
Algoritmos e Programação II

Apontadores - Parte I

Prof. Viviane Bonadia dos Santos

“Os conceitos de endereço e ponteiro são fundamentais em qualquer linguagem de programação, embora fiquem ocultos em algumas linguagens. Em C, esses conceitos são explícitos.”

- A memória RAM de qualquer computador é uma sequência de **bytes**. Cada byte armazena um de 256 possíveis valores. Os bytes são numerados sequencialmente e o número de um byte é o seu **endereço**.
- Cada objeto armazenado na memória do computador ocupa um certo número de bytes consecutivos. Um `char` ocupa 1 byte. Um `int` ocupa 4 bytes e um `double` ocupa 8 bytes em muitos computadores.

Endereços

- Cada objeto na memória tem um endereço. Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte.

Por exemplo, depois das declarações:

```
int vetor[4] = {1, 2, 3, 4};
```

```
char caractere = 'a';
```

os endereços das variáveis poderiam ser os seguintes:

vetor[0] 89434

vetor[1] 89438

caractere 89421

O operador de endereços &

- O endereço de um objeto é dado pelo operador &.
- Se **i** é uma variável então **&i** é o seu endereço.

Exemplo:

```
int i;  
scanf ("%d", &i);
```

O segundo argumento da função de biblioteca `scanf` é o **endereço** da variável onde deve ser depositado o objeto lido do dispositivo padrão de entrada.

O operador de endereços &

- Para conhecer o endereço ocupado por uma variável basta usar o operador de endereços &.

```
1 #include <stdio.h>
2
3 void main(void){
4     int a = 5;
5
6     printf("%p\n", &a);
7 }
```

```
$ gcc endereco.c -o endereco
$ ./endereco
0x7ffd5ed30324
```

PONTEIROS

“Um **ponteiro** (ou **apontador**) é um tipo especial de variável que armazena o **endereço** (localização) de outra variável.”

“Dizemos que uma variável **aponta** para outra variável quando a primeira contém o endereço da segunda.”

- “Um **ponteiro** (ou **apontador**) é um tipo especial de variável que armazena o **endereço** (localização) de outra variável.”
- “Dizemos que uma variável **aponta** para outra variável quando a primeira contém o endereço da segunda.”

“Um **ponteiro** (ou **apontador**) é um tipo especial de variável que armazena o **endereço** (localização) de outra variável.”

“Dizemos que uma variável **aponta** para outra variável quando a primeira contém o endereço da segunda.”

Exemplo

Vamos supor que temos uma variável **ponteiro** apt e uma variável int n. Como podemos fazer com que a variável apt guarde o endereço de n?

“Um **ponteiro** (ou **apontador**) é um tipo especial de variável que armazena o **endereço** (localização) de outra variável.”

“Dizemos que uma variável **aponta** para outra variável quando a primeira contém o endereço da segunda.”

Exemplo

Vamos supor que temos uma variável **ponteiro** `apt` e uma variável `int` `n`. Como podemos fazer com que a variável `apt` guarde o endereço de `n`?

```
apt = &n;
```

Em termos um pouco mais abstratos, diz-se que `apt` é uma referência à variável `n`.

Declaração de variável tipo ponteiro

- Uma variável do tipo ponteiro deve ser declarada da seguinte forma:

tipo *****nome_variavel;

```
1 void main(){  
2  
3     int *ponteiroInt;  
4     float *ponteiroFloat;  
5     char *ponteiroChar;  
6 }
```

As variáveis `ponteiroInt`, `ponteiroFloat` e `ponteiroChar` são ponteiros para `int`, `float` e `char` respectivamente.

Associando um ponteiro a uma variável

```
int n;  
int *ap1;
```



```
ap1 = &n;
```



O operador indireto *

- Os ponteiros proporcionam um modo de acesso à variável sem referenciá-la diretamente (modo indireto de acesso).
- Uma vez que o ponteiro está apontando para uma variável (ou seja, guardando o seu endereço) **é possível ter acesso ao conteúdo da variável apontada usando o operador ***.

“Se *i* é uma variável e *p* vale `&i` então dizer `*p` é o mesmo que dizer *i*”.

O operador indireto *

```
1 #include <stdio.h>
2
3 void main(){
4     int *p;
5     int i = 5;
6     p = &i;
7     printf("%d ", *p);
8     printf("%d\n", i);
9
10    *p = 10;
11    printf("%d ", *p);
12    printf("%d\n", i);
13 }
```

```
$ gcc ponteiros.c -o ponteiros
```

```
$ ./ponteiros
```

```
5 5
```

```
10 10
```

Uso de variáveis tipo ponteiro

- Não se pode atribuir um valor para o endereço apontado pelo ponteiro, sem antes ter certeza de que o endereço é válido:

```
int a,b;
```

```
int *c;
```

```
b=10;
```

```
*c=13; //Vai armazenar 13 em qual endereço?
```

- O correto seria por exemplo:

```
int a,b;
```

```
int *c;
```

```
b=10;
```

```
c = &a;
```

```
*c=13;
```


Atenção!

