AULA 11

PONTEIROS E FUNÇÕES

Nesta aula revemos ponteiros e funções na linguagem C. Até o momento, aprendemos algumas "regras" de como construir funções que têm parâmetros de entrada e saída ou argumentos passados por referência. Esses argumentos/parâmetros, como veremos daqui por diante, são na verdade ponteiros. Um endereço de uma variável é passado como argumento para uma função. O parâmetro correspondente que recebe o endereço é então um ponteiro. Qualquer alteração realizada no conteúdo do parâmetro tem reflexos externos à função, no argumento correspondente. Esta aula é baseada especialmente na referência [7].

11.1 Parâmetros de entrada e saída?

A abstração que fizemos antes para compreendermos o que são parâmetros de entrada e saída na linguagem C será revista agora e revelará algumas novidades. Veja o programa 11.1.

Programa 11.1: Exemplo de parâmetros de entrada e saída e ponteiros.

```
#include <stdio.h>

/* Recebe dois valores e devolve os mesmos valores com conteúdos trocados */
void troca(int *a, int *b)
{
   int aux;

   aux = *a;
   *a = *b;
   *b = aux;
}

/* Recebe dois valores e troca seus conteúdos */
int main(void)
{
   int x, y;

   scanf("%d%d", &x, &y);
   printf("Antes da troca : x = %d e y = %d\n", x, y);
   troca(&x, &y);
   printf("Depois da troca: x = %d e y = %d\n", x, y);

   return 0;
}
```

Agora que entendemos os conceitos básicos que envolvem os ponteiros, podemos olhar o programa 11.1 e compreender o que está acontecendo, especialmente no que se refere ao uso de ponteiros como argumentos de funções. Suponha que alguém está executando esse programa. A execução inicia na primeira linha da função `main` e seu efeito é ilustrado na figura 11.1.

Figura 11.1: Execução da linha 11.

Em seguida, suponha que o(a) usuário(a) do programa informe dois valores quaisquer do tipo inteiro como, por exemplo, 3 e 8. Isso se reflete na memória como na figura 11.2.

Figura 11.2: Execução da linha 13.

Na linha seguinte da função `main`, a execução da função `printf` permite que o(a) usuário(a) verifique na saída os conteúdos das variáveis que acabou de informar. Na próxima linha do programa, a função `troca` é chamada com os endereços das variáveis `x` e `y` como argumentos. Isto é, esses endereços são copiados nos argumentos correspondentes que compõem a interface da função. O fluxo de execução do programa é então desviado para o trecho de código da função `troca`. Os parâmetros da função `troca` são dois ponteiros para valores do tipo inteiro com identificadores `a` e `b`. Esses dois parâmetros recebem os endereços das variáveis `x` e `y` da função `main`, respectivamente, como se pode observar na figura 11.3.

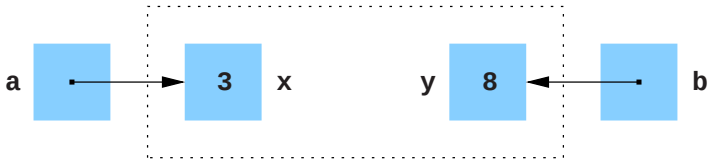

Figura 11.3: Execução da linha 15.

A seguir, na primeira linha do corpo da função `troca`, ocorre a declaração da variável `aux` e um espaço identificado por `aux` é reservado na memória principal, como podemos ver na figura 11.4.

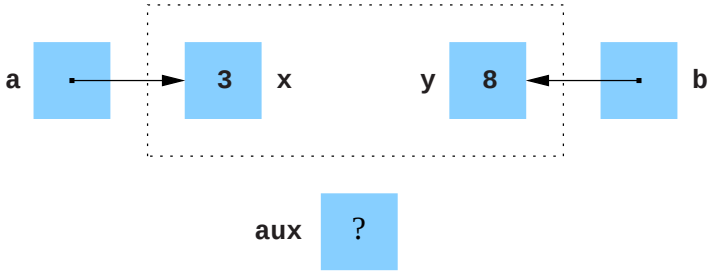

Figura 11.4: Execução da linha 4.

