Revisão - TP

Escopo de variáveis

Def: Escopo de variável é o conjunto de regras que determinam a utilização de uma variável em um programa.

Podemos dividir as variáveis quanto ao escopo em três tipos:

- Variáveis locais:
  - Declaradas dentro do bloco de uma função;
  - Não podem ser usadas ou modificadas por outras funções;
  - Só existem enquanto a função onde foi declarada estiver sendo executada.
  - Declaração:

```
int main(){
   int x;
   ...
}
```

- Variáveis globais:
  - Declaradas fora de todos os blocos de funções;
  - São usáveis e modificáveis em qualquer parte do programa;
  - Existem durante toda a execução do programa.
  - Declaração:

```
int x;
int main(){
   ...
}
```

- Parâmetros formais:
  - São variáveis passadas como parâmetro de uma função.
  - Somente existem na função em que as mesmas foram declaradas;
  - Declaração:

```
int func(int a, int b){
   ...
}
```

Passando um vetor/matriz como argumento para uma função:

Forma 1:

```
//Vetor
void func(int vet[]){//Sempre pegamos a primeira posição do array
   ...
}

int main(){
   int numeros[5] = {1, 2, 3, 4, 5};
   func(numeros);
}

//Matriz
void func(int mat[][3]){ //Deixamos claros ou a quantidade de linhas ou colunas
   ...
}

int main(){
   int numeros[2][3] = {{1, 2, 3}, {4, 5, 6}};
   func(numeros);
}
```

Forma 2:

```
//Vetor
void func(int *vet){ //Sempre pegamos a primeira posição do array
   ...
}

int main(){
   int numeros[5] = {1, 2, 3, 4, 5};
   func(numeros);
}

//Matriz
void func(int *mat){ //Deixamos claros ou a quantidade de linhas ou colunas
   ...
}

int main(){
   int numeros[2][3] = {{1, 2, 3}, {4, 5, 6}};
   func(*numeros);
}
```

Recursão

Def: Chamamos de recursividade o ato de uma função chamar a si mesma, repetindo um determinado comportamento similarmente.

Uma função que realiza esse comportamento é chamada de função recursiva.

A cada chamada recursiva, dividimos o problema maior em um conjunto de problemas menores, empilhando os resultados encontrados para posteriormente desempilhá-los e retornarmos o valor encontrado.

- Condição de parada: Define quando se dará o encerramento das chamadas recursivas;
  - Estrutura:
    if(condição)
- Chamada recursiva: A função chama a si mesma.
  - Estrutura:
    return função(parametro);

Exemplos de funções recursivas:

01) Fatorial

```
int fat(int n){
   if(n==0 || n==1){
      return 1;
   }
   return n*fat(n-1);
}
```

02) Fibonacci

```
int fib(int n){
   if(n==0){
      return 0;
   }
   if(n==1){
      return 1;
   }
   return fib(n-1)+fib(n-2);
}
```

03) Imprimir os números naturais de 0 até N

```
void imprimir(int n){
   if(n==0){
      printf("%d", n);
   }
   else{
      imprimir(n-1);
      printf("%d", n);
   }
}
```

Ponteiro

Def: Ponteiro é uma variável que armazena um endereço de memória.

O ponteiro aponta para a localização exata na memória onde estão os dados armazenados desejados.

Isso permite manipular e acessar eficientemente a memória, sendo útil ao trabalhar com: arrays, alocação dinâmica e passagem de parâmetros por referência

Declaração de um ponteiro:

```
int *p = 0;
```

ou

```
int *p = NULL;
```

obs: Os ponteiros devem ser sempre inicializados ao serem declarados.

Atribuindo o endereço de memória (&) de uma variável a um ponteiro:

Ex 01)

```
int x;
int *p = NULL;
p = &x; //Atribuindo o endereço da variável x após a declaração do ponteiro p
```

Ex 02)

```
int x;

int *p = &x; //Inicializando o ponteiro diretamente com o endereço da variável x
```

Desreferenciação: Caso quisermos o conteúdo contido no endereço referenciado pelo ponteiro, utilizamos a chamada desreferenciação, tratando de dados e valores em si, mas não do endereço

Ex: printf(“Valor contido no endereço %p: %d”, p, *p);

Podemos ainda modificar os valores contidos no endereço referenciado pelo ponteiro, onde a variável que possui tal endereço sofrerá as alterações

Ex: *p = 100;

printf(“Novo valor do numero: %d”, x);

Manipulação de arrays: Podemos manipular os vetores de diversas formas, de acordo com nossas necessidades, por exemplo: quando queremos um elemento do vetor ou sua posição.

