RAFAEL VIEIRA COELHO

IFRS

PONTEIROS

PROGRAMAÇÃO 2

CAPÍTULO 22: PONTEIROS

https://books.goalkicker.com/CBook/

PONTEIROS

Um ponteiro é declarado como qualquer outra variável, mas para indicar que se trata de um ponteiro, usa-se o *****.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid object yet */
```

Durante o seu uso, se usarmos ***** antes, estamos obtendo o valor para onde ele está apontando.

```
int value = 1;
pointer = &value;
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

O símbolo **&** é usado para obter o endereço de memória que é armazenado no ponteiro.

PONTEIROS X STRUCTS

Usamos **->** para acessar o membro de um struct que está armazenado em um ponteiro.

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

Podemos inicializar um ponteiro com valor nulo.

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

Para testar se o ponteiro é nulo (não aponta para endereço válido):

```
if (!pointer)
```

PONTEIROS

Quando alteramos através do ponteiro o conteúdo do endereço armazenado, o valor da variável original é alterado.

```c
int *pointer;
int value = 1;
pointer = &value;
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", value);
/* Value of pointed to variable after change: 2 */
```

PONTEIROS X VETORES

O ponteiro quando recebe uma variável que armazena um vetor, ele aponta para o endereço de memória da primeira posição do vetor.

```c
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;
/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

Quando queremos passar um vetor por parâmetro, devemos passá-lo como um ponteiro.

```c
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

PERCORRENDO VETOR COM PONTEIROS

```c
#include <stdio.h>
#define SIZE (10)
int main()
{
    size_t i = 0;
    int *p = NULL;
    int a[SIZE];

    /* Setting up the values to be i*i */
    for(i = 0; i < SIZE; ++i)
    {
        a[i] = i * i;
    }

    /* Reading the values using pointers */
    for(p = a; p < a + SIZE; ++p)
    {
        printf("%d\n", *p);
    }

    return 0;
}
```

```
0
1
4
9
16
25
36
49
64
81
```

PASSAGEM DE PARÂMETRO POR VALOR

Os parâmetros em linguagem C são passados por valor por padrão.

Isto significa que alterações na variável v dentro da função modify() não reflete em alterações na variável v da função main().

```c
void modify(int v) {
    printf("modify 1: %d\n", v); /* 0 is printed */
    v = 42;
    printf("modify 2: %d\n", v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(v);
    printf("main 2: %d\n", v); /* 0 is printed, not 42 */
    return 0;
}
```

PASSAGEM DE PARÂMETRO POR REFERÊNCIA

Podemos passar o endereço da variável v (&) e armazená-lo em um ponteiro.

Neste exemplo, quando voltamos para a função main() após chamar o método modify(), o valor de v é realmente alterado.

```c
void modify(int* v) {
    printf("modify 1: %d\n", *v); /* 0 is printed */
    *v = 42;
    printf("modify 2: %d\n", *v); /* 42 is printed */
}

int main(void) {
    int v = 0;
    printf("main 1: %d\n", v); /* 0 is printed */
    modify(&v);
    printf("main 2: %d\n", v); /* 42 is printed */
    return 0;
}
```

RETORNANDO MÚLTIPLOS VALORES

Precisamos usar ponteiros para conseguir retirar mais de um valor.

```c
void Get( int* c , double* d )
{
    *c = 72;
    *d = 175.0;
}

int main(void)
{
    int a = 0;
    double b = 0.0;

    Get( &a , &b );

    printf("a: %d, b: %f\n", a , b );

    return 0;
}
```

PASSANDO MATRIZ POR PARÂMETRO

Podemos usar ** para indicar um ponteiro para outro ponteiro.

```c
#define ROWS 3
#define COLS 2

void fun1(int **, int, int);

int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int **a, int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

PASSANDO MATRIZ POR PARÂMETRO

Mas a melhor solução é usar (*)[]

```c
#define ROWS 3
#define COLS 2

void fun1(int (*)[COLS], int, int);
int main()
{
  int array_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };
  int n = ROWS;
  int m = COLS;

  fun1(array_2D, n, m);

  return EXIT_SUCCESS;
}

void fun1(int (*a)[COLS], int n, int m)
{
  int i, j;
  for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
      printf("array[%d][%d]=%d\n", i, j, a[i][j]);
    }
  }
}
```

PASSANDO VETORES MULTIDIMENSIONAIS POR PARÂMETRO

Podemos declarar um vetor multidimensional de duas formas: **vetor de vetores** ou vetor de ponteiros.