Depois da declaração da variável `aux`, a execução faz com que o *conteúdo da variável apontada por* `a` seja armazenado na variável `aux`. Veja a figura 11.5.

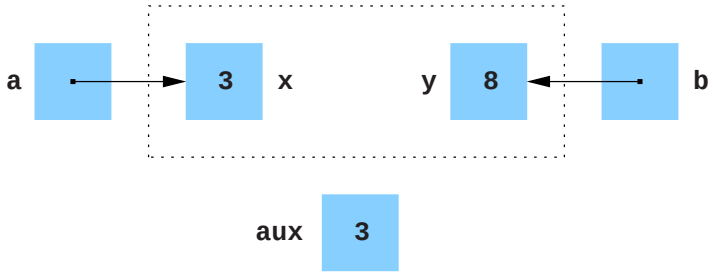

Figura 11.5: Execução da linha 5.

Na execução da linha seguinte, o *conteúdo da variável apontada por* `a` recebe o *conteúdo da variável apontada por* `b`, como mostra a figura 11.6.

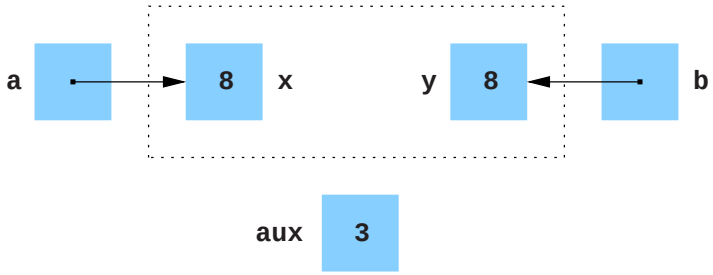

Figura 11.6: Execução da linha 6.

Por fim, na execução da próxima linha, o *conteúdo da variável apontada por* `b` recebe o conteúdo da variável `aux`, conforme a figura 11.7.

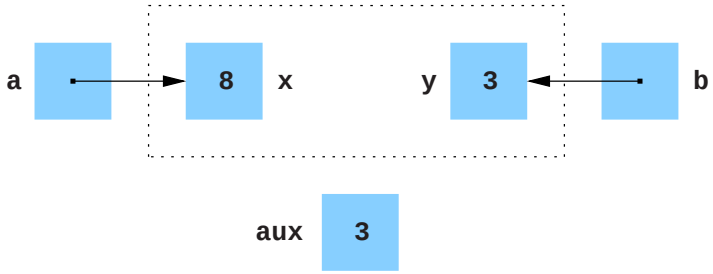

Figura 11.7: Execução da linha 7.

Após o término da função `troca`, seus parâmetros e variáveis locais são destruídos e o fluxo de execução volta para a função `main`, no ponto logo após onde foi feita a chamada da função `troca`. A memória neste momento encontra-se no estado ilustrado na figura 11.8.

Figura 11.8: Estado da memória após o término da função `troca`.

Na linha seguinte da função `main` é a chamada da função `printf`, que mostra os conteúdo das variáveis `x` e `y`, que foram trocados, conforme já constatado e ilustrado na figura 11.8. O programa então salta para a próxima linha e chega ao fim de sua execução.

Esse exemplo destaca que são realizadas cópias do valores dos argumentos – que nesse caso são endereços das variáveis da função `main` – para os parâmetros respectivos da função `troca`. No corpo dessa função, sempre que usamos o operador de indireção para acessar algum valor, estamos na verdade acessando o conteúdo da variável correspondente dentro da função `main`, que chamou a função `troca`. Isso ocorre com as variáveis `x` e `y` da função `main`, quando copiamos seus endereços nos parâmetros `a` e `b` da função `troca`.

