Escopo de Variáveis

- Variáveis locais
- Parâmetros formais
- Variáveis globais

Já foi dada uma introdução ao escopo de variáveis. O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa.

Variáveis locais

O primeiro tipo de variáveis que veremos são as variáveis locais. Estas são aquelas que só têm validade dentro do bloco no qual são declaradas. Sim. Podemos declarar variáveis dentro de qualquer bloco. Só para lembrar: um bloco começa quando abrimos uma chave e termina quando fechamos a chave. Até agora só tínhamos visto variáveis locais para funções completas. Mas um comando **for** pode ter variáveis locais e que não serão conhecidas fora dali. A declaração de variáveis locais é a primeira coisa que devemos colocar num bloco. A característica que torna as variáveis locais tão importantes é justamente a de serem exclusivas do bloco. Podemos ter quantos blocos quisermos com uma variável local chamada **x**, por exemplo, e elas não apresentarão conflito entre elas.

A palavra reservada do C **auto** serve para dizer que uma variável é local. Mas não precisaremos usá-la pois as variáveis declaradas dentro de um bloco já são consideradas locais.

Abaixo vemos um exemplo de variáveis locais:

```
func1 (...)
{
        int abc,x;
        ...
}
func (...)
{
        int abc;
        ...
}
int main ()
{
        int a,x,y;
        for (...)
        {
                float a,b,c;
        ...
        }
        ...
return 0;
}
```

No programa acima temos três funções. As variáveis locais de cada uma delas não irão interferir com as variáveis locais de outras funções. Assim, a variável **abc** de **func1()** não tem nada a ver (e pode ser tratada independentemente) com a variável **abc** de **func2()**. A variável **x** de **func1()** é também completamente independente da variável **x** de **main()**. As variáveis **a**, **b** e **c** são locais ao bloco **for**. Isto quer dizer que só são conhecidas dentro deste bloco **for** e são desconhecidas no resto da função **main()**. Quando usarmos a variável **a** dentro do bloco **for** estaremos usando a variável **a** local ao **for** e não a variável **a** da função **main()**.

Parâmetros formais

O segundo tipo de variável que veremos são os parâmetros formais. Estes são declarados como sendo as entradas de uma função. Não há motivo para se preocupar com o escopo deles. É fácil: o parâmetro formal é uma variável local da função. Você pode também alterar o valor de um parâmetro formal, pois esta alteração não terá efeito na variável que foi passada à função. Isto tem sentido, pois quando o C passa parâmetros para uma função, são passadas apenas cópias das variáveis. Isto é, os parâmetros formais existem independentemente das variáveis que foram passadas para a função. Eles tomam apenas uma cópia dos valores passados para a função.

Variáveis globais

Variáveis globais são declaradas, como já sabemos, fora de todas as funções do programa. Elas são conhecidas e podem ser alteradas por todas as funções do programa. Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local. Vamos ver um exemplo:

```
int z,k;
func1 (...)
{
        int x,y;
        ...
}

func2 (...)
{
        int x,y,z;
        ...
        z=10;
        ...
}

int main ()
{
        int count;
...
return 0;
}
```

No exemplo acima as variáveis **z** e **k** são globais. Veja que **func2()** tem uma variável local chamada **z**. Quando temos então, em **func2()**, o comando **z=10** quem recebe o valor de 10 é a variável *local*, não afetando o valor da variável global **z**.

Evite *ao máximo* o uso de variáveis globais. Elas ocupam memória o tempo todo (as locais só ocupam memória enquanto estão sendo usadas) e tornam o programa mais difícil de ser entendido e menos geral.

Passagem de parâmetros por valor e passagem por referência

Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros. Veja o exemplo abaixo:

```
#include <stdio.h>
float sqr (float num);
int main ()
{
```

```
        float num,sq;
        printf ("Entre com um numero: ");
        scanf ("%f",&num);
        sq=sqr(num);
        printf ("\n\nO numero original e: %f\n",num);
        printf ("O seu quadrado vale: %f\n",sq);
return 0;
}

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

No exemplo acima o parâmetro formal **num** da função **sqr()** sofre alterações dentro da função, mas a variável **num** da função **main()** permanece inalterada: é uma chamada por valor.

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recurso que permite ao programador fazer chamadas por referência. Há entretanto, no C, um recurso de programação que podemos usar para simular uma chamada por referência.

Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterar a 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. Veja um exemplo:

