

ESTRUTURA DE DADOS

PONTEIROS

Profa. Fabrícia Damando Santos

fabriadamando@gmail.com

Introdução

2

- Servem para dar **apoio** às rotinas de **alocação** dinâmica de **memória**
- Fornecem as maneiras pelas quais as funções podem modificar o conteúdo de variáveis locais das funções chamadoras
- Podem substituir as matrizes para aumentar a eficiência

- Em geral, interessa ao **programador** apenas os nomes simbólicos que representam as informações, pois é com estes nomes que são realizadas as operações do seu algoritmo.
-
- Porém, ao **processador** interessa os endereços dos blocos de informação pois é com estes endereços que vai trabalhar.

Introdução

4

- Toda **informação** (dado armazenado em variável simples ou vetor) que **manipulamos** em um programa está armazenado na **memória** do computador.
- Cada informação é representada por um certo conjunto de *bytes*
 - ▣ Por exemplo:
 - caracter (char): 1 *byte*,
 - inteiro (int): 2 *bytes*,
 - etc.

Introdução

5

- Conjuntos de *bytes*, que chamaremos de **bloco**, tem um nome e um **endereço** de localização específica na memória.
 - **Exemplo:** Observe a instrução abaixo:
 - `int num = 17;`
 - ao interpretar esta instrução, o processador pode especificar:
 - Nome da informação: num
 - Tipo de informação: int
 - Tamanho do bloco (número de *bytes* ocupados pela informação): 2
 - Valor da informação: 17
 - Endereço da informação (localização do primeiro *byte*): 8F6F:FFF2 (hexadecimal)

Introdução

6

- Uma variável do tipo **PONTEIRO** é capaz de armazenar um **endereço de memória**
- Sempre que uma variável contiver um endereço de memória, dizemos então que tal variável **aponta** para determinado endereço de memória

- Declaração:

```
char *a;  
int *b;
```

End:	Conteúdo da Memória:
0000h	
0001h	
...	
FFFFh	

- Ponteiros são variáveis que contém endereços.
- Neste sentido, estas variáveis *apontam* para algum determinado endereço da memória.
- Em geral, o ponteiro aponta para o endereço de alguma variável declarada no programa.

- Quando trabalhamos com ponteiros, queremos fazer duas coisas basicamente:
 - ▣ conhecer **endereço** de uma variável;
 - ▣ conhecer o **conteúdo** de um endereço.

- Para realizar estas tarefas a linguagem C nos providencia dois operadores especiais:
 - ▣ o operador de **endereço**: &
 - ▣ o operador de **conteúdo**: *

Sintaxe

9

- Quando declaramos um ponteiro, devemos declará-lo com o mesmo tipo (int, char, etc.) do bloco a ser apontado.

- **Sintaxe:** A sintaxe da declaração de um ponteiro é a seguinte:
 - `tipo_ptr *nome_ptr_1;`
 - ou
 - `tipo_ptr* nome_ptr_1, nome_ptr_2, ...;`
 - Onde:
 - `tipo_ptr` : é o tipo de bloco para o qual o ponteiro apontará.
 - `*` : é um operador que indica que `nome_ptr` é um ponteiro.
 - `nome_ptr_1, nome_ptr_2, ...`: são os nomes dos ponteiros.

Exemplo

10

□ Exemplo:

- `int *p;`
- `float* s_1, s_2;`

□ Observe:

- A primeira instrução declara um ponteiro chamado `p` que aponta para um inteiro. Este ponteiro aponta para o **primeiro** endereço de um bloco de **dois bytes**.
- Sempre é necessário declarar o tipo do ponteiro. Neste caso dizemos que declaramos um ponteiro tipo `int`.
- A segunda instrução declara dois ponteiros (`s_1` e `s_2`) do tipo `float`.
- Observe que o `*` está junto ao tipo: assim todos os elementos da lista serão declarados ponteiros

Operador &

11

- O operador '&' é capaz de fornecer o endereço de uma variável.

Ex:

...

```
int a, *ptr;
```

```
a = 3;
```

```
ptr = &a;
```

...

End:

A01Ch

3

- O operador de **endereço (&)** determina o endereço de uma variável
 - ▣ Por exemplo, `&val` determina o endereço do bloco ocupado pela variável `val`.

- Esta informação não é totalmente nova pois já a usamos antes: na função `scanf()`.
 - ▣ **Exemplo:** Quando escreve-se a instrução:
 - ▣ `scanf("%d", &num);`

 - ▣ estamos nos referimos **endereço** do bloco ocupado pela variável `num`.
 - ▣ A instrução significa:

"leia o buffer do teclado, transforme o valor lido em um valor inteiro (2 bytes) e o armazene no bloco localizado no endereço da variável `num`".

