

ponteiros de variáveis

- uma função pode retornar um tipo de valor para função que chama.
 - ↳ a possibilidade de retornar um valor não sempre é satisfatória
- muitas vezes precisaremos transferir mais do que um valor para a função que chama.
 - ↳ isso não pode ser feito com o retorno explícito de valores

• Para ilustrar isso, inicialmente, vamos considerar uma função simples que calcula a soma de dois valores inteiros

```
1 #include <stdio.h>
2
3 int soma (int a, int b){
4     int c;
5     c = a + b;
6     return c;
7 }
8
9 int main (void){
10    int s;
11    s = soma (3, 5);
12    printf("Soma = %d\n", s);
13
14    return 0;
15 }
```

↳ esse exemplo não apresenta nenhuma dificuldade, pois o resultado da soma pode ser retornado de forma explícita.

↳ o problema aparece quando definirmos que a função resulte em mais de um valor.

• Como exemplo, vamos considerar agora uma função para calcular a soma e o produto de dois números

```
1 #include <stdio.h>
2
3 void somaprod (int a, int b, int c, int d){
4     c = a + b;
5     d = a * b;
6 }
7
8 int main (void){
9     int s, p;
10    somaprod(3, 5, s, p);
11    printf("Soma = %d produto = %d\n", s, p);
12
13    return 0;
14 }
```

↳ essa seria uma forma INCORRETA de implementar essa função.

↳ s e p não foram inicializados na função main, e seus valores não serão alterados.
↳ vai retornar valores "livre"!

↳ alterados não os valores de c e d dentro da função somaprod mas eles não representam os variáveis da função main e não espelham os valores inicializados com os valores de c e d.

• Para que a função que chama tenha acesso aos dois valores calculados, é preciso entender antes o conceito de ponteiros para variáveis.

variáveis do tipo ponteiro

- a linguagem C permite o armazenamento e manipulação de valores e endereços de memória.
 - para cada tipo existente, há um tipo de ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.
- ↳ os ponteiros permitem a transferência de dados entre funções, permitindo aos programadores armazenar e alterar dados de outras partes do programa.

```
int a;
```

↳ declaramos uma variável de nome a que pode armazenar valores inteiros

↳ automaticamente, reservar-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes)

* assim como declaramos variáveis para armazenar inteiros, podemos declarar variáveis, os quais em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória em que há valores inteiros armazenados

```
int* p;
```

↳ usamos a mesma palavra de tipo com os mesmos das variáveis precedidos por asterisco (*)

↳ declaramos uma variável de nome p que pode armazenar endereços de memória em que existem inteiros armazenados

ENDERECO DE UMA VARIÁVEL: local onde a mesma é armazenada

&

↳ endereço de

- resulta no endereço da posição da memória reservada para a variável.
- é usado para acessar o endereço de memória de uma variável, permitindo que o programador manipule os dados associados a ela.
- não é permitido modificar o endereço de uma variável por meio de uma atribuição.

```
● ● ●
1 int numero = 2; // Cria uma variável inteira 'numero' com valor 2.
2 int *ponteiro; // Declara um ponteiro para inteiro chamado 'ponteiro'.
3
4 ponteiro = &numero; // Faz o ponteiro 'ponteiro' apontar para o endereço de memória da variável 'numero'.
```

*

↳ conteúdo de

- acessa o conteúdo do endereço de memória armazenado pela variável ponteira.
- usando o operador *, podemos acessar o conteúdo da variável referenciada pelo ponteiro, permitindo acesso direto ao conteúdo da variável.

```
● ● ●
1 float var = 3.14;
2 float *pointeiro; // Declara um ponteiro 'pointeiro'.
3 float pi = *pointeiro; // Atribui a 'pi' o valor de 'var' através do ponteiro.
4 *pointeiro = 3.1456; // Altera o valor de 'var' (e consequentemente 'pi') para 3.1456 através do ponteiro.
```

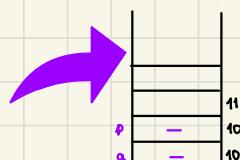
↳ quando alteramos o conteúdo, acontece uma desreferenciação

na prática

```
● ● ●
1 //variável inteira
2 int a;
3
4 //variável p/ ponteiro
5 int *p;
```

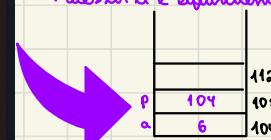
```
● ● ●
1 //a recebe o valor 5
2 a = 5;
3
4 /*p recebe o endereço de a
5 p = &a;
6
7 //conteúdo de p recebe o valor 6
8 *p = 6;
```

↳ as variáveis armazenam valores liso, pois ainda não foram inicializadas



↳ fazendo as atribuições, a variável a receberá indiretamente o valor 6.

↳ acessar a é equivalente a acessar *p, pois p armazena o endereço de a



declarando ponteiros

- os ponteiros são declarados usando o operador '*' seguido do tipo de dados específico

```
1 /*declarando ponteiros:  
2 tipo_apontado * nome_da_variável_ponteiro;*/  
3 int * point;  
4  
5 //imprimindo o valor de um ponteiro:  
6 printf("O ponteiro: %p\n", point);  
7 //usa %p
```

o ponteiro nulo

↳ NULL

- um ponteiro nulo é um ponteiro que não aponta para nenhum objeto
- é usado para indicar ausência de um valor válido.

```
1 #include <stdlib.h> //Inclui a biblioteca padrão para alocação dinâmica de memória.  
2 char * point; //Declara um ponteiro para caractere chamado 'point'.  
3 point = NULL; //Indica que não aponta para nenhum endereço de memória específico.
```

compatibilidade e conversões entre ponteiros

- ponteiros são versáteis, permitindo aos programadores criar algoritmos complexos e modernos
- a conversão entre diferentes tipos de ponteiros pode ser usada para acessar memória e dados de forma segura

```
1 float number; //Declara uma variável float chamada 'number'.  
2 int * pointer; //Declara um ponteiro para inteiro chamado 'pointer'.  
3 pointer = &number; //Faz o ponteiro 'pointer' apontar para o endereço de memória da variável 'number', mas isso pode causar problemas de tipo.  
4 pointer = (int*) &number; //Corrigi o problema de tipo, convertendo o ponteiro para apontar corretamente para o endereço de memória de 'number'.
```