```
#include <stdio.h>
void Swap (int *a,int *b);
int main ()
{
        int num1,num2;
        num1=100;
        num2=200;
        Swap (&num1,&num2);
        printf ("\n\nEles agora valem %d  %d\n",num1,num2);
return 0;
}
void Swap (int *a,int *b)
{
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
}
```

Não é muito difícil. O que está acontecendo é que passamos para a função Swap o endereço das variáveis num1 e num2. Estes endereços são copiados nos ponteiros a e b. Através do operador * estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Mas, quem é este conteúdo? Nada mais que os valores armazenados em num1 e num2, que, portanto, estão sendo modificados!

Espere um momento... será que nós já não vimos esta estória de chamar uma função com as variáveis precedidas de **&**? Já! É assim que nós chamamos a função **scanf()**. Mas porquê? Vamos pensar um pouco. A função **scanf()** usa chamada por referência porque ela precisa alterar as variáveis que passamos para ela! Não é para isto mesmo que ela é feita? Ela lê variáveis para nós e portanto precisa alterar seus valores. Por isto passamos para a função o endereço da variável a ser modificada!

Vetores como Argumentos de Funções

Quando vamos passar um vetor como argumento de uma função, podemos declarar a função de três maneiras equivalentes. Seja o vetor:

```
int matrx [50];
```

e que queiramos passá-la como argumento de uma função **func()**. Podemos declarar **func()** das três maneiras seguintes:

```
void func (int matrx[50]);
void func (int matrx[]);
void func (int *matrx);
```

Nos três casos, teremos dentro de **func()** um **int*** chamado **matrx**. Ao passarmos um vetor para uma função, na realidade estamos passando um ponteiro. Neste ponteiro é armazenado o endereço do primeiro elemento do vetor. Isto significa que não é feita uma cópia, elemento a elemento do vetor. Isto faz com que possamos alterar o valor dos elementos do vetor dentro da função.

Os Argumentos argc e argv

A função **main()** pode ter parâmetros formais. Mas o programador não pode escolher quais serão eles. A declaração mais completa que se pode ter para a função **main()** é:

```
int main (int argc,char *argv[]);
```

Os parâmetros **argc** e **argv** dão ao programador acesso à linha de comando com a qual o programa foi chamado.

O **argc** (argument count) é um inteiro e possui o número de argumentos com os quais a função **main()** foi chamada na linha de comando. Ele é, no mínimo 1, pois o nome do programa é contado como sendo o primeiro argumento.

O **argv** (argument values) é um ponteiro para uma matriz de strings. Cada string desta matriz é um dos parâmetros da linha de comando. O **argv[0]** sempre aponta para o nome do programa (que, como já foi dito, é considerado o primeiro argumento). É para saber quantos elementos temos em **argv** que temos **argc**.

Exemplo: Escreva um programa que faça uso dos parâamentros *argv* e *argc*. O programa deverá receber da linha de comando o dia, mês e ano correntes, e imprimir a data em formato apropriado. Veja o exemplo, supondo que o executável se chame data:

```
data 19 04 99
```

O programa deverá imprimir:

```
19 de abril de 1999
```

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
int mes;
char *nomemes [] = {"Janeiro", "Fevereiro", "Março", "Abril", "Maio",
                    "Junho", "Julho", "Agosto", "Setembro", "Outubro",
                    "Novembro", "Dezembro"};

if(argc == 4) /* Testa se o numero de parametros fornecidos esta' correto
                 o primeiro parametro e' o nome do programa, o segundo o dia
                 o terceiro o mes e o quarto os dois ultimos algarismos do ano */
{
    mes = atoi(argv[2]);  /* argv contem strings. A string referente ao mes deve ser
                             transformada em um numero inteiro. A funcao atoi esta
                             sendo usada para isto: recebe a string e transforma no
                             inteiro equivalente */

    if (mes<1 || mes>12)  /* Testa se o mes e' valido */
        printf("Erro!\nUso: data dia mes ano, todos inteiros");
    else
      printf("\n%s de %s de 19%s", argv[1], nomemes[mes-1], argv[3]);
}

else printf("Erro!\nUso: data dia mes ano, todos inteiros");

return 0;
```

EXERCÍCIOS DE PASSAGEM POR REFERÊNCIA

1) Faça um programa que receba um número inteiro do usuário pelo teclado. Em seguida, o programa deverá passar esse número por referência para uma função. Na função, o programa retornará o resto da divisão desse número por 10 e alterará o número pelo resultado da divisão dele mesmo por 10. Ao final mostre o número e o valor retornado.

2) Fazer um programa para receber uma frase do usuário lida caracter por caracter (no máximo 30 caracteres) utilizando a função getch() e quando for pressionado a tecla Enter (‘\r’) o programa pára de ler a frase. Ao final mostrar a frase digitada, utilizando subrotina com passagem por referência e ponteiro.

TESTES

1- Qual a afirmativa verdadeira?

