

Estrutura de Dados

Ponteiros

Versão: 0.2

Data: março de 2021

Autor

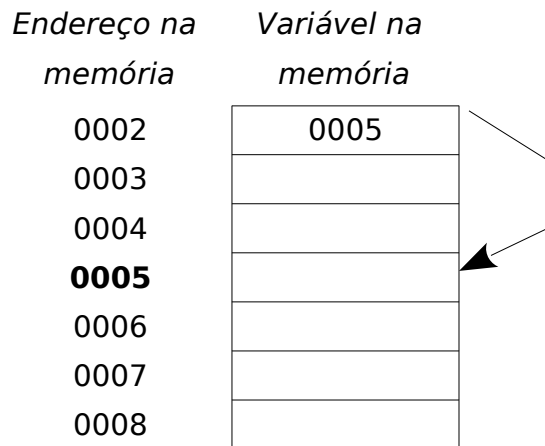
Oclair Prado

1. Ponteiros (ou apontadores)

O correto entendimento e uso de ponteiros é crítico para programação bem-sucedida em C.

Ponteiros são um dos aspectos mais fortes e mais perigosos de C.

Um ponteiro é uma variável que contém um endereço de memória. Esse endereço é normalmente a posição de uma outra variável na memória. Um apontador aponta para uma outra variável.



Uma declaração de ponteiro consiste no tipo de base, um * e o nome da variável.

A forma geral para declarar uma variável é

```
tipo *nome
```

Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória. Porém, toda a aritmética de ponteiros é feita por meio do tipo base, assim, é importante declarar o ponteiro corretamente.

2. Operadores de Ponteiros

Existem dois operadores especiais para ponteiros: * e &.

O operador & é um operador unário que devolve o endereço na memória do seu operando. Por exemplo:

```
m = &count; //m recebe o endereço de count
```

m recebe o endereço na memória que contém a variável count. O endereço não tem relação alguma com o valor de count.

O operador & pode ser imaginado como retornando “o endereço de”. Assim, o comando de atribuição anterior significa “m recebe o endereço de count”.

O segundo operador de ponteiro é o *. Ele é um operador unário que devolve o valor da variável que o segue. Por exemplo:

```
q = *m; //q recebe o valor que está no endereço m.
```

`q` recebe o valor da área de memória apontada por `m`. Imaginando que `count` contenha o valor 5, então `q` recebeu o valor 5.

O operador `*` pode ser imaginado como **“no endereço”**. O exemplo anterior pode ser lido como: `q` recebe o valor que está **no endereço** `m`.

Exemplo errado:

```
#include <stdio.h>

int main(){
    float x, y;
    int *p;
    p = &x
    y = *p //Só 2 bytes são transferidos e não os 8 do tipo float.
}
```

Exemplo correto:

```
#include <stdio.h>

int main(){
    float x, y;
    float *p;
    p = &x;
    y = *p //Agora todos os bytes serão transferidos.
}
```

Exercícios:

Exercício exPont1: Elabore um programa que leia um valor inteiro do teclado e depois atribua o endereço desta variável a um apontador. Ao final do processamento imprima o valor lido usando o apontador e não a variável com a função `printf()`.

3. Aritmética de Ponteiros

Existem apenas duas operações que podem ser usadas com ponteiros:

adição e subtração

Considerando que o tipo inteiro ocupa 2 bytes, então se `p` for um ponteiro para inteiro contendo o endereço 2000, e o programa executar a instrução `p++` ele passa a conter o endereço 2002, ou seja, ele aponta para o próximo inteiro.

O mesmo raciocínio vale para decremento.

Exemplo: Neste exemplo está sendo considerado que os valores inteiros ocupam 2 bytes da memória.

```
...  
int v[5];  
int *p;  
...//Carga do vetor  
p = v;  
p++  
...
```

A figura a seguir representa a região da memória do vetor `v` que foi declarado com 5 posições de inteiros.

<i>Endereço na memória</i>	<i>Variável na memória</i>
0010	54
0012	375
0014	2
0016	65
0018	7

Como o apontador `p` foi declarado como inteiro, após a instrução `p++` ele contém o endereço 0012, ou seja, ele aponta para a segunda posição do vetor que por sua vez contém o valor 375.

Este resultado se aplica a qualquer tipo de variáveis mesmo para as estruturas complexas definidas pelo usuário com a instrução `struct`.

Exercícios:

Exercício exPont2: Elabore um programa que leia um vetor de 5 inteiros do teclado e depois o imprima com a ajuda de um apontador para percorrer todas as suas posições.

4. Alocação dinâmica de memória

Ponteiros fornecem o suporte necessário para o poderoso sistema de alocação dinâmica de C.

O coração do sistema de alocação dinâmica de C consiste nas funções `malloc()` e `free()` da biblioteca **`stdlib.h`**. Na verdade, estas são as duas mais importantes. Existem diversas outras funções de alocação dinâmica de memória em C.

O protótipo da função `malloc()` é:

```
void *malloc(size_t número_de_bytes);
```

Ela devolve um ponteiro do tipo void, isto significa que ela pode ser usada com qualquer tipo de ponteiro. Em caso de falha ela devolve um nulo.

Exemplo:

```
...  
char *p;  
p = malloc(10 * sizeof(char));  
...
```

O fragmento de código anterior é equivalente a:

```
...  
char *p = "";  
...
```

Como a memória é finita, deve-se sempre testar o resultado de malloc().

Corrigindo o exemplo anterior temos:

```
...  
char *p;  
if( ! (p=malloc(10 * sizeof(char))) ){  
    printf("Sem memoria!\n");  
    exit(1);  
}  
...
```

Como boa prática de programação devemos sempre limpar a área de memória alocada.

Reaproveitando o exemplo anterior temos:

```
...  
char *p;  
if( ! (p=malloc(10 * sizeof(char))) ){  
    printf("Sem memoria!\n");  
    exit(1);  
}  
memset(p, 0, 10 * sizeof(char));  
...
```

O fragmento de código anterior é equivalente a:

```
...
char *p = "0000000000";
...
```

A função `free()` é o oposto de `malloc()`. Ela devolve para o sistema a memória que estava alocada.

O protótipo da função `free()` é:

```
void free(void *p);
```

Onde `p` é um ponteiro para uma área de memória alocada anteriormente com a função `malloc()`. Um argumento errado para `free()` pode **destruir** toda a lista de memória livre.

Exemplo:

```
...
char *p;
if( ! (p=malloc(10 * sizeof(char))) ){
    printf("Sem memoria!\n");
    exit(1);
}
memset(p, 0, 10 * sizeof(char));
...
free(p);
...
```

Exercício exPont5: Elabore um programa para criar duas variáveis usando apontadores do tipo inteiro. Solicite seus valores ao usuário e em seguida imprima seu conteúdo. Ao final do processamento libere a memória que você alocou.