *aPtr é %d",a,*aPtr);
  printf("\n&*aPtr=%p\n*&aPtr=%p\n",&*aPtr,*&aPtr);
  return 0;
}
```

O endereço de a é 0012FF7C
O valor de aPtr é 0012FF7C
O valor de a é 7
O valor de *aPtr é 7
&*aPtr =0012FF7C
*&aPtr=0012FF7C

Ponteiros

- Uma declaração **void * <nome_ptr>** cria um ponteiro do tipo genérico.
- Conversões durante atribuição entre ponteiros diferentes normalmente eram permitidas no C tradicional.
- Porém, conversões de tipo envolvendo ponteiro não são permitidas no padrão ANSI C.
- No ANSIC, a conversão ocorre apenas se um dos tipos é um ponteiro void ou o lado direito é uma constante 0.

Ponteiros

Declaração	
int *p; float *q; void *v;	
Permitido	**Não permitido**
p=0;	p = 1;
p = (int *) 1;	v = 1;
p = v = q;	p = q;
p = (int *) q;	

Ponteiros e Vetores

- Um ponteiro pode acessar diferentes endereços de memória.

- Uma determinada posição em um vetor é um endereço ou um ponteiro que está fixo.

- Considere o vetor **a[ ],** onde **a** aponta para a posição i=0. Temos que:

  a[i] ⇔ *(a + i)

- Considere o ponteiro **p**, também temos que:

  p[i] ⇔ *(p + i)

Ponteiros e Vetores

- Incrementando ou decremento ponteiros
- Exemplo

```
int vet[100], *p_vet;
p_vet =  vet;

p_vet++; // aponta próximo elemento do vetor

p_vet--; // aponta elemento anterior do vetor

p_vet += 4; //aponta posição atual do vetor+4;
```

Ponteiros e Vetores

```
int vet[] = {10,11,12,13}, *p_vet, cnt;

  p_vet =  vet;
  for(cnt=0; cnt<3; cnt++){
     printf("\n %d", *p_vet++);
     }

  p_vet =  vet;
     for(cnt=0; cnt<3; cnt++){
     printf("\n %d", *++p_vet);
     }
```

Qual a diferença do resultado impresso pelos dois“printf”?

Ponteiros e Vetores

- Considere as declarações abaixo:

```
# define N  100
int  a[N], i, *p, sum=0;
```

- Temos que:

```
p = a ⇔ p=&a[0];
p = a+1 ⇔ p=&a[1];
```

- Suponha que a[N] tenha sido inicializado. As rotinas abaixo são equivalentes.

```
for (p=a; p < & a[N]; ++p)
  sum += *p;
```

```
for (i=0; i < N; ++i)
    sum += *(a+i);
```

```
p=a;
for (i=0; i < N; ++i)
    sum += p[ i ];
```

Ponteiros e Vetores

- No exemplo anterior, o vetor **a[N]** tem o identificador **a** como um ponteiro constante.

- Logo, as expressões abaixo são ilegais:

  a = p ++a a+=2 &a

- Não é possível mudar o valor de **a**.

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

int sumVet(int *, int );

int main()
{
 int v[10]={2,2,2,2,2,2,2,2,2,2};
 int i;
 int sum;
 sum = sumVet(v,10);
 printf("sum1=%d\n",sum);
 sum = sumVet(&v[5],5);
 printf("sum2=%d\n",sum);
 sum = sumVet(&v[2],3);
 printf("sum3=%d\n",sum);

}

int sumVet(int *p, int n)
{
  int i=0, sum=0;
  while(i++<n){
    sum += *p++;
  }
return sum;
}

```

Método da Bolha (Bubble Sort)

- A cada passagem pelo vetor, o elemento da i-ésima posição é selecionado e transferido para a sua posição adequada.

