LINGUAGEM DE PROGRAMAÇÃO II

PONTEIROS - CONTINUAÇÃO

Aula 6

Outras Operações com Ponteiros

- As próximas operações, também muito usadas, são o incremento e o decremento.

- Considere um ponteiro **pt** que aponte para uma variável de um *tipo_base* válido em C. Vamos supor que os endereços seguintes ao apontado por **pt** armazenem variáveis do mesmo *tipo_base* apontado por **pt**.

- Quando incrementamos o ponteiro **pt** ele passa a apontar para o próximo valor do *tipo_base*. Isto é, se o ponteiro **pt** aponta para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro.

- Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **char*** ele anda 1 byte na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória.

Outras Operações com Ponteiros

- O decremento funciona de forma semelhante. Supondo que **pt** é um ponteiro, as operações de decremento e incremento são escritas como:

```
pt++;
pt--;
```

- É preciso ficar bem claro que estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **pt**, faz-se:

```
(*pt)++;
```

Outras Operações com Ponteiros

- A linguagem C permite adicionar ou subtrair qualquer valor dos ponteiros. Vamos supor que você deseje apontar para o oitavo elemento do *tipo_base* a partir do elemento apontado por **pt**. Basta fazer:

```
pt=pt+8;
```

- E se você desejar acessar o conteúdo do ponteiro 8 posições adiante:

```
*(pt+8);
```

- A subtração funciona da mesma maneira.

Exercícios

- 1. Explique a diferença entre
  - y = p+1;
  - y = p++;
  - y = (∗p)++;
  - y = ∗(p++);

- e entre:
  - y = p-1;
  - y = p--;
  - y = (*p)--;
  - y = (*p--);

- 2. O que quer dizer ∗(p+10) ? E *(p-5)?

Exercícios

- 3. Qual o valor de **y** no final do programa?
-

```
int main()
{
    int y, *p, x;
    y = 0;
    p = &y;
    x = *p;
    x = 4;
    (*p)++;
    x--;
    (*p) = (*p) + x;
    printf ("y = %d\n", y);
}
```

Passagem de Argumentos

- Em geral, nas linguagens de programação, é possível passar argumentos para as funções de duas formas:
  - por valor: esse método copia o valor de um argumento para o parâmetro formal da função.
  - por referência: esse método copia o endereço de um argumento para o parâmetro formal da função.

- Na chamada por valor, as alterações que você efetuar nos parâmetros da função não terão efeito sobre as variáveis utilizadas para chamar a função.

- Na chamada por referência as alterações feitas no parâmetro afetarão o valor da variável utilizada para chamar a função.

Passagem de Argumentos

- Na linguagem C somente existe chamada por valor.
- Isto é bom quando queremos usar os parâmetros formais a vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função.
- Mas pode ser inconveniente algumas vezes, porque podemos querer que alterações realizadas nos parâmetros continuem valendo fora da função.

Passagem de Argumentos

```
float quadrado(float num) {
    num=num*num;
    return num;
}

void main () {
    float num,sq;
    printf ("Entre com um numero: ");
    scanf ("%f",&num);
    sq=quadrado(num);
    printf ("\n\nO numero original e: %f\n",num);
    printf ("O seu quadrado vale: %f\n",sq);
}
```

Passagem de Argumentos

- No exemplo anterior, o parâmetro formal num da função quadrado() sofre alterações dentro da função, mas a variável num da função main() permanece inalterada.

- Podemos utilizar ponteiros para obter o mesmo efeito da chamada por referência, que o C não possui originalmente.

- Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*.

Passagem de Argumentos

- Os ponteiros são a "referência" que precisamos para poder alterar o conteúdo da variável fora da função.

- O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um **&** na frente das variáveis que estivermos passando para a função.

- Na verdade você já viu isso, lembre-se da função **scanf()**. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela.

Passagem de Argumentos

```
void swap (int *a,int *b){
     int temp;
     temp=*a;
     *a=*b;
     *b=temp;
}

void main (void){
     int num1,num2;
     num1=100;
     num2=200;
     swap (&num1,&num2);
     printf ("\n\nEles agora valem %d  %d\n",num1,num2);
}
```

Passagem de Argumentos

- A função swap() recebe o endereço das variáveis num1 e num2.
- Estes endereços são copiados nos ponteiros a e b.
- Sabemos que através do operador * é possível acessar o conteúdo apontado pelos ponteiros a e b e modificá-los.
- Estes conteúdos são os valores das variáveis num1 e num2, que, portanto, estão sendo modificados.

Exercícios

2. Escreva um programa com uma função Incremento() e Decremento() as quais, respectivamente, aumenta de 1 o valor de uma variável e diminui de 1 o valor de uma variável.

3. Escreva uma função que receba duas variáveis inteiras e "zere" o valor das variáveis.

4. Escreva um programa para calcular as raízes de uma equação do segundo grau, onde uma função raizes() deverá receber o valor do delta e utilizar passagem por referência para guardar em duas variáveis x1 e x2 o valor da raízes da equação.