Atribuição

13

- **Exemplo:** Para se atribuir a um ponteiro o endereço de uma variável escreve-se:
 - `int *p, val=5; //` declaração de ponteiro e variável
 - `p = &val; //` atribuição

Operador *

14

- O operador '*' pode ser utilizado para inserir um conteúdo (valor) no endereço apontado pelo ponteiro

Ex:

```
...  
int a, *ptr;  
a = 3;  
ptr = &a;  
...
```

End:
A01Ch

3

```
...  
int a, *ptr;  
ptr = &a;  
*ptr = 3;  
...
```

- O operador **conteúdo** (*) determina o conteúdo (valor) do dado armazenado no endereço de um bloco apontado por um ponteiro.
- Por exemplo, *p determina conteúdo do bloco apontado pelo ponteiro p.
- De forma resumida: **o operador (*) determina o conteúdo de um endereço.**
- **Exemplo:** Para se atribuir a uma variável o conteúdo de um endereço escreve-se:
 - ▣ `int *p = 0x3f8, val; //` declaração de ponteiro e variável
 - ▣ `val = *p; //` atribuição

Preste Atenção



16

- ❑ O operador **endereço** (&) somente pode ser usado em uma única variável.
- ❑ Não pode ser usado em expressões como, por exemplo, &(a+b).
- ❑ O operador **conteúdo** (*) somente pode ser usado em variáveis ponteiros.

Operações com ponteiros

17

- A um ponteiro pode ser atribuído o endereço de uma variável comum.

▣ **Exemplo:** Observe o trecho abaixo:

...

```
int *p;
```

```
int s;
```

```
p = &s; // p recebe o endereço de s
```

...

- Um ponteiro pode receber o valor de outro ponteiro, desde que os ponteiros sejam de mesmo tipo.

▣ **Exemplo:** Observe o trecho abaixo:

...

```
float *p1, *p2, val;
```

```
p1 = &val; // p1 recebe o endereço de val...
```

```
p2 = p1; // ...e p2 recebe o conteúdo de p1 (endereço de val)
```

- A um ponteiro pode ser atribuído o valor **nulo** usando a constante simbólica NULL (declarada na biblioteca `stdlib.h`).
- Um ponteiro com valor NULL não aponta para lugar nenhum!

▣ **Exemplo:** Observe o trecho abaixo:

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

```
...
```

```
char *p;
```

```
p = NULL;
```

```
...
```

- Uma quantidade inteira pode ser adicionada ou subtraída de um ponteiro.
- A adição de um inteiro n a um ponteiro p fará com que ele aponte para o endereço do n -ésimo bloco seguinte.

□ **Exemplo:** Observe o trecho abaixo:

```
...  
double *p, *q, var;  
p = &var  
q = ++p; // q aponta para o bloco seguinte ao ocupado por var  
p = q - 5; // p aponta para o quinto bloco anterior a q  
...
```

- Dois ponteiros podem ser comparados (usando-se operadores lógicos) desde que sejam de mesmo tipo.

□ **Exemplo:** Observe o trecho abaixo:

...

`if(px == py){ // se px aponta para o mesmo bloco que py ...`

`if(px > py){ // se px aponta para um bloco posterior a py ...`

`if(px != py){ // se px aponta para um bloco diferente de py ...`

`if(px == NULL) // se px é nulo...`

A pergunta que não quer calar...

22

- Porque usar ponteiros?
 - A primeira vantagem da utilização de ponteiros em programas talvez esteja relacionada a sua utilização como argumentos de funções.

Vamos executar esse código e entendê-lo

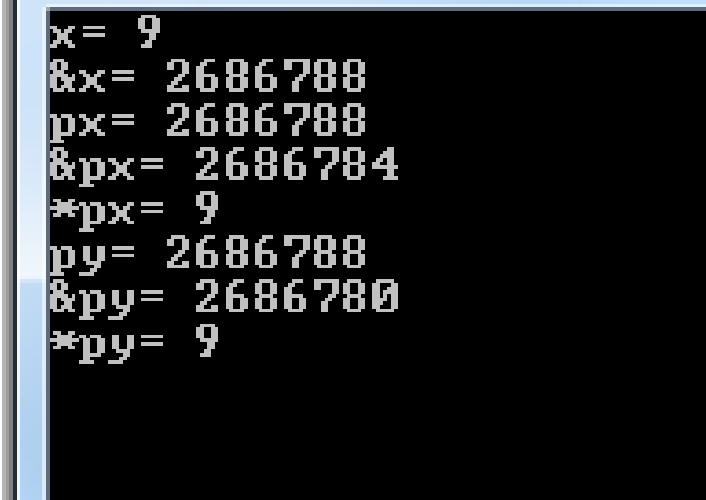
23

```
#include <stdio.h> //EXEMPLO DE PONTEIROS

main(){
int x,*px,*py;
x=9;
px=&x;
py=px;
printf("x= %d\n",x);
printf("&x= %d\n",&x);

printf("px= %d\n",px);
printf("&px= %d\n",&px);
printf("*px= %d\n",*px);

printf("py= %d\n",py);
printf("&py= %d\n",&py);
printf("*py= %d\n",*py);
getchar();}
```



```
x= 9
&x= 2686788
px= 2686788
&px= 2686784
*px= 9
py= 2686788
&py= 2686780
*py= 9
```