```c
#include <assert.h>
#include <stdlib.h>

/* When passing a multidimensional array (i.e. an array of arrays) to a
   function, it decays into a pointer to the first element as usual.  But only
   the top level decays, so what is passed is a pointer to an array of some fixed
   size (4 in this case). */
void f(int x[][4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}

/* This prototype is equivalent to f(int x[][4]).
   The parentheses around *x are required because [index] has a higher
   precedence than *expr, thus int *x[4] would normally be equivalent to int *(x[4]),
   i.e. an array of 4 pointers to int.  But if it's declared as a
   function parameter, it decays into a pointer and becomes int **x,
   which is not compatable with x[2][4]. */
void g(int (*x)[4]) {
    assert(sizeof(*x) == sizeof(int) * 4);
}
```

PASSANDO VETORES MULTIDIMENSIONAIS POR PARÂMETRO

Podemos declarar um vetor multidimensional de duas formas: vetor de vetores ou **vetor de ponteiros.**

```c
/* An array of pointers may be passed to this, since it'll decay into a pointer
   to pointer, but an array of arrays may not. */
void h(int **x) {
    assert(sizeof(*x) == sizeof(int *));
}

int main(void) {
    int foo[2][4];
    f(foo);
    g(foo);
    /* Here we're dynamically creating an array of pointers.  Note that the
       size of each dimension is not part of the datatype, and so the type
       system just treats it as a pointer to pointer, not a pointer to array
       or array of arrays. */
    int **bar = malloc(sizeof(*bar) * 2);
    for (size_t i = 0; i < 2; i++) {
        bar[i] = malloc(sizeof(*bar[i]) * 4);
    }
    h(bar);
    for (size_t i = 0; i < 2; i++) {
        free(bar[i]);
    }
    free(bar);
}
```

Declarando estaticamente

Alocando dinamicamente

Precisamos liberar O espaço alocado dinamicamente

PONTEIRO PARA UMA FUNÇÃO

O primeiro operando deve ser um ponteiro para uma função, identificando qual função chamar e quais seus parâmetros.

```c
int myFunction(int x, int y)
{
    return x * 2 + y;
}

int (*fn)(int, int) = &myFunction;
int x = 42;
int y = 123;

printf("(*fn)(%i, %i) = %i\n", x, y, (*fn)(x, y)); /* Outputs "fn(42, 123) = 207". */
printf("fn(%i, %i) = %i\n", x, y, fn(x, y)); /* Another form: you don't need to dereferenc
explicitly */
```

PONTEIROS: ERROS COMUNS

Não verificar se a alocação dinâmica funcionou.

For example, unsafe way:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Safe way:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

PONTEIROS: ERROS COMUNS

Usar números fixos (literais) ao invés de usar sizeof().

Non-portable allocation:

```
int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Portable allocation:

```
int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

Or, better still:

```
int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

PONTEIROS: ERROS COMUNS

Não desalocar espaço alocado de memória (função free()).

Devemos seguir os passos:

1) Alocação de memória (**malloc** ou **calloc**);

2) uso dos dados armazenados;

3) Liberação do espaço de memória(**free**).

PONTEIROS: ERROS COMUNS

Criar ponteiros para armazenar variáveis.

Ex: a variável x deixa de existir após o término da execução da função myFunction().

```c
int* myFunction()
{
    int x = 10;
    return &x;
}
```

```c
int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}
```

PONTEIROS: ERROS COMUNS

Incrementando e decrementando.

Jamais use *p++ e *p--

Use:

(*p)++

(*p)—

PONTEIRO PARA UM PONTEIRO

Ao alterar o valor para 100 com **p, estamos alterando o conteúdo que está sendo apontado por *p1

```c
#include <stdio.h>

int main(void) {
    int a, b, *p1, **p;

    p1 = &b;
    p = &p1;
    p = &p1;
    *p = &a;
    **p = 100;
    printf("&a = %p, a = %d \n", &a, a);
    printf("&b = %p, b = %d \n", &b, b);
    printf("&p1 = %p, p1 = %p, *p1 = %d \n", &p1, (void *)p1, *p1);
    printf("p = %p , **p = %d \n", (void *)p, **p);
    return 0;
}
```

```
&a = 0x7ffeea0d8938, a = 100
&b = 0x7ffeea0d8934, b = 32766
&p1 = 0x7ffeea0d8928, p1 = 0x7ffeea0d8938, *p1 = 100
p = 0x7ffeea0d8928 , **p = 100
```

PONTEIRO PARA UM PONTEIRO

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

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */

  return EXIT_SUCCESS;
}
```

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

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
  int*** pppA = &&&A;  /* Compilation error here! */
```

PONTEIROS X FUNÇÕES

Ponteiros podem ser usados para apontar para o endereço de memória de uma função.

```c
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
int (*my_pointer)(int, int);
my_pointer = &my_function;
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);
/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Declara a função

Define um ponteiro para a função

Atribui a função ao ponteiro

Chama a função

Passando a função como parâmetro

EXEMPLO COMPLETO

```c
#include <stdio.h>

int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);
    printf("Result: %d \n", result);
}

int main(void) {
    int (*my_pointer)(int, int);

    my_pointer = &my_function;

    /* Calling the pointed function */
    int result = (*my_pointer)(4, 2);

    printf("Result: %d \n", result);
    another_function(my_pointer);
    return 0;
}
```

```
Result: 14
Result: 14
```

VOID*

Quando não sabemos o tipo que será recebido, podemos usar void*.

Um exemplo é a função malloc que pode ser usada para alocar espaço de memória para vários tipos.

```
void* malloc(size_t);
int* vector = malloc(10 * sizeof *vector);
```

Como ele é genérico, o ideal é fazer o cast para indicar o tipo.

```
int* vector = (int*)malloc(10 * sizeof int*);
```

OBRIGADO!

RAFAELVC2@GMAIL.COM

https://books.goalkicker.com