Técnicas de Programação

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
```

Obs: É uma boa prática na programação sempre verificar se o ponteiro foi de fato alocado corretamente, para isso fazemos a seguinte verificação:

```
if(p==NULL)  {

   //Então ele está vazio

   printf("Erro\n");
   exit(1); //Fecha todos os arquivos abertos

}
```

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"};

Alocação dinâmica

Def: Alocação dinâmica é o processo de reservar memória de acordo com nossa necessidade durante a execução de um programa.

Ela permite uma maior flexibilidade ao tratar dados cujo tamanho não é constante.

A alocação é feita por meio de ponteiros de variáveis que desejamos alocar dinamicamente.

Existem diversos comandos para alocar dinamicamente, os principais são:

- malloc (Memory Allocation):
  int *ptr = (int *)malloc(50*sizeof(int)); //Alocando um vetor com, inicialmente, 50 posições do tipo inteiro

- calloc (Contiguous Allocation):

  int *ptr = (int *)calloc(10, sizeof(int)); // Alocando e inicializando um vetor de 10 inteiros com 0 em cada posição.

- realloc (Reallocation):

  int *ptr = (int *)realloc(ptr, 20 * sizeof(int)); // Expandindo os espaços de uma variável para 20 espaços para números inteiros

- free (Liberar Memória):

  free(ptr); // Libera a memória apontada por ptr

Registros/Struct

Def: Registros, structs ou variáveis compostas heterogêneas é uma estrutura que engloba diferentes tipos de dados, conhecidos como atributos, em uma única entidade/objeto.

É como se tivéssemos criando um novo tipo de variável que contém diferentes tipos de dados possíveis para ela.

Pode-se criar um registro dentro ou fora da função principal, mas por convenção criamos fora por se tratar de uma entidade global, que todas as funções do programa devem ter acesso.

Vejamos um exemplo prático de como criar:

Maneira 1 - struct

```
struct Dados{
   char nome[100], sexo;
   int idade;
   float salario;
};


int main() {
   //Declarando uma variável do tipo pessoa
   struct Dados pessoa1;
   struct Dados pessoas[50]; //Podemos ainda definir um vetor do tipo Dados, para 50 pessoas por exemplo


//Acessando um campo de pessoa1 e pessoas
   pessoa1.idade = 32;
   pessoas[0].salario = 7500.0;


//Lendo e imprimindo um dado lido
   scanf(“ %c %c”, &pessoa1.sexo, &pessoas[0].sexo);
   printf(“ %c %c”, pessoa1.sexo, pessoas[0].sexo);
```

```
    return 0;
}
```

Maneira 2 – typedef struct

```
typedef struct {
    char nome[100], sexo;
    int idade;
    float salario;
} Dados;

int main() {
    //Declarando uma variável do tipo pessoa
    Dados pessoa1;
    Dados pessoas[50]; //Podemos ainda definir um vetor do tipo Dados, para 50 pessoas por exemplo

    //Acessando um campo de pessoa1 e pessoas
    pessoa1.idade = 32;
    pessoas[0].salario = 7500.0;

    //Lendo e imprimindo um dado lido
    scanf(" %c %c", &pessoa1.sexo, &pessoas[0].sexo);
    printf(" %c %c", pessoa1.sexo, pessoas[0].sexo);

    return 0;
}
```

Para ler e imprimir um atributo específico de uma struct, utilizamos a mesma lógica do scanf e printf

Arquivos

Def: Arquivos são utilizados para armazenar dados de diferentes formas, geralmente permanentemente.

Ao utilizar um arquivo é necessário que este seja aberto e depois de seu uso o mesmo seja fechado.

Para abrir um arquivo, utilizamos a estrutura FILE declarando um ponteiro para ele, juntamente da função fopen()

```
FILE *p = fopen("nomeDoArquivo.extensao", "modo de abertura");
```

O primeiro parâmetro da função fopen é o nome do arquivo e sua extensão, por exemplo: clientes.txt ou code.dat

Em seguida, temos um modo de abertura, isso define como o arquivo será aberto. Em C, temos os seguintes modos de abertura:

- "r" (Read):

  FILE *p = fopen("clientes.txt", "r"); //Abrindo um arquivo no modo de leitura

- "w" (Write):

  FILE *p = fopen("clientes.txt", "w"); //Abrindo um arquivo no modo de escrita (sobrescreve sobre o que havia no arquivo)

- "a" (Append):

  FILE *p = fopen("clientes.txt", "r"); //Abrindo um arquivo no modo de anexo (escreve em seguida do último conteúdo do arquivo)

- "r+" ou "w+" ou "a+" (Modo de leitura e escrita):

  FILE *p = fopen("clientes.txt", "r+"); //Abrindo um arquivo no modo de leitura e escrita

Como um arquivo é aberto em um ponteiro do tipo FILE, também é importante verificar se ele foi aberto adequadamente, visto como uma boa prática na programação.

```
if(p==NULL) {
   printf("Erro ao abrir o arquivo!\n");
   exit(1); //Fecha todos os arquivos abertos
}
```

Os principais comandos para gerenciar um arquivo são:

- **ABRIR**: FILE *arq1 = fopen("arq1.dat", "r")
  Parâmetros: Nome do arquivo com extensão e modo de abertura
  Obs: Nome do arquivo pode conter o caminho absoluto ou relativo do mesmo
  Caso o arquivo não exista e esteja no modo de abertura "w", o mesmo será criado na mesma pasta em que o programa está inserido

- **FECHAR**: fclose(arq1)
  Parâmetros: Ponteiro do arquivo
  Obs: Todo arquivo aberto deve ser fechado

- **ABORTAR**: exit(1)
  Parâmetros: Código de retorno
  Obs: Todos os arquivos abertos durante a execução do programa são imediatamente fechados.

- **LEITURA**:
  - fgetc(arq1)
    Parâmetros: Ponteiro do arquivo para pegar o caractere da linha atual dele
  - fgets(str, sizeof(str), arq1)
    Parâmetros: Vetor de char que armazenará a string da linha atual do arquivo, tamanho do vetor de char e ponteiro do arquivo para pegar a string

- **ESCRITA/IMPRESSÃO**:
  - fprintf(arq1, "%s", str)
    Parâmetros: Ponteiro do arquivo que será colocado os dados, parâmetros dos tipos de dados e variável que contém os dados desejados
  - fputc(c, arq1)
    Parâmetros: Caractere a ser escrito e ponteiro do arquivo
  - fputs(str, arq1)
    Parâmetros: String a ser escrita e ponteiro do arquivo

- **ESCRITA/IMPRESSÃO**: fputc(int c, arq1)
  Parâmetros: Caractere a ser escrito e ponteiro do arquivo
  Obs: Todo arquivo aberto deve ser fechado

- **FIM DO ARQUIVO**: feof(arq1)
  Parâmetros: Ponteiro do arquivo a verificar seu final
  Obs: Valor diferente de 0 caso chegou no final

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;
}
```