Ponteiros

Em um programa, cada variável tem um **endereço**, que indica sua localização na memória do computador, e um **conteúdo**, que é o valor armazenado. Geralmente os valores armazenados são escalares (inteiros, reais ou caracteres) ou não-escalares (vetores, matrizes e estruturas).

Variáveis do tipo **ponteiro** armazenam **endereços** de memória de outras variáveis; em outras palavras, ponteiros são variáveis que **referenciam** outras variáveis. Ponteiros podem ser bem úteis em determinadas situações, por isso são muito usados na programação em C.

Ponteiro é uma variável cujo conteúdo é um **endereço** na memória.

Declaração

Um ponteiro é declarado através do modificador *, seguindo este modelo:

```
<tipo_base> * <nome_do_ponteiro> ;
```

Exemplo:

```
int   *pi ;  // pi é um ponteiro para inteiros
float *pf ;  // pf é um ponteiro para floats
char  *pc ;  // pc é um ponteiro para caracteres
```

No exemplo acima, `pi` é um ponteiro para inteiros, ou seja, pode referenciar (“conter o endereço de” ou “apontar para”) um local da memória que contém um valor do tipo `int`.

É importante lembrar que o ponteiro também é uma variável: ele tem **tipo**, **tamanho**, ocupa uma posição na memória (ou seja, tem um **endereço** próprio) e armazena um **valor** (que é o endereço de outra variável).

Uso

Ponteiros são usados sobretudo de três formas:

- **Atribuir**: atribuir um valor ao ponteiro
- **Ler**: ler o valor armazenado no ponteiro
- **Desreferenciar**: ler o valor da posição de memória referenciada (“apontada”) pelo ponteiro.

Eis um exemplo mais elaborado dessas operações:

ponteiros.c

```
#include <stdio.h>

int main ()
{
```

```c
  int *p ;                        // ponteiro para inteiros
  int a = 231 ;
  int b = 7680 ;

  printf ("&a vale %p\n", &a) ;   // endereço de a
  printf ("&b vale %p\n", &b) ;   // endereço de b
  printf ("&p vale %p\n", &p) ;   // endereço de p

  printf ("p vale %p\n", p) ;     // valor de p (leitura)

  p = &a ;                        // atribuir valor a p
  printf ("p vale %p\n", p) ;     // ler valor de p
  printf ("*p vale %d\n", *p) ;   // desreferenciar p

  p = &b ;
  printf ("p vale %p\n", p) ;
  printf ("*p vale %d\n", *p) ;

  *p = 500 ;                      // desreferenciar p
  printf ("b vale %d\n", b) ;

  return 0 ;
}
```

A execução do código acima gera a seguinte saída (`(nil)` indica um ponteiro nulo):

```
&a vale 0x7ffe2b852890
&b vale 0x7ffe2b852894
&p vale 0x7ffe2b852898

p vale (nil)

p vale 0x7ffe2b852890
*p vale 231

p vale 0x7ffe2b852894
*p vale 7680

b vale 500
```

A tabela a seguir mostra o conteúdo da memória em diversos momentos da execução do código:

variável	**a**	**b**	**p**
endereço	0x7ffe2b852890	0x7ffe2b852894	0x7ffe2b852898
valor inicial	231	7680	(nil)
valor após p = &a	231	7680	0x7ffe2b852890
valor após p = &b	231	7680	0x7ffe2b852894
valor após *p = 500	231	500	0x7ffe2b852894

Observe que os endereços das variáveis não mudam, apenas seus valores.

Ponteiros nulos

Um ponteiro nulo é aquele que aponta para nada, ou seja nenhum endereço válido. A macro NULL define o valor de ponteiros nulos, que equivalem a zero (0) no C ANSI.

A tentativa de desreferenciar um ponteiro nulo resulta em **erro de acesso à memória**, que geralmente leva à interrupção da execução com uma mensagem de *Segmentation Fault* ou similar. Um exemplo de código contendo esse tipo de erro:

ptr-errado.c

```c
#include <stdio.h>

int main ()
{
  int *p ;

  // na linha abaixo, qual o valor apontado por p?
  printf ("p vale %p e *p vale %d\n", p, *p) ;

  return 0 ;
}
```

A forma correta de tratar esse problema é **testar cada ponteiro antes de usá-lo**, como mostra o exemplo abaixo:

ptr-correto.c

```c
#include <stdio.h>

int main ()
{
  int *p ;

  if (p)  // (equivale a "if (p != NULL)"
    printf ("p vale %p e *p vale %d\n", p, *p) ;

  return 0 ;
}
```

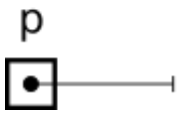
Ponteiros não-inicializados (contendo “lixo”) também podem levar ao mesmo tipo de erro, pois podem apontar para áreas de memória que não estão acessíveis ao programa.