“Infelizmente o **operador de ponteiros** é igual a **multiplicação**, portanto preste atenção em como utiliza-lo.”

```
1 #include <stdio.h>
2 void main(void){
3     int b,a;
4     int *c;
5     b= 10;
6     c= &a;
7     *c = 11;
8     a = b * c;
9     printf("\n%d\n",a);
10 }
```

Ocorre um erro de compilação pois o * é interpretado como operador de ponteiro sobre c.

Passagem de parâmetro

- Argumentos passados para funções tem seus valores copiados para as variáveis parâmetros da função. Este processo é idêntico a uma atribuição e é chamado de **passagem por valor**.
- Desta forma, alterações nos parâmetros dentro da função **não alteram** os valores que foram passados.

```
1 void nao_troca(int x, int y) {  
2     int aux;  
3     aux = x;  
4     x = y;  
5     y = aux;  
6 }  
7  
8 void main(){  
9     int x=4, y=5;  
10    nao_troca(x,y);  
11 }
```

Passagem de argumentos por referência

- Em C só existe passagem de parâmetros por valor.
- Em algumas linguagens existem construções para se passar parâmetros por referência.
 - Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
- Algo semelhante pode ser obtido em C utilizando ponteiros:
 - Podemos passar para uma função o **endereço** da variável, e não o seu valor.

Passagem de argumentos por referência

- Em C só existe passagem de parâmetros por valor.
- Em algumas linguagens existem construções para se passar parâmetros por referência.
 - Neste último caso, alterações de um parâmetro passado por referência também ocorrem onde foi feita a chamada da função.
- Algo semelhante pode ser obtido em C utilizando ponteiros:
 - Podemos passar para uma função o **endereço** da variável, e não o seu valor.

Passagem de argumentos por referência

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

```
int f (int x, int *y) {  
    *y = x + 3;  
    return *y + 4;  
}
```

```
int main () {  
    int a, b, c;
```

```
    a = 5; b = 3;
```

```
    c = f (a, &b);
```

```
    printf("a = %d, b = %d, c = %d\n", a, b, c);
```

```
    return 0;
```

```
}
```

- Vetores são estruturas indexadas utilizadas para armazenar dados de um mesmo tipo: `int`, `char`, `float` ou `double`.
- Como um vetor fica armazenado na memória? A organização das variáveis na memória depende de como o sistema operacional faz gerenciamento da memória. Em geral, para ser mais eficiente, o sistema operacional tende a colocar as variáveis sucessivamente.
- **A implementação de vetores em C está bastante interligada com a de ponteiros visando a facilitar a manipulação de vetores.**

Vetores e ponteiros

- Considere a seguinte declaração de variáveis:

```
int v[80];
```

```
int *p;
```

- Podemos utilizar a sintaxe "normal" para fazer um ponteiro apontar para uma casa do vetor:

```
p = &v[2];
```

- Mas podemos utilizar a **sintaxe especial para ponteiros e vetores**, junto com as operações para ponteiros:

- Podemos fazer um ponteiro apontar para o início do vetor v fazendo

```
p = v.
```

- O comando acima equivale a fazer `p = &v[0]`.

Vetores e ponteiros

- Considere a seguinte declaração de variáveis:

```
int v[80];  
int *p;
```

- Podemos utilizar a sintaxe "normal" para fazer um ponteiro apontar para uma casa do vetor:

```
p = &v[2];
```

- Mas podemos utilizar a **sintaxe especial para ponteiros e vetores**, junto com as operações para ponteiros:

- Podemos fazer um ponteiro apontar para o início do vetor `v` fazendo

```
p = v.
```

- O comando acima equivale a fazer `p = &v[0]`.

Vetores e ponteiros

- Se fizermos $p = v$, podemos acessar o elemento que está na casa i de v fazendo $p[i]$, ou seja, ambos $p[i]$ e $v[i]$ acessam a casa i do vetor v .

```
1 #include <stdio.h>
2
3 void main(){
4
5     int v[5] = {1, 2, 3, 4, 5};
6     int *p;
7     p = v;
8     printf("%d %d\n", p[1], v[1]);
9 }
```

```
$ gcc vetor.c -o vetor
$ ./vetor
2 2
```

Vetores como parâmetro de funções

```
# include <math.h>
float modulo ( float v[], int n) {
    int i ;
    float r = 0;
    for (i=0; i<n; i++) {
        r = r + v[i]*v[i];
    }
    r = sqrt (r);
    return r;
}
```

```
float modulo ( float *p , int n) {  
    int i ;  
    float r = 0;  
    for (i=0; i<n; i++) {  
        r = r + p[i]*p[i];  
    }  
    r = sqrt (r);  
    return r;  
}
```

Ponteiros para registros

- Uma variável struct é armazenada na memória como qualquer outro variável, e portanto, possui um endereço.
- É possível, então, criar um ponteiro para uma variável de um tipo struct!