Por padrão, ao referenciarmos o endereço de um array, não necessitamos de utilizar o &, pois já conseguimos a primeira posição do vetor em questão

Ex: int numeros[5] = {10, 20, 30, 40, 50}
int *p;

p = numeros; //Estamos conseguindo a 1° posição do array números

Caso quisermos imprimir o elemento da posição atual de um array, temos:

```
for(int i=0; i<5; i++){
   printf(“A[%d] = %d\n”, i, *p); //Imprimindo a posição e elemento atual
   printf(“&A[%d] = %p\n”, i, p); //Imprimindo a posição e endereço atual
   p++; //Avançando para a próxima posição do array
}
```

É possível também, preenchermos um vetor com strings (sequências de caracteres), utilizando a referência de endereço do ponteiro:

Char *diasSemana[7] = {“Domingo”, “Segunda”, “Terca”, “Quarta”, “Quinta”, “Sexta”, “Sabado”};

Exercícios

01) Escreva um programa que contenha duas variáveis inteiras. Leia essas variáveis do teclado. Em seguida, compare seus endereços e exiba o conteúdo do maior endereço.

```c
int main()
{
    int x, y;
    scanf("%d%d", &x, &y);
    if(&x>&y)
    {
        printf("%d\n", x);
    }
    else
    {
        printf("%d\n", y);
    }
    return 0;
}
```

02) Crie um programa que contenha um array de float com 10 elementos. Imprima o endereço de cada posição desse array.

```c
int main()
{
    float x[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    for(int i=0; i<10; i++)
    {
        printf("&x[%d] = %p\n", i, &x[i]);
    }
    return 0;
}
```

03) Crie um programa que contenha uma matriz de float com três linhas e três colunas. Imprima o endereço de cada posição dessa matriz.

```c
int main()
{
    float x[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            printf("&x[%d][%d] = %p\n", i, j, &x[i][j]);
        }
    }

}
```

04) Crie um programa que contenha um array com cinco elementos inteiros. Leia esse array do teclado e imprima o endereço das posições contendo valores pares.

```c
int main()
{
    int x[5];
    for(int i=0; i<5; i++)
    {
        scanf("%d", &x[i]);
    }
    for(int i=0; i<5; i++)
    {
        if(x[i]%2==0)
        {
            printf("&x[%d] = %p\n", i, &x[i]);
        }
    }
    return 0;
}
```

05) Crie uma função que receba como parâmetro um vetor de números inteiros e o imprima. Não utilize índices para percorrer o vetor, apenas noções utilizando de ponteiros.

```c
//Usando laço de repetição
void imprimeVetor(int *vet)
{
    for(int i=0; i<5; i++)
    {
        printf("x[%d] = %d\n", i, *vet);
        vet++;
    }
}

int main()
{
    int x[5] = {1, 2, 3, 4, 5};
    imprimeVetor(x);
    return 0;
}
```

```c
//Usando recursão
int imprimeVetor(int *vet, int i)
{
    if(*vet>5)
    {
        return 0;
    }
    printf("&x[%d] = %d\n", i, *vet);
    i++;
    return imprimeVetor(vet+1, i);
}

int main()
{
    int x[5] = {1, 2, 3, 4, 5}, a, i = 0;
    a = imprimeVetor(x, i);
    return 0;
}
```

06) Crie uma função que receba como parâmetro uma matriz de números inteiros e a imprima. Não utilize índices para percorrer o vetor, apenas noções utilizando de ponteiros.

```c
//Usando laço de repetição
void imprimeMatriz(int *mat)
{
    for(int i=0; i<2; i++)
    {
        for(int j=0; j<2; j++)
        {
            printf("x[%d][%d] = %d\n", i, j, *mat);
            mat++;
        }
    }
}

int main()
{
    int x[2][2] = {{1, 2}, {3, 4}};
    imprimeMatriz(*x);
    return 0;
}
```

```c
//Usando recursão
int imprimeMatriz(int *mat, int tot)
{
    if(tot==0)
    {
        return 0;
    }
    printf("%d\n", *mat);
    return imprimeMatriz(mat+1, tot-1);
}

int main()
{
    int x[2][2] = {{1, 2}, {3, 4}}, a, tot_elem = 4;
    a = imprimeMatriz(*x, tot_elem);
    return 0;
}
```