Passagem de dados por valor ou por referencia

24

- o valor de uma variável `var` de uma função `fun_1()` passada para uma outra função `fun_2()` **não** podem ser alterado pela função `fun_2()`.
- De fato, isto é verdade se passamos o **valor** da variável `var` para a função `fun_2()`. Mas o valor de `var` pode ser alterado por `fun_2()` passamos seu **endereço**.
 - ▣ No primeiro caso, dizemos que a passagem de dados de uma função para outra ocorreu por **valor**.
 - ▣ No segundo caso, dizemos que houve uma passagem por **referência**

Expressões com ponteiros

25

- Os ponteiros podem aparecer em expressões, se px aponta para um inteiro x , então $*px$ pode ser utilizado em qualquer lugar que x seria
- O operador $*$ tem maior precedência que as operações aritméticas, assim a expressão abaixo pega o conteúdo do endereço que px aponta e soma 1 ao seu conteúdo.
 - $y = *px + 1;$
- No próximo caso somente o ponteiro será incrementado e o conteúdo da próxima posição da memória será atribuído a y :
 - $y = *(px + 1);$

- Os incrementos e decrementos dos endereços podem ser realizados com os operadores ++ e --, que possuem precedência sobre o * e operações matemáticas e são avaliados da direita para a esquerda:
 - `*px++;` /* sob uma posição na memória */
 - `*(px--);` /* mesma coisa de `*px--` */

- No exemplo abaixo os parênteses são necessários, pois sem eles `px` seria incrementado em vez do conteúdo que é apontado, porque os operadores `*` e `++` são avaliados da direita para esquerda.
 - ▣ `(*px)++ /*` equivale a `x=x+1;` ou `*px+=1 /*`

```

#include <stdio.h>

//EXEMPLO DE PONTEIROS 2

main()
{int x,*px;

x=1;

px=&x;

printf("x= %d\n",x);

printf("px= %u\n",px);

printf("*px+1= %d\n",*px+1);

printf("px= %u\n",px);

printf("*px= %d\n",*px);

printf("*px+=1= %d\n",*px+=1);

printf("px= %u\n",px);

printf("(*px)++= %d\n",(*px)++);

printf("px= %u\n",px);

printf("*(px++)= %d\n",*(px++));

printf("px= %u\n",px);

printf("*px++-= %d\n",*px++);

26 printf("px= %u\n",px);

  getchar(); }

```

```

x= 1
px= 2293572
*px+1= 2
px= 2293572
*px= 1
*px+=1= 2
px= 2293572
(*px)++= 2
px= 2293572
*(*px++)= 3
px= 2293576
*px++-= 2293624
px= 2293580
_

```

Passagem por Valor

29

- ▣ A passagem por valor significa que passamos de uma função para outra o **valor** de uma variável, isto é, a função chamada recebe um cópia do valor da variável.
- ▣ Assim qualquer alteração deste valor, pela função chamada, será uma alteração de uma cópia do valor da variável.
- ▣ O valor original na função chamadora **não é alterado** pois o valor original e copia ficam em blocos de memória diferentes.

Passagem por Referencia

30

- ▣ A passagem por referencia significa que passamos de uma função para outra o **endereço** de uma variável, isto é, a função chamada recebe sua **localização** na memória através de um ponteiro.
- ▣
- ▣ Assim qualquer alteração no conteúdo apontado pelo do ponteiro será uma alteração no conteúdo da variável original
- ▣ O valor original é **alterado**

Exemplo

31

```
#include <stdio.h>

void troca(int *p1, int *p2);

int main(void){    // programa principal
int a=1,b=2;        // declaração das variáveis
//scanf("%d %d",&a,&b);    // leitura das variáveis
troca(&a,&b);        // passagem dos endereços de a e b
printf("Valor da variavel a:%d - Valor da variavel b: %d",a,b);
getchar();        // imprime valores (trocados!)
}

void troca(int *p1, int *p2) { // declaração da função
    // Observe: ponteiros
int temp;    // variável temporária
temp = *p1; // temp recebe o conteúdo apontado por p1
*p1 = *p2;  // o conteúdo de p1 recebe o conteúdo de p2
*p2 = temp; // o conteúdo de p2 recebe o valor de temp
}
```

Neste exemplo temos uma função `troca()` que troca entre si os valores de duas variáveis.