Representação gráfica

As estruturas de dados que usam ponteiros podem ser tornar grandes e complexas, como listas, árvores e grafos. Por isso, muitas vezes é interessante poder representá-las graficamente. Uma variável de tipo ponteiro é usualmente representada como um quadrado com uma seta, que indica para onde a variável aponta.

Na figura a seguir está representado o ponteiro declarado no código abaixo. Também é mostrada a representação de um ponteiro nulo, que não aponta para um conteúdo válido.

```
int a = 231 ;
int *p = &a ;
...
p = NULL ;
```



Ponteiros e vetores

Um vetor pode ser visto como um ponteiro para uma área de memória que contém uma sequência de valores do mesmo tipo. Por exemplo:

```
short int valor[5] ;
```

O nome `valor` é equivalente a `&valor[0]`, ou seja, o endereço do primeiro elemento do vetor.

Dessa forma, o nome de um vetor pode ser visto como um ponteiro para dados do tipo definido no vetor e aponta para o endereço do primeiro elemento do vetor. Em consequência, as seguintes declarações são equivalentes:

```
int *ptr ;   // ponteiro para inteiros
int ptr[] ;  // endereço de um vetor de inteiros (não alocado)
```

Aritmética de ponteiros

Ponteiros são valores numéricos e portanto podem sofrer algumas operações aritméticas simples. Considerando *T* o **tamanho** em bytes do tipo apontado por um ponteiro, temos:

- ++: o valor do ponteiro é incrementado de *T* bytes.
- --: o valor do ponteiro é decrementado de *T* bytes.
- +: somando *n* ao ponteiro, seu valor é incrementado de $n \times T$ bytes.
- -: subtraindo *n* do ponteiro, seu valor é decrementado de $n \times T$ bytes.

Além disso, ponteiros podem ser comparados (<, >, >=, <=, ==, !=, etc).

Exemplo:

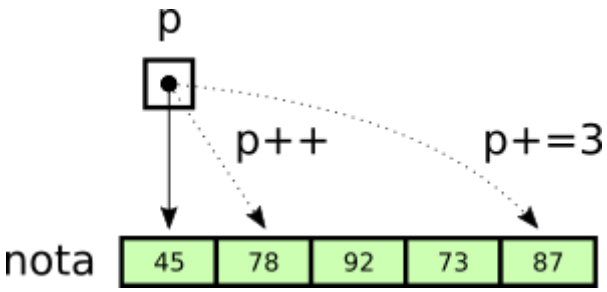
```
int nota[5] = { 45, 78, 92, 73, 87 } ;
int *p ;

p = nota ; // p aponta para nota[0]
printf ("p: %p, *p: %d\n", p, *p) ;

p++ ;      // p aponta para nota[1]
```

```
printf ("p: %p, *p: %d\n", p, *p) ;

p += 3 ;   // p aponta para nota[4]
printf ("p: %p, *p: %d\n", p, *p) ;
```

Resultado da execução:

```
p: 0x7ffd28f319a0, *p: 45
p: 0x7ffd28f319a4, *p: 78
p: 0x7ffd28f319b0, *p: 87
```

A evolução do ponteiro p do código acima pode ser representada graficamente:



Dessa forma, a manipulação de vetores pode ser feita usando aritmética de ponteiros:

```
#define SIZE 100

int v[SIZE] ;
int i, *p ;

// preenchimento usando []
for (i = 0 ; i < SIZE ; i++)
  v[i] = 0 ;

// preenchimento usando aritmética de ponteiros
for (p = v ; p < v + SIZE ; p++)
  *p = 0 ;

// ou ainda...
for (i = 0 ; i < SIZE ; i++)
  *(v + i) = 0 ;
```

Ponteiros para ponteiros

Como visto acima, um ponteiro é uma variável que pode conter o endereço (“apontar”) de outras variáveis. Nada impede um ponteiro de conter o endereço de outro ponteiro, o que chamados de *referencia indireta*, *ponteiro duplo* ou *ponteiro para ponteiro*.

