**Lista de Exercícios 1 – Ponteiros**

Aluno: Ricardo Martins Venâncio Varela - (Matricula: 20160108323)

**QUESTÃO 1 -** A partir do uso de ponteiros, podemos localizar o endereço de memória de uma determinada variável, dessa forma podemos acessar uma variável em diferentes partes de um programa, sendo úteis em várias situações: Alocação dinâmica de memória, manipulação de Arrays, entre outras utilidades.

**QUESTÃO 2 -**

int i=3,j=5;

int \*p, \*q;

p = &i; // O valor de p é o endereço de i

q = &j; // O valor de q é o endereço de j

p == &i; // Verdadeiro, p recebe o endereço de i.

Valor da expressão = 6356740.

\*p - \*q; // \*p igual ao conteúdo em p e \*q igual ao conteúdo em q.

Valor da expressão = -2.

\*\*&p; // O ponteiro p para um ponteiro que apontará para um inteiro, o conteúdo do endereço de p é o endereço de i.

Valor da expressão = 3.

3 - \*p/(\*q) + 7; // Valor da expressão = 10.

**QUESTÃO 3 –**

int i=5, \*p;

p = &i;

printf("%x %d %d %d %d", p,\*p+2,\*\*&p,3\*\*p,\*\*&p+4);

}

Supondo que i ocupa o endereço 4094 na memória, obtive os seguintes resultados:

p = 4096;

\*p+2 = 7;

\*\*&p = 5;

3\*\*p= 15;

\*\*p&p+4 = 9;

**QUESTÃO 4 -**

p = i; // Atribuição legal, p recebe o valor de i.

q = &j; // Atribuição legal, q recebe o endereço de j.

p = &\*&i; // Atribuição legal.

i = (\*&)j; //Atribuição ilegal, devido ao uso do parênteses

i = \*&j; // Atribuição legal.

i = \*&\*&j; Atribuição legal, é executado.

q = \*p; // Atribuição ilegal.

i = (\*p)++ + \*q; Atribuição legal, sabendo que p e j recebem o valor de i e j.

**QUESTÃO 5 -**

int main() {

int valor;

int \*p1;

float temp;

float \*p2;

char aux;

char \*nome = "Ponteiros";

char \*p3;

int idade;

int vetor[3];

int \*p4;

int \*p5;

// A questão pede para determinar os valores para os seguintes problemas abaixo:

/\* (a) \*/

 valor = 10;

 p1 = &valor;

 \*p1 = 20;

 printf("%d \n", valor);

// 🡪 Valor encontrado = 20

 /\* (b) \*/

 temp = 26.5;

 p2 = &temp;

 \*p2 = 29.0;

 printf("%.1f \n", temp);

// 🡪 temp = 29.0

/\* (c) \*/

 p3 = &nome[0];

 aux = \*p3;

 printf("%c \n", aux);

// 🡪 aux = P

 /\* (d) \*/

 p3 = &nome[4];

 aux = \*p3;

 printf("%c \n", aux);

// 🡪 aux = e

 /\* (e) \*/

 p3 = nome;

 printf("%c \n", \*p3);

// 🡪 \*p = P

 /\* (f) \*/

 p3 = p3 + 4;

 printf("%c \n", \*p3);

// 🡪 p3 = e

 /\* (g) \*/

 p3--;

 printf("%c \n", \*p3);

// 🡪 p3 = t;

 /\* (h) \*/

 vetor[0] = 31;

 vetor[1] = 45;

 vetor[2] = 27;

 p4 = vetor;

 idade = \*p4;

 printf("%d \n", idade);

// 🡪 idade = 31

 /\* (i) \*/

 p5 = p4 + 1;

 idade = \*p5;

 printf("%d \n", idade);

// 🡪 idade = 45

 /\* (j) \*/

 p4 = p5 + 1;

 idade = \*p4;

 printf("%d \n", idade);

// 🡪 idade = 27

 /\* (l) \*/

 p4 = p4 - 2;

 idade = \*p4;

 printf("%d \n", idade);

// 🡪 idade =31

 /\* (m) \*/

 p5 = &vetor[2] - 1;

 printf("%d \n", \*p5);

// 🡪\*p5 =45

 /\* (n) \*/

 p5++;

 printf("%d \n", \*p5);

 return(0);

// 🡪 \*p5 =27

return 0;

}

**QUESTÃO 6 -** Fazendo a compilação do programa, foi encontrado:

contador/valor/valor/endereco/endereco

i = 0vet[0] = 1.1\*(f + 0) = 1.1&vet[0] = 60FEF4(f + 0) = 60FEF4

i = 1vet[1] = 2.2\*(f + 1) = 2.2&vet[1] = 60FEF8(f + 1) = 60FEF8

i = 2vet[2] = 3.3\*(f + 2) = 3.3&vet[2] = 60FEFC(f + 2) = 60FEFC

i = 3vet[3] = 4.4\*(f + 3) = 4.4&vet[3] = 60FF00(f + 3) = 60FF00

i = 4vet[4] = 5.5\*(f + 4) = 5.5&vet[4] = 60FF04(f + 4) = 60FF04

**QUESTÃO 7 -** Fazendo os testes no compilador, a seguinte expressão faz referência ao valor do terceiro elemento do vetor: \*(pulo + 2);

Pois, “pulo” está apontando para o primeiro endereço, acrescentando o +2, irá para o terceiro elemento do vetor, já fora dos parênteses o \* vai mostrar o conteúdo da terceira posição.

