

NOTAS DE AULA 08 – Ponteiros

1. Antes dos Ponteiros: Variáveis e a Memória

Antes de tentar compreender o que é um ponteiro, o aluno precisa entender o que é, exatamente, uma variável.

Quando se estuda arquitetura e organização de computadores, compreendemos que o computador é composto de CPU, memória e unidades de entrada e saída. Qualquer informação que precise ser armazenada, será colocada na memória.

Simplificadamente, cada byte da memória é chamado de **posição de memória**. Algumas informações cabem em uma única posição de memória (como uma letra, do tipo de dado **char**), outras exigem várias posições de memória (como um **int** que, por ter 32 bits, exige 4 posições de memória).

Cada posição de memória tem um **endereço**, o chamado **endereço de memória**. O primeiro endereço de memória é o **0** (zero), o segundo é o **1**, o terceiro é o **2...** e assim por diante.

Quando declaramos uma variável do tipo **char**, por exemplo, na verdade solicitamos que o computador **reserve uma posição de memória**, para que possamos guardar uma letra na memória. Essa posição de memória, é claro, possui um **endereço**, que fica associado ao **nome da variável**, de maneira que não precisamos saber o endereço numérico, já que podemos nos referir a ele pelo **nome da variável**.

Assim, quando escrevemos esse código:

```
char   letra;
```

```
letra = 'a';
```

No fundo estamos dizendo ao computador que “reserve um byte na memória e dê o nome a essa posição de ‘letra’”... e depois pedimos que ele guarde o valor referente à letra ‘a’ nessa posição de memória.

No caso dos valores inteiros, para citar outro exemplo:

```
int    num;
```

```
num = 10;
```

No fundo estamos dizendo ao computador que “reserve 4 bytes (espaço para um inteiro) e dê o nome de ‘num’ à posição do primeiro desses quatro bytes”... e depois disso solicitamos que o computador armazene o valor ‘10’ nessas posições de memória.

Se quisermos saber o endereço de memória em que uma variável foi armazenada, podemos usar o operador &:

```
int num;  
cout << &num;
```

Execute esse código e veja que número estranho aparece. Esse “número estranho” é o endereço de memória em que a variável foi alocada.

Nota: Em Python conseguimos ver o endereço de uma variável com o comando `id()`. Por exemplo, para saber em que endereço está armazenada a variável `K`, usamos `print(id(K))`.

Agora, experimente o código a seguir:

```
int num1, num2;  
cout << &num1 << endl;  
cout << &num2;
```

Observe que o endereço de `num2` é exatamente 4 posições após `num1`. Isso ocorre porque, como já dito, variáveis inteiras ocupam 4 bytes (ou 4 posições de memória).

Se quisermos saber quantas posições de memória uma variável ocupa, podemos usar a função `sizeof()`. Experimente o código abaixo para descobrir quantos bytes uma variável do tipo float e uma do tipo double ocupam:

```
cout << sizeof(float);  
cout << sizeof(double);  
cout << sizeof(produto); // Pressupõe que a estrutura produto foi declarada!
```

Note, porém, que não é “direto” guardar o endereço de uma variável em outra variável. O exemplo abaixo, por exemplo, irá causar algumas reclamações por parte do compilador:

```
int i, end;  
end = &i;
```

2. Enfim, os Ponteiros

Embora nem sempre seja a única solução, algumas vezes pode ser útil trabalhar diretamente com os endereços de memória. Um exemplo clássico é quando criamos uma função que retorna uma resposta de tamanho não definido previamente (como um texto).

Para passarmos uma variável por referência (para que o texto fosse colocado nela) precisaríamos saber, com antecedência, o tamanho máximo do texto, mas isso nem sempre é possível. O que fazer?

Atualização: 14/10/2020 – Prof. Daniel Caetano

Bem, a função que vai “devolver” o texto poderia devolver apenas o endereço onde se encontra o texto. E como saber onde começa e termina o texto? Simples: ele começa no endereço retornado e vai até o byte com valor ‘\0’ (o byte de terminação é uma convenção).

Já que é útil, como fazemos para lidar direto com os endereços? Simples: vamos declarar um tipo especial de variável chamado “ponteiro”, o que pode ser feito com um * antes do nome da variável:

```
int *end;
```

Essa declaração permite que eu armazene o **endereço de um inteiro** na variável de nome “end”.

Observe o código (e o execute para ver o resultado):

```
int i, *end;  
end = &i;  
cout << &i;  
cout << end;
```

Observe que o tipo de dado é importante para os ponteiros. O código abaixo dá erro:

```
int *a;  
char b;  
a = &b;
```

Não podemos guardar o endereço de um **char** em um ponteiro para inteiros.

Ok, interessante... temos uma variável que guarda um endereço de memória... mas como ler o valor lá armazenado? Simples: usamos o operador * conforme indicado:

```
int *a, b;  
b = 10;  
a = &b;  
cout << &b;    // Imprime o endereço de 'b'  
cout << b;     // Imprime o valor de 'b'  
cout << a;     // Imprime o endereço de 'b' (armazenado na variável 'a')  
cout << *a;    // Imprime o valor de 'b' (o valor armazenado no endereço apontado por a)
```

O nome dessas variáveis é **ponteiro** porque elas **apontam uma posição de memória**. Para saber o valor armazenado no endereço apontado, é preciso usar o operador de desreferenciamento *.

3. Ponteiros para Estruturas

É possível criar ponteiros para estruturas, sem problemas. Considere a estrutura abaixo:

```
struct produto {  
    int id;  
    char nome[60];  
    float preco;  
};
```

Para criar um ponteiro para ela do mesmo jeito que já vimos para os tipos de dados básicos:

```
produto *p;  
produto q;  
p = &q;
```

A grande diferença, nesse caso, é como fazemos para acessar os valores da estrutura. Quando usamos o ponteiro, temos de desreferenciá-lo. Observe no código a seguir.

```
produto *p;  
produto q;  
p.id = 10;  
p = &q;  
cout << p.id;  
cout << (*q).id;
```

É muito importante colocar o (*q) entre parênteses, caso contrário não irá funcionar corretamente. Para evitar confusão, entretanto, existe um “substituto” para o “.” Que vale para o caso de ponteiro, que é o -> :

```
produto *p;  
produto q;  
p.id = 10;  
p = &q;  
cout << p.id;  
cout << (*q).id;  
cout << q->id;           // q->id Substitui a forma (*q).id
```