VETOR DESORDENADO

[15 46 91 59 62 76 10 93]

[15 46 91 59 62 76 10 93]

[15 46 91 59 62 10 76 93]

[15 46 91 59 10 62 76 93]

[15 46 91 10 59 62 76 93]

[15 46 10 91 59 62 76 93]

[15 10 46 91 59 62 76 93]

[10 15 46 91 59 62 76 93]

[10 15 46 59 91 62 76 93]

[10 15 46 59 62 91 76 93]

[10 15 46 59 62 76 91 93]

VETOR ORDENADO

Método da Bolha (Bubble Sort)

```
Funcao_Ordena_Bolha(inteiro vetor[], inteiro tam)
Inicio
      inteiro i, j, aux;
      para i de 0 até tam faça
            para j de tam-1 até i passo –1 faça
                  se (vetor[j-1] > vetor[j]
                    aux=vetor[j-1];
                    vetor[j-1]=vetor[j];
                    vetor[j]=aux;
                  fim-se;

            fim-para;
      fim-para;
Fim.
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<time.h>
#define MAX 100
#define Aleatorio() (-10+rand()%(((10)-(-10))+1))

void iniciaVet(int *vector, int n);
void imprimeVet(int *vector, int n);
void bubbleSort(int *, int );

int main(){
  int v[MAX], n=5;
  srand(time(NULL));
  iniciaVet(v,n);
  imprimeVet(v,n);
  bubbleSort(v,n);
  imprimeVet(v,n);
  return 0;
}
```

```
21
22  void iniciaVet(int *vector, int n){
23   int i;
24   for(i=0; i<5; i++){
25    vector[i] = Aleatorio();
26   }
27  }
28
29  void imprimeVet(int *vector, int n){
30   int i;
31   printf("[");
32   for(i=0; i<n; i++){
33    printf(" %d",vector[i]);
34   }
35   printf("]\n");
36  }
37
38  void bubbleSort(int *vector, int size){
39
40   int i, j, aux;
41  for(i=0; i < size-1; i++){
42    for(j=size-1;j>i;--j){
43      if(vector[j-1]>vector[j]){
44        aux = vector[j-1];
45        vector[j-1] = vector[j];
46        vector[j]= aux;
47      }
48    }
49  }
50  }
```

Alocação Dinâmica de Memória

- Disponibiliza espaços contíguos de memória.
- malloc(), calloc(), realloc(), free()
- São funções utilizadas para trabalhar com alocação dinâmica (em tempo de execução) de memória.

Alocação Dinâmica de Memória

- Malloc

```
void *malloc(size_t size);
```

- size = tamanho do bloco de memória em bytes.

- size_t é um tipo pré-definido usado em stdlib.h que faz size_t ser equivalente ao tipo unsigned int.

```
typedef unsigned int size_t
```

- Retorna um ponteiro para o bloco de memória alocado.

Alocação Dinâmica de Memória

- Malloc

```
void *malloc(size_t size);
```

- Quando não consegue alocar memória, retorna um ponteiro nulo.
- A região alocada contém valores desconhecidos
- Sempre verifique o valor de retorno!

Alocação Dinâmica de Memória

```
#include <stdlib.h>

char *str;
str = (char *)malloc(100));
if(str == NULL)
{
 printf("Espaco insuficiente para
  alocar buffer \n");
   exit(1);
}
printf("Espaco alocado para str\n");
```

Alocação Dinâmica de Memória

```
#include <stdlib.h>
int *num;
num = (int *)malloc(50 * sizeof(int);
if(num==NULL)
{
  printf("Espaco insuficiente para alocar
  buffer \n");
  exit(1);
}
printf("Espaco alocado para num\n");
```

Alocação Dinâmica de Memória

- Calloc

```
void * calloc ( size_t  num, size_t  size );
```

- A função calloc() aloca um bloco de memória para um “array” de *num* elementos, sendo cada elemento de tamanho *size*.