Ponteiros e Vetores

- Quando declaramos um vetor na forma já descrita, ou seja, fazendo:

```
tipo nome_do_vetor [tamanho];
```

- O compilador C calcula o espaço, em bytes, necessário para armazenar este vetor. Este espaço = tamanho x tamanho_do_tipo

- O compilador então aloca este número de bytes em um espaço livre de memória.

Ponteiros e Vetores

- O *nome_do_vetor* que você declarou é na verdade um ponteiro para o tipo do vetor.

- Este conceito é fundamental! Quando o compilador aloca na memória o espaço para o vetor, ele toma o nome do vetor (que é um ponteiro) e aponta para o *primeiro* elemento do vetor.

- A variável *nome_do_vetor* armazenará o endereço do primeiro elemento do vetor. Em outras palavras o ponteiro *nome_do_vetor* contém o endereço &nome_do_vetor[0], que indica onde na memória está guardado o valor do primeiro elemento do vetor.

Ponteiros e Vetores

- Então:
  - nome_do_vetor[0] é equivalente a *(nome_do_vetor)
  - nome_do_vetor[1] é equivalente a *(nome_do_vetor+1)
  - nome_do_vetor[2] é equivalente a *(nome_do_vetor+2)

- Ou seja, a notação *nome_do_vetor[índice]* é equivalente a **(nome_do_vetor+índice).*

- A notação *&nome_do_vetor[índice]* é válida e retorna o endereço do elemento do vetor indexado por índice. Isto seria equivalente a usar *nome_do_vetor + indice*.

Ponteiros e Vetores

```
void main(){
   int i, *pont, numeros[10];

   for(i=0;i<10;i++)
        numero[i]  = i;

   printf("\nEscrevendo  conteúdo de números[0]: %d",numeros[0]);
   printf("\nEscrevendo  conteúdo de números[0]: %d",*numeros);
   printf("\nEscrevendo  conteúdo de números[5]: %d",numeros[5]);
   printf("\nEscrevendo  conteúdo de números[5]: %d",*(numeros+5));
   pont = numeros;   //pont aponta para o vetor apontado por numeros
   printf("\nEscrevendo  conteúdo de números[9]: %d",pont[9]);
   printf("\nEscrevendo  conteúdo de números[9]: %d",numeros[9]);
}
```

Vetor como Argumento de uma Função

- Quando um vetor (ou matriz) é usado como argumento para uma função, somente o endereço do vetor é passado para a função e não o vetor inteiro, uma vez que, em C, um nome de um vetor corresponde a um ponteiro para o primeiro elemento do mesmo.

- Existem duas maneiras de se declarar um parâmetro formal que receberá como argumento o endereço de um vetor.

Vetor como Argumento de uma Função

- Na primeira forma, o parâmetro pode ser declarado como um vetor do mesmo tipo e tamanho daquele usada como argumento na chamada da função:

```
void display(int num[10]){
  int i;
  for(i=0;i<10;i++)
     printf("%d ",num[i]);
}

main(void) {
  int t[10],i;
  for(i=0;i<10;++i)
     t[i]=i;
  display(t);
}
```

Vetor como Argumento de uma Função

- A segunda forma é declará-lo como um ponteiro para o tipo do vetor:

```
void display(int *num) {
   int i;
   for(i=0; i<10; i++)
      printf("d ", num[i]);
}
```

- Esta forma de declaração é permitida já que qualquer ponteiro pode ser indexado usando-se num[i] como se fosse um vetor, o que equivale a usar *(num + i).

- OBS:
  - devemos ter o cuidado de acessar apenas o espaço de memória alocado para o vetor!

Vetor como Argumento de uma Função

- Quando um vetor é usado como argumento de função, passamos seu endereço. Isso significa que o código na função estará operando sobre o conteúdo atual do vetor e seu conteúdo poderá ser alterado:

```
void passa_para_maiuscula(char *string) {
   int t;
   for(t=0; string[t]; t++)
      string[t] = toupper(string[t]);
}

main(void) {
   char s[80];
   printf("informe uma string: ");
   gets(s);
   passa_para_maiuscula(s);
   printf("\na string original é alterada: %s", s);
}
```

Exercícios

1. Implemente uma função que converte uma string para minúsculas.
2. Implemente uma função que inverte uma string s1 guardando a string invertida em uma outra string s2.
3. Implemente uma função que inverte uma string s1 guardando a string invertida na mesma string s1. Ou seja, apagando o conteúdo de s1 e escrevendo por cima s1 invertida.
4. Implemente as funções abaixo, que devem ter que deve ter o mesmo comportamento das funções originais da biblioteca string.h:
   a) strcat2()
   b) strlen2()
   c) strcmp2()
5. Implemente uma função para verificar se o conteúdo de dois vetores de inteiros de mesmo tamanho é igual. Se for a função retornará 1 e se não for, retornará zero.