A declaração e uso de ponteiros indiretos é simples:

pontpont.c

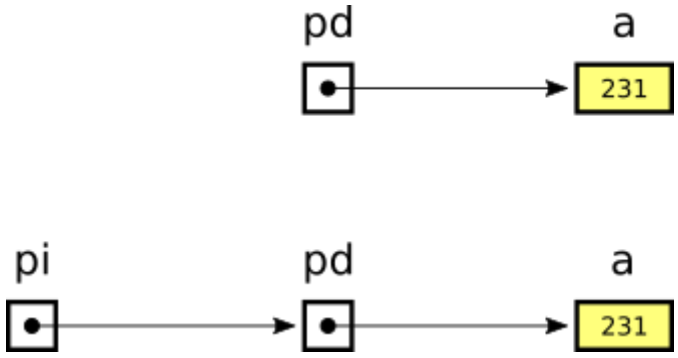
```
#include <stdio.h>

int main ()
```

```c
{
  int a = 231 ;
  int *pd ;  // ponteiro direto
  int **pi ; // ponteiro indireto, equivale a int *(*p)

  pd = &a ;  // pd recebe o endereço de um int
  pi = &pd ; // pi recebe o endereço de um ponteiro para int

  printf ("a  está em %p e vale %d\n", &a, a) ;
  printf ("pd está em %p e vale %p\n", &pd, pd) ;
  printf ("pi está em %p e vale %p\n", &pi, pi) ;

  printf ("*pd  vale %d\n", *pd) ;
  printf ("*pi  vale %p\n", *pi) ;
  printf ("**pi vale %d\n", **pi) ;

  return 0 ;
}
```

O resultado da execução do código acima é:

```
a  está em 0x7ffda4afa4ac e vale 231
pd está em 0x7ffda4afa4b0 e vale 0x7ffda4afa4ac
pi está em 0x7ffda4afa4b8 e vale 0x7ffda4afa4b0
*pd  vale 231
*pi  vale 0x7ffda4afa4ac
**pi vale 231
```

Graficamente, os ponteiros `pd` e `pi` são representados assim:



Ponteiros void

Um ponteiro de tipo `void` é considerado um **ponteiro genérico**, que pode referenciar qualquer endereço de memória independente de seu tipo. Ponteiros `void` são muito usados para trasferir parâmetros genéricos para funções, ou para construir estruturas de dados genéricas, que podem armazenar/referenciar dados de diversos tipos.

```c
void *ptr ;
```

Por não terem um tipo predefinido, ponteiros `void` não podem ser desreferenciados. As operações aritméticas sobre ponteiros `void` consideram como tamanho básico 1 byte.

Exemplo:

void.c

```c
#include <stdio.h>

int main ()
{
  int a = 34 ;
  int b ;
  void *p ;

  p = &a ;
  b = *p ;  // erro de compilação!

  printf ("p vale %p\n", p) ;
  p++ ;
  printf ("p vale %p\n", p) ;

  return (0) ;
}
```

Ponteiros e vetores II

O uso de ponteiros pode facilitar muito a escrita de código envolvendo vetores. Um exemplo clássico é a implementação da função `strcpy (s,t)`, que copia a string `t` para a string `s` (extraído e adaptado do livro *C Programming ANSI* - Kernighan & Ritchie).

Implementação vetorial básica:

```c
void strcpy (char *s, char *t)
{
  int i ;
  i = 0 ;
  while (t[i] != '\0')
  {
    s[i] = t[i] ;
    i++ ;
  }
  s[i] = t[i] ;
}
```

Implementação vetorial melhorada:

```c
void strcpy (char *s, char *t)
{
  int i ;
  i = 0 ;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

Versão usando ponteiros:

```
void strcpy (char *s, char *t)
{
  while ((*s = *t) != '\0')
  {
    s++;
    t++;
  }
}
```

Podemos mudar os operadores de incremento para dentro da condição:

```
void strcpy (char *s, char *t)
{
  while ((*s++ = *t++) != '\0') ;
}
```

Podemos retirar a comparação `!= '\0'`, pois é redundante:

```
void strcpy (char *s, char *t)
{
  while (*s++ = *t++) ;
}
```

Exercícios

1. Assistir o vídeo Binky Pointer Fun, da Stanford University (versão com legendas em português).
2. Escreva um programa que leia 10 inteiros da entrada padrão, armazene-os em um vetor e os escreva na saída padrão na ordem contrária; todos os acessos ao vetor devem ser feitos usando somente ponteiros, sem usar índices (vet[i], etc).
3. Mude o programa anterior para ordenar o vetor usando o algoritmo da bolha.
4. Escreva um programa para calcular o tamanho de uma string usando somente ponteiros.
5. Escreva um programa para concatenar duas strings usando somente ponteiros.