- A região da memória alocada é inicializada com o valor zero

- A função retorna um ponteiro para o primeiro byte

- Se não houver alocação, retorna um ponteiro nulo

Alocação Dinâmica de Memória

```
#include <stdlib.h>
unsigned int num;
int *ptr;
printf("Digite o numero de variaveis inteiras: ");
scanf("%d", &num);
ptr = (int *)calloc(num, sizeof(int);
if(ptr == NULL)
{
   printf("Espaco insuficiente para alocar \“num\”
  \n");
   exit(1);
}
printf("Espaco alocado com o calloc\n");
```

Alocação Dinâmica de Memória

- `calloc(n, sizeof(int));` ⇔ `malloc(n*sizeof(int));`
- A função malloc() não inicializa o espaço disponibilizado em memória.
- A função calloc() inicializa com valor zero.
- Em programas extensos, malloc() pode levar menos tempo do que calloc().
- As duas funções retornam um ponteiro do tipo void em caso de sucesso.
- Caso contrário, NULL é retornado.

Alocação Dinâmica de Memória

- Realloc

```
void * realloc (void * ptr, size_t size );
```

- A função realloc() aumenta ou reduz o tamanho de um bloco de memória previamente alocado com malloc() ou calloc()

- O argumento *ptr* aponta para o bloco original de memória.

- O argumento *size* indica o novo tamanho desejado em bytes

Alocação Dinâmica de Memória

- Se houver espaço para expandir, a memória adicional é alocada e prt é retornado.
- Se não houver espaço suficiente para expandir o bloco atual, um novo bloco de tamanho size é alocado em outra região da memória.
- O conteúdo do bloco original é copiado para o novo bloco.
- O espaço de memória do bloco original é liberado e a função retorna um ponteiro para o novo bloco.

Alocação Dinâmica de Memória

- Se o argumento size for zero, a memória indicada por ptr é liberada e a função retorna NULL.

  - Se não houver memória suficiente para a realocação (nem para um novo bloco), a função retorna NULL e o bloco original permanece inalterado.

  - Se o argumento ptr for NULL, a função atua como malloc().

Alocação Dinâmica de Memória

- Exemplo:

```
unsigned int num; int *ptr;
 printf("Digite o numero de variaveis do tipo int: ");
 scanf("%d", &num);

ptr = (int *)calloc(num, sizeof(int);
 if(ptr == NULL){
    printf("Espaco insuficiente para alocar \"num\" \n");
        exit(1);
 }

//duplica o tamanho da região alocada para ptr
ptr = (int *)realloc(ptr, 2*num*sizeof(int);
 if(ptr == NULL){
        printf("Espaco insuficiente para alocar \"num\"
\n");
    exit(1);
}
printf("Novo espaço \“realocado\“ com sucesso\n");
```

Alocação Dinâmica de Memória

- Free

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

- O espaço alocado dinamicamente com calloc() ou malloc() não retorna ao sistema quando o fluxo de execução deixa uma função.

- A função free() “desaloca”/libera um espaço de memória previamente alocado usando *malloc*, *calloc* ou *realloc*.

- O espaço de memória fica disponível para uso futuro.

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

#define Aleatorio() (-10+rand()%(((10)-(-10))+1))

int *criaVetor(int *);
int *criaVetor_B(int *);
void imprimeVet(int *vector, int n);

int main(){
   int *v, n;
   srand(time(NULL));
   v=criaVetor_B(&n);
   imprimeVet(v,n);
   free(v);
   return 0;
}
```

```c
int *criaVetor(int *size){
  int *vector, i;
  printf("Tamanho vetor:");
  scanf("%d",size);
  vector = (int *) malloc((*size)*sizeof(int));
  if(vector==NULL){
    printf("Erro na alocação!!!");
    exit(1);
  }
  for(i=0; i<(*size); i++){
   vector[i]=Aleatorio();
  }
  return vector;
}
int *criaVetor_B(int *size){
  int *vector=NULL,aux;
  *size = 0;
  aux=Aleatorio();
  while(aux!=-1){
       *size=*size+1;
       vector = (int *) realloc(vector,(*size)*sizeof(int));
       if(vector==NULL){
        printf("Erro na alocação!!!");
        exit(1);
       }
       vector[*size-1]=aux;
       aux = Aleatorio();
  }
  return vector;
}
```