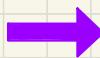
↳ o (int*) força que o número seja do tipo int*

aritmética de ponteiros

- a aritmética de ponteiros permite que os programadores manipulem memória e dados usando operações matemáticas simples
- o cálculo de deslocamento de ponteiros pode ser usado para acessar dados específicos em estruturas de dados complexas.
- considerando que p é um ponteiro:
 - soma de um inteiro a um ponteiro: p+2
 - subtração de um inteiro a um ponteiro: p-3
 - incremento de ponteiro: ++p ou p++
 - decremento de ponteiro: --p ou p--
 - subtração entre dois ponteiros do mesmo tipo: p-p2

```
1 int * pointer;  
2 pointer + 3;  
3 /*representa o endereço da posição de  
4 memória que está três objetos do tipo  
5 int adiante do endereço do objeto para  
6 o qual pointer aponta*/
```

...	...
112	P+3
108	P+2
104	P+1
100	P
96	P-1
92	P-2
88	P-3
...	...



SOBE E DESCE AS POSIÇÕES DA MEMÓRIA

usando funções com ponteiros

- já vimos que as funções não podem alterar diretamente valores de variáveis da função que faz a chamada.
 - mas se passarmos para uma função os valores dos endereços de memória em que suas variáveis estão armazenadas, essa função pode alterar, indiretamente, os valores das variáveis da função que a chamou.
- Numa das maneiras de usar ponteiros na linguagem C é passar ponteiros como argumentos para uma função. Isso permite que a função acesse e modifique o valor de uma variável fora do seu próprio escopo.
- retomando o exemplo de uma função para calcular a soma e o produto de dois números inteiros: a solução para esse problema é fazer com que a função somaprod receba os endereços das variáveis da função main e, assim, alterar os valores indiretamente.

```

● ● ●
1 #include <stdio.h>
2
3 //função chamada somaprod que recebe dois inteiros (a e b) e dois ponteiros para inteiros (p e q).
4 void somaprod(int a, int b, int *p, int *q){
5     *p = a + b;
6     *q = a * b;
7     //calcula a soma de a e b, armazenando o resultado em *p, e o produto de a e b, armazenando o resultado em *q.
8 }
9
10 int main (void){
11     int s, p;
12
13     somaprod(3, 5, &s, &p); //Chama a função somaprod com os argumentos 3, 5, e os endereços de s e p.
14     printf("Soma = %d. Produto = %d\n", s, p); //Imprime na tela a soma e o produto calculados pela função somaprod
15
16     return 0;
17 }
18 }
```

EXEMPLOS DOS SLIDES:

```

● ● ●
1 // função chamada incrementa que recebe um ponteiro para inteiro como argumento.
2 void incrementa (int * ponteiro){
3     (*ponteiro)++; //incrementa o conteúdo do ponteiro
4 }
5
6 int main (){
7     int variavel = 5;
8     incrementa (&variavel); //passa o endereço da variável como argumento
9     printf("Valor da variável: %d", variavel);
10    return 0;
11 }
```

- a função somaprod recebe o endereço de duas variáveis, armazena os valores calculados nesses endereços de memória e altera os valores das variáveis originais

```
1 // função chamada cria_variavel que retorna um ponteiro para inteiro.
2 int *cria_variavel () {
3     int variavel = 5;
4     return &variavel;
5     //cria uma variável inteira local chamada variavel com valor 5 e retorna o endereço dessa variável.
6 }
7
8 int main () {
9     int *ponteiro = cria_variavel(); //declara um ponteiro para inteiro chamado ponteiro e inicialização com o valor retornado pela função cria_variavel.
10    printf("Valor da variável: %d", *ponteiro);
11
12    return 0;
13 }
```

PONTEIROS COMO RETORNO DE FUNÇÃO: permite que uma função retorne um endereço de memória para uma variável.

ponteiros de função

- os ponteiros de função não apontam para uma função específica, permitindo que a função possa ser chamada diretamente
- também permitem que a função seja passada como parâmetro para outras funções
- Para declarar um ponteiro de função, você deve especificar o tipo de retorno da função seguido do nome do ponteiro e o tipo de parâmetros entre parenteses:

```
1 int (*ponteiro)(int, int);
```

Neste exemplo, estou declarando um ponteiro chamado "ponteiro" que é um ponteiro para uma função que tem um tipo de retorno inteiro e dois parâmetros inteiros.

EXEMPLO DOS SLIDES:

```
1 //função chamada soma que recebe dois inteiros como argumentos e retorna a soma deles.
2 int soma (int a, int b){
3     return a + b;
4 }
5
6 // função chamada calcula que recebe dois inteiros (x e y) e um ponteiro para uma função que recebe dois inteiros e retorna um inteiro (operacao).
7 int calcula (int x, int y, int (*operacao)(int, int)){
8     return (operacao)(x,y);
9     //chama a operacao passando os argumentos x e y e retorna o resultado.
10 }
11
12 int main (){
13     int resultado = calcula (5, 3, soma); //Chama a função calcula passando os valores 5, 3 e a função soma como operacao. O resultado é armazenado na variável resultado.
14     printf("Resultado: %d", resultado);
15
16     return 0;
17 }
```