- **a.** Você pode retornar para um programa quantas variáveis de uma função desejar através do comando return
- **b.** Uma função só pode ter um comando return
- **c.** Os protótipos de função servem para declarar as funções, isto é, indicar para o compilador qual o seu nome, tipo de retorno e o número e tipos dos parâmetros
- **d.** Uma função não pode retornar um ponteiro
- **e.** Nenhuma das opções anteriores

2- : Qual das seguintes razões não é uma razão válida para o uso de funções em C?

- **a.** Funções usam menos memória do que se repetirmos o mesmo código várias vezes
- **b.** Funções rodam mais rápido
- **c.** Funções fornecem um meio de esconder cálculos em uma "caixa preta" que pode ser usada sem a preocupação de detalhes internos de implementação
- **d.** Funções mantêm variáveis protegidas das outras partes do programa

3- Qual a afirmativa falsa?

- **a.** Se uma função não retorna nada ela deve ser declarada como void
- **b.** O retorno da função main é feito para o sistema operacional
- **c.** stdio.h e string.h contêm o protótipo de algumas funções da biblioteca do C
- **d.** Funções podem ser definidas dentro de outras funções
- **e.** Uma das opções anteriores é falsa

4- Qual a afirmativa verdadeira?

- **a.** stdio.h e string.h contêm o corpo de algumas funções da biblioteca do C
- **b.** Funções podem ser chamadas por outras funções
- **c.** Em um programa C todas as funções de um programa devem estar em um único arquivo .c
- **d.** Variáveis declaradas em uma função são acessíveis por todas as outras funções
- **e.** Nenhuma das opções anteriores

5- Qual a afirmativa verdadeira?

- **a.** A palavra reservada auto é utilizada para dizer que uma variável é local (automática). Porém, ela pode ser omitida dentro de uma função, pois todas as variáveis são locais por default.
- **b.** Não se pode utilizar variáveis com o mesmo nome em funções diferentes.
- **c.** Os parâmetros recebidos por uma função têm o mesmo endereço das variáveis usadas na chamada à função
- **d.** Quando uma variável local tem o mesmo nome de uma variável global, a variável local se torna inacessível e a variável global é acessível

e. Nenhuma das opções anteriores

6- Qual a afirmativa falsa?

a. Os parâmetros recebidos por uma função armazenam cópias das variáveis usadas na chamada da função
b. Variáveis globais são conhecidas e podem ser alteradas por todas as funções do programa
c. Quando queremos alterar as variáveis que são passadas como parâmetros para uma função, devemos declará-las como ponteiros na função
d. A função scanf necessita receber como parâmetro o endereço da variável de entrada, porque ela precisa alterar esta variável.
e. Uma das opções anteriores é falsa

7- O que imprime o programa abaixo?

```
#include <stdio.h>
void func();
int i = 10;
void main()
{
    int i=20;
    func();
    printf("i= %d ", i);
    {
        int i = 30;
        func();
        printf("i= %d ", i);
    }
}
void func()
{
    printf("i = %d ", i);
}
```

a. i= 20 i= 20 i= 30 i= 30
b. i= 10 i= 20 i=10 i= 30
c. i= 20 i=10 i=10 i=30
d. i= 10 i=10 i=10 i=10
e. Nenhuma das opções anteriores

8- Ao se utilizar um vetor como parâmetro para uma função que informação está sendo passada à função?

a. Uma cópia de todos elementos do vetor
b. Uma cópia do primeiro elemento do vetor
c. O endereço do primeiro elemento do vetor
d. O endereço de todos os elementos do vetor
e. Nenhuma das opções anteriores

9- Sejam par1, par2 e par3 variáveis inteiras. Se chamarmos uma função pela instrução:

```
func(&par1,&par2,&par3);
```

Para que servem &par1, &par2 e &par3 ?

a. São valores inteiros passados para a função

b. Servem para armazenar os endereços da função e das funções que chamaram

c. São os endereços das variáveis da função que chamou. Nestes endereços iremos armazenar os valores a serem modificados pela função

d. Armazenam os endereços das funções de biblioteca usados na função

e. Nenhuma das opções anteriores

10- O que imprime o programa a seguir?

```
#include <stdio.h>
func(int *a, int b)
{
    int temp;
    temp = *a;
    *a = b;
    b = temp;
}
void main()
{
    int a = 10, b = 20;
    func(&a, b);
    printf("a = %d, b = %d", a, b);
}
```

a. a = 10, b = 20

b. a = 20, b = 10

c. a = 10, b = 10

d. a = 20, b = 20

e. Nenhuma das opções anteriores

GABARITO:

1 – C | 2 = B | 3 – D | 4 = B | 5 = A | 6 = E | 7 = B | 8 = C | 9 = C | 10 = D