Esta função recebe os **endereços** das variáveis passadas pela função `main()`, armazenando-os nos ponteiros `p1` e `p2`.

Dentro da função, troca-se os **conteúdos** dos endereços apontados

Retorno de dados em funções

32

- A **passagem por referencia** permite que (formalmente) uma função retorne quantos valores se desejar
- Mas uma função pode retornar somente um valor.
- Isto continua sendo valido pois o C assim define funções.
- Porem com o uso de ponteiros pode-se contornar esta situação.

Estudo de caso

33

- Imagine que queremos escrever uma função `stat()` com a finalidade de calcular a *media aritmética* e o *desvio padrão* de um conjunto de dados.
- Observe: o retorno destes dados não poder ser via instrução `return()` pois isto não é permitido.
- A solução é criar (na função `main()`, por exemplo) duas variáveis para armazenar os valores desejados (`med` e `desvio`, por exemplo) e então passar os endereços destas variáveis para a função `stat()`.
- A função recebe esses endereços e os armazena em ponteiros (`pm` e `pd`, por exemplo).
- Em seguida, faz os cálculos necessários, armazenando-os nos endereços recebidos.
- Ao término da execução da função os valores de `med` e `desvio` serão atualizados automaticamente

Ponteiro como argumento de função

34

- Observe que nos exemplos acima, a passagem de endereços foi feita através do *operador endereço* (&).
- Também é possível passar um endereço através de um ponteiro já que o conteúdo de um ponteiro é um endereço.

▣ **Exemplo:** Observe o trecho de programa abaixo.

...

```
float *p, x;
```

```
p = &x;
```

```
função(p); // passagem do ponteiro com o endereço de x.
```

Ponteiros e vetores

35

- **Vetores** são intimamente relacionados a **ponteiros**.
- Em C, o **nome** de um vetor é tratado como o **endereço** de seu primeiro elemento.
- Assim ao se passar o nome de um vetor para uma função está se passando o endereço do primeiro elemento de um conjunto de endereços de memória.
- Exemplo:
 - se `vet` é um vetor, então `vet` e `&vet[0]` representam o mesmo **endereço**

Ponteiros e Strings

36

- **Sintaxe:** As duas maneiras mais comuns de declararmos uma *string* são:
 `char nome[tam];`
 ou
 `char *nome;`
- onde:
 - ▣ *nome* é o nome do vetor de caracteres e
 - ▣ *tam* seu tamanho.
- Observe que sendo um vetor, uma *string* pode ser declarada também como um ponteiro.
- Alias a segunda declaração representa justamente isto. Sabendo isto podemos realizar uma grande variedade de manipulações com *strings* e caracteres.
- Existe uma biblioteca padrão C chamada `string.h` que providencia algumas funções de manipulação de *strings* muito úteis.

Ponteiros e Strings

37

- Sendo um ponteiro para caracter **char *texto;**, podemos atribuir uma **constante** string para texto, que não é uma cópia de caracteres, somente ponteiros são envolvidos.
- Neste caso a string é armazenada como parte da função em que aparecem, ou seja, como constante.
 - Char *texto="composto"; /* funciona como **static char texto[]="composto";**
*/