Cópia? Como assim cópia? Nós aprendemos que parâmetros passados desta mesma forma são parâmetros de entrada e saída, ou seja, são parâmetros passados por referência e não por cópia. Esse exemplo mostra uma característica muito importante da linguagem C, que ficou dissimulada nas aulas anteriores: *só há passagem de argumentos por cópia na linguagem C*, ou ainda, *não há passagem de argumentos por referência na linguagem C*. O que fazemos de fato é simular a passagem de um argumento por referência usando ponteiros. Assim, passando (por cópia) o endereço de uma variável como argumento para uma função, o parâmetro correspondente deve ser um ponteiro e, mais que isso, um ponteiro para a variável correspondente cujo endereço foi passado como argumento. Dessa forma, qualquer modificação indireta realizada no corpo dessa função usando esse ponteiro será realizada na verdade no conteúdo da variável apontada pelo parâmetro, que é simplesmente o conteúdo da variável passada como argumento na chamada da função.

Não há nada de errado com o que aprendemos nas aulas anteriores sobre argumentos de entrada e saída, isto é, passagem de argumentos por referência. No entanto, vale ressaltar que passagem de argumentos por referência é um tópico conceitual quando falamos da linguagem de programação C. O correto é repetir sempre que *só há passagem de argumentos por cópia na linguagem C*.

11.2 Devolução de ponteiros

Além de passar ponteiros como argumentos para funções também podemos fazê-las devolver ponteiros. Funções como essas são comuns, por exemplo, quando tratamos de cadeias de caracteres conforme veremos na aula 14.

A função abaixo recebe dois ponteiros para números inteiros e devolve um ponteiro para um maior dos números inteiros, isto é, devolve o endereço onde se encontra um maior dos números.

```
/* Recebe dois valores e devolve o endereço de um maior */
int *max(int *a, int *b)
{
   if (*a > *b)
      return a;
   else
      return b;
}
```

Quando chamamos a função `max`, passamos ponteiros para duas variáveis do tipo `int` e armazenamos o resultado em uma variável ponteiro:

```
int i, j, *p;
⋮
p = max(&i, &j);
```

Na execução da função `max`, temos que `*a` é um apelido para `i` e `*b` é um apelido para `j`. Se `i` tem valor maior que `j`, então `max` devolve o endereço de `i`. Caso contrário, `max` devolve o endereço de `j`. Depois da chamada, `p` aponta para `i` ou para `j`.

Não é possível que uma função devolva o endereço de uma variável local sua, já que ao final de sua execução, essa variável será destruída.

Exercícios

11.1 (a) Escreva uma função com a seguinte interface:

```
void min_max(int n, int v[MAX], int *max, int *min)
```

que receba um número inteiro n, com $1 \leqslant n \leqslant 100$, e um vetor v com $n > 0$ números inteiros e devolva um maior e um menor dos elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um menor elemento desse conjunto.
Simule no papel a execução de seu programa antes de implementá-lo.

11.2 (a) Escreva uma função com a seguinte interface:

```
void dois_maiores(int n, int v[MAX], int *p_maior, int *s_maior)
```

que receba um número intero n, com $1 \leqslant n \leqslant 100$, e um vetor v com $n > 0$ números inteiros e devolva um maior e um segundo maior elementos desse vetor.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior e um segundo maior elemento desse conjunto.
Simule no papel a execução de seu programa antes de implementá-lo.

11.3 (a) Escreva uma função com a seguinte interface:

```
void soma_prod(int a, int b, int *soma, int *prod)
```

que receba dois números inteiros a e b e devolva a soma e o produto destes dois números.

(b) Escreva um programa que receba n números inteiros, com $n > 0$ par, calcule a soma e o produto deste conjunto usando a função do item (a) e determine quantos deles são maiores que esta soma e quantos são maiores que o produto. Observe que os números na entrada podem ser negativos.

Simule no papel a execução de seu programa antes de implementá-lo.

11.4 (a) Escreva uma função com a seguinte interface:

```
int *maximo(int n, int v[MAX])
```

que receba um número intero n, com $1 \leqslant n \leqslant 100$, e um vetor v de n números inteiros e devolva o endereço do elemento de v onde reside um maior elemento de v.

(b) Escreva um programa que receba $n > 0$ números inteiros, armazene-os em um vetor e, usando a função do item (a), mostre na saída um maior elemento desse conjunto.

Simule no papel a execução de seu programa antes de implementá-lo.