**QUESTÃO 8 -**

p = mat + 1; // Essa expressão é válida. Pois ela mostra o endereço de memória do vetor da posição seguinte (+1).

p = mat++; // Essa expressão não é válida. Pois o comando é de incremento, essa operação não é permitida para o que queremos fazer.

p = ++mat; // Essa expressão não é válida. Segue o mesmo raciocínio da anterior, não é possível realocar.

x = (\*mat)++; Essa expressão é válida. Pois, pleo fator de \* ser usado , “x” recebe o conteúdo da variável mais um (+1).

**QUESTÃO 9 -**

No primeiro caso, como \* foi usado na printf, ou seja, durante a implementação do conteúdo, seu uso serviu para mostrar o valor das variáveis de cada posição do vetor.

No segundo caso, ao retirar \*, foi exibido para o usuário o endereço das variáveis de cada posição do vetor.

**QUESTÃO 10 -**

**Char:** 4092 +1; 4092 + 2; 4092 +3;

**Int:** 4092 + 2; 4092 + 4; 4092 + 6;

**Float:** 4092 + 4; 4092 + 8; 4092 + 12;

**Double:** 4092 + 8; 4092 + 16; 4092 + 24;

**QUESTÃO 11-**

Na prática, fizemos os testes utilizando: Char, int, float e double. Ao compilar o programa, foi obtido os seguintes resultados:

Char: 1 Byte 🡪 6356747 ; 6356748 ; 6356749 ;

Int: 4 Bytes 🡪 6356732; 6356736; 6356740;

Float: 4 Bytes 🡪 6356740; 6356744; 6356748;

Double: 8 Bytes 🡪 6356712; 6356720 ; 6356728;

**QUESTÃO 12 -**

float aloha[10], coisas[10][5], \*pf, value = 2.2;

int i=3;

// Identificar quais dos seguintes comandos é válido ou inválido:

aloha[2] = value; // Válido, pois recebe 2.2;

scanf("%f", &aloha); // Válido, o conteúdo vai para o endereço de aloha[0].

aloha = value"; Inválidos, pois aloha é um ponteiro.

printf("%f", aloha); Inválido, pois retorna o primeiro elemento do vetor que não pode ser do tipo float

coisas[4][4] = aloha[3]; Válido.

coisas[5] = aloha; Inválido, pois é uma matriz e não um vetor.

pf = value; Inválido, pois “pf” é um ponteiro de float.

pf = aloha; Válido.

**QUESTÃO 13 -**

Com base nos conhecimentos adquiridos por meio da pesquisa ao tema, ponteiros para funções trabalha passando uma função como parâmetro para outra função, ou seja, o programador pode ter acesso ao determinado endereço de uma função.

Exemplificando melhor com um simples programa, temos:

|  |  |  |
| --- | --- | --- |
|  |  | #include <stdio.h> |
|  |  | #include<stdlib.h> |
|  |  | #include<math.h> |
|  |  |  |
|  |  | #define pi 3.14 |
|  |  |  |
|  |  | int divisao (int a, int b); |
|  |  | int (\*divi)(int,int) = divisao; |
|  |  | int operacao (int x, int y, int (\*func)(int, int)); |
|  |  |  |
|  |  | int main(void){ |
|  |  | int result\_1=0; |
|  |  | result\_1 = operacao(1, 1, divi); |
|  |  | printf("Valor da operacao = %d", result\_1); |
|  |  | return 0; |
|  |  | } |
|  |  | int divisao(int a, int b){ |
|  |  | return(a/b); |
|  |  | } |
|  |  | int operacao(int x, int y, int (\*func)(int, int)){ |
|  |  | int op=0, angulo=0; |
|  |  | int valor =0; |
|  |  | valor = 180/pi; |
|  |  | op = func(x, y); |
|  |  | angulo = atan(op) \* valor; |
|  |  | return(angulo); |
|  |  | } |

**QUESTÃO 14 -**

Utilizando os conhecimentos a respeito de Malloc para realizar a tarefa, temos:

#include <stdio.h>  
#include <stdlib.h>

int main(void){  
  float \*ptr;  
  int n, i, j, aux;  
  n = 4;  
  ptr = (float\*) malloc(n\*sizeof(float));  
  
  for(j=0;j<n;j++){  
      scanf("%f",&ptr[j]);  
  }  
  
  for(i=0; i<n; i++){  
      for (j=i+1;j<n;j++){  
          if(ptr[i]>ptr[j]){  
          aux =ptr[i];  
          ptr[i]= ptr[j];  
          ptr[j]=aux;  
          }  
      }  
  }  
  for(j=0;j<n;j++){  
      if(j==n-1){  
      printf("%1f ",ptr[j]);  
      }  
      else {  
          printf("%1f, ",ptr[j]);  
      }  
  }  
free(ptr);

return 0;  
}

**QUESTÃO 15 -**

Com base nos estudos a respeito da função qsort(), a sua utilização visa ordenar qualquer vetor em qualquer ordem escolhida.

* qsort(vetor,quantidade,tamanho,função);

Fazendo a implementação dessa função no programa anterior, temos:

#include <stdio.h>  
#include <stdlib.h>  
  
int comparador (const void \*a, const void \*b){

return (\*(int\*)a - \*(int\*)b);

// Nesse passo ocorre a conversão do valor dos parâmetros para inteiro e adentra o conteúdo da variável.  Se for positivo, “a” é maior que “b”, então realiza a troca, já se for negativo, “a” é menor que “b”, faz as trocas necessárias para ordenar.

}  
  
int main(void){  
  float \*ptr;  
  int n, i, j, aux;  
  n = 4;  
  ptr = (float\*) malloc(n\*sizeof(float));  
  
  for(j=0;j<n;j++){  
      scanf("%f",&ptr[j]);  
  }

qsort(ptr, n, sizeof(float), comparador);

// “ptr” é o endereço do Array que será ordenado, já “n” é o tamanho do Array (quantidade), “sizeof(float)” é o tamanho, dada em bytes, de cada posição do Array, “comparador” é o ponteiro para a função de comparação que recebe dois parâmetros.

for(i=0;j<n;j++){  
      printf("%f",ptr[i]);  
  }  
  
  free(ptr);  
  return 0;  
}

**QUESTÃO 16 -**

#include <stdio.h>

#include <stdlib.h>

int comparador (const void \*a, const void \*b) {

   return ( \*(int\*)a - \*(int\*)b );

}

int comparador2(const void \*a, const void \*b) {

if (\*(int\*)a > \*(int\*)b) {

      return 1;

  } else if (\*(int\*)a < \*(int\*)b) {

      return -1;

   } else {

      return 0;

   }

}

int main (void) {

   int i, array[] = { 6, 12, 3, 9, 24, 42 };

   qsort(array, 6, sizeof(int), comparador);

   for( i = 0 ; i < 6; i++ ) {

      printf("%i ", array[i]);

   }

   return(0);

}

**QUESTÃO 17 -**

Alguns dos mecanismos que possibilitam medir tempos de execução de rotinas computacionais, pode ser pela biblioteca lib time.h, calculando quanto tempo demorou para o determinado programa executar com base no horário que iniciou a compilação até o horário do término da compilação. Para isso podemos usar dois operadores que possibilitam executar a medição, o time(NULL) e o clock().

Fazendo os testes no programas anteriores, foi usado:

float tempo;

time\_t time\_ini, time\_f;

time\_ini = time(NULL);

time\_f = time(NULL);

tempo = difftime(time\_f, time\_ini);

Com isso foram obtidos resultados entre 0.7s e 2.0s no primeiro programa, de acordo com o término da execução do programa e já no segundo caso, algo em torno de 0.1s

**QUESTÃO 18 -**

#include <stdio.h>

#include <stdlib.h>

void soma(int \*a, int \*b, int\*c, int tamanho);

int main(void){

int \*x, \*y,\*vet;

int tam=4;

int i=0,j=0;

  x = (int\*) malloc(tam\*sizeof(int));

  y = (int\*) malloc(tam\*sizeof(int));

  vet = (int\*) malloc(tam\*sizeof(int));

       for(j=0;j<tam;j++){

          printf("Digite o valor de x[%d]\n",j);

          scanf("%d",&x[j]);

        }

       for(i=0;i<tam;i++){

          printf("Digite o valor de y[%d]\n",i);

          scanf("%d",&y[i]);

        }

         soma(x,y,vet,tam);

return 0;

free(x);

free(y);

free(vet);

}

void soma(int \*a, int \*b, int \*c, int tamanho){

  int j=0, i=0;

  for(j=0; j<tamanho; j++){

       \*(c+j)=\*(a+j) + \*(b+j);

  }

  for(i=0;i<tamanho;i++){

      printf("Valor de vet[%d] e: %d ",i,\*(c+i))

 }

}

**QUESTÃO 20 -**

O libGC (Garbage Collector) que implementa um coletor de lixo C, tem como finalidade substituir o malloc. Resumidamente, o que ela faz é a alocação de memória, sem desalocar a memória que não é mais útil.

Por meio de algumas pesquisas sobre a biblioteca libGC e sua construção, um exemplo abaixo de demonstração:

#include "gc.h" // Biblioteca GC (Garbage Collector)

#include <assert.h>

#include <stdio.h>

int main ()

{

int i;

GC\_INIT

para (i = 0; i <10000000; ++ i)

{

int \*\* p = (int \*\*) GC\_MALLOC (tamanho de (int \*));

int \* q = (int \*) GC\_MALLOC\_ATOMIC (tamanho de (int));

assert (\* p == 0);

\* p = (int \*) GC\_REALLOC (q, 2 \* tamanho de (int));

if (i% 100000 == 0)

printf ("Tamanho do heap =% d \ n", GC\_get\_heap\_size ());

}

return 0;

}