Exercício

38

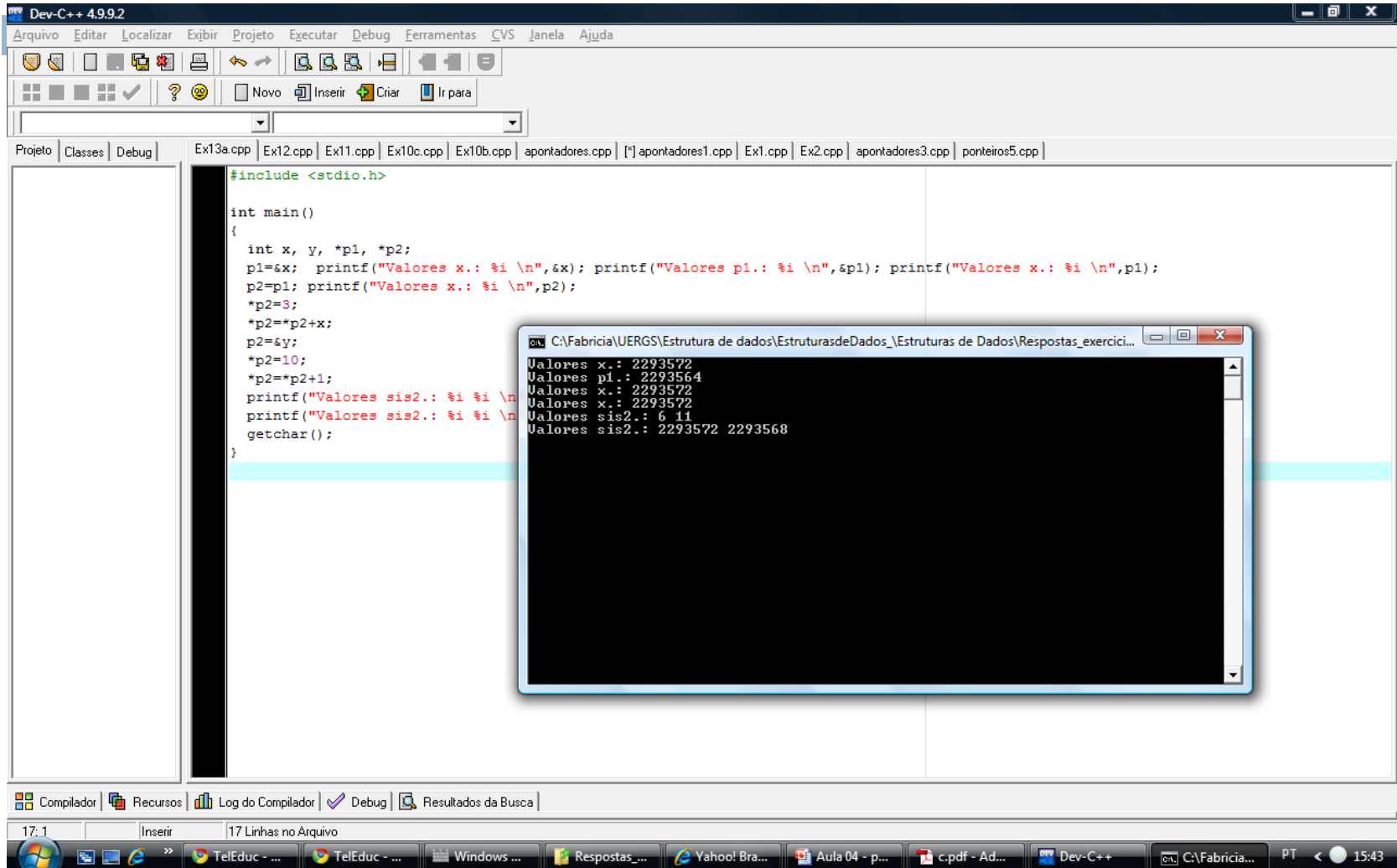
1. Descubra quais os valor das variáveis após a execução de cada trecho de código a seguir.

```
int x, y, *p1, *p2;  
p1=&x;  
p2=p1;  
*p2=3;  
*p2=*p2+x;  
p2=&y;  
*p2=10;  
*p2=*p2+1;
```

```
int a, b, c, *p1, *p2;  
p1=&c;  
*p1=2;  
p2=p1;  
*p1=*p2+3;  
p2=&a;  
a=3;  
a=*p1+*p2;  
b=*p2;  
b++;  
p2=&b;  
*p2=*p2+1;
```

Exercício 3a

39



Exercício 3b

40

```
int a, b, c, *p1, *p2;
```

```
p1=&c;
```

```
*p1=2;
```

```
p2=p1;
```

```
*p1=*p2+3;
```

```
p2=&a;
```

```
a=3;
```

```
a=*p1+*p2;
```

```
b=*p2;
```

```
b++;
```

```
p2=&b;
```

```
*p2=*p2+1;
```

- Imprima TUDO ao longo do código.

Resultado para discussão

C:\Fabricia\UERGS\Estrutura de da

```
Valores p1.: 2293564
Valores &p1.: 2293560
Valores c.: 2
Valores &c.: 2293564
Valores *p1.: 2
Valores &p1.: 2293560
Valores p2.: 2293564
Valores *p1.: 5
Valores p2.: 2293572
Valores &p2.: 2293556
Valores a.: 3
Valores a.: 8
Valores b.: 8
Valores p2.: 2293568
Valores *p2.: 9
Valores a.: 8
Valores b.: 10
Valores p2.: 2293568
Valores *p2.: 10
```