Ponteiros para registros

```
1 #include <stdio.h>
2
3 typedef struct { float x, y;} Coordenada;
4
5 void main(){
6
7     Coordenada c1, c2, *c3, *c4;
8     c3 = &c1;
9     c4 = &c2;
10
11     (*c3).x = 1.5; // mesmo efeito que c1.x = 1.5
12     (*c3).y = 1.5;
13     c4->x = 2.5; // mesmo efeito que (*c4).x = 2.5
14     c4->y = 2.5;
15
16     printf("%.2f\n", c4->x);
17     printf("%.2f\n", (*c3).y);
18 }
```

Operações com ponteiros

- Considere a seguinte declaração:

```
int vi[5] = {1, 2, 3, 4, 5};  
int *pi, *pi2, num;  
float vf[5] = {1.1, 2.2, 3.3, 4.4, 5.5};  
float *pf;
```

Quando somamos 1 a um ponteiro para **int** (por exemplo, **pi**) ele passa a apontar para o endereço de memória logo após a memória reservada para este inteiro. Exemplo, se **pi = &vi[4]**, então **pi+1** é o endereço de **vi[5]**, **pi+2** é o endereço de **vi[6]**.

Operações com ponteiros

- Considere a seguinte declaração:

```
int vi[5] = {1, 2, 3, 4, 5};  
int *pi, *pi2, num;  
float vf[5] = {1.1, 2.2, 3.3, 4.4, 5.5};  
float *pf;
```

Se somamos 1 a um ponteiro para **float** (por exemplo pf) ele avança para o endereço após este float. Por exemplo, se **pf=&vf[3]**, então **pf+1** é o endereço de **vf[4]**.

Operações com ponteiros

- Considere a seguinte declaração:

```
int vi[5] = {1, 2, 3, 4, 5};  
int *pi, *pi2, num;  
float vf[5] = {1.1, 2.2, 3.3, 4.4, 5.5};  
float *pf;
```

Somar ou subtrair um inteiro de um ponteiro:

```
pi = &vi[2];  
pi = pi-1;
```


Operações com ponteiros

- Considere a seguinte declaração:

```
int vi[5] = {1, 2, 3, 4, 5};  
int *pi, *pi2, num;  
float vf[5] = {1.1, 2.2, 3.3, 4.4, 5.5};  
float *pf;
```

Subtrair dois ponteiros:

```
pi = &vi[2];  
pi2 = &vi[3];  
num = pi2 - pi; //diferença entre os índices
```

Comparações entre ponteiros

- Testes relacionais $>$, $<$, $>=$, $<=$, $==$ ou $!=$ são aceitos somente entre ponteiros do mesmo tipo.
- Quando um ponteiro não está associado com nenhum endereço válido é comum atribuir o valor **NULL** para este.
- **NULL** é usado também para saber se um determinado ponteiro possui valor válido ou não.

EXERCÍCIOS

Exercício

■ O que será impresso?

```
1 #include <stdio.h>
2 void main(){
3     int a=3, b=2, *p = NULL, *q = NULL;
4     p = &a;
5     q = p;
6     *q = *q + 1;
7     q = &b;
8     b = b + 1;
9     printf("%d\n", *q);
10    printf("%d\n", *p);
11 }
```

Exercício

■ O que faz este código?

```
1 #include <stdio.h>
2 #define MAX 5
3
4 void main(){
5     int i, *p, s=0;
6     int a[MAX] = {1, 2, 3, 4, 5};
7
8     for(p = a; p < &a[MAX]; ++p)
9         s += *p;
10
11    for(i = 0; i<MAX; ++i)
12        s += *(a+i);
13
14    p = a;
15    for(i = 0; i<MAX; ++i)
16        s += p[i];
17 }
```

Algumas referências...

- Treinamento em Linguagem C. Victorine Viviane Mazrahi. Capítulo 9.
- <http://www.ic.unicamp.br/~mc102>
- <http://www.ime.usp.br/~pf/algoritmos/aulas/pont.html>
- <http://www.ime.usp.br/~hitoshi/introducao/17